

# Ohjelmistotuotanto Kevät 2017

Matti Luukkainen

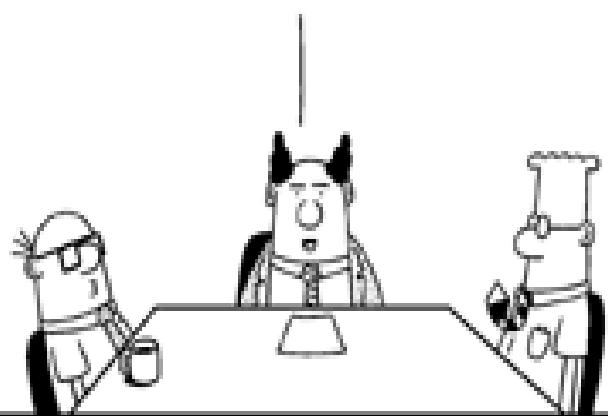
assistentteina:

Juho Lamminmäki, Juha-Pekka Moilanen ja Tuomo  
Torppa

Luento 1

13.3.2017

WE'RE GOING TO  
TRY SOMETHING  
CALLED AGILE  
PROGRAMMING.



scottadams@scot.com

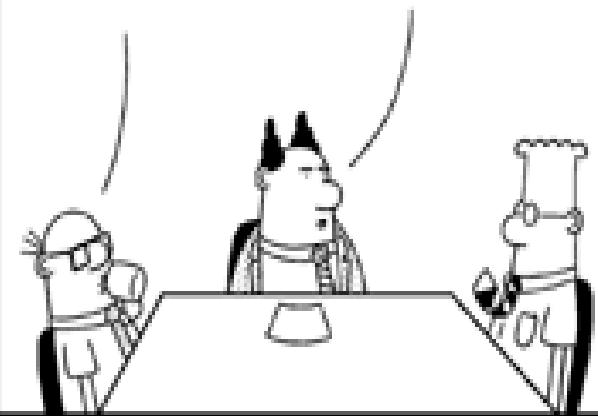
[www.dilbert.com](http://www.dilbert.com)

THAT MEANS NO MORE  
PLANNING AND NO MORE  
DOCUMENTATION. JUST  
START WRITING CODE  
AND COMPLAINING.



© 2007 Scott Adams, Inc./Dist. by UFS, Inc.

I'M GLAD  
IT HAS A  
NAME.



THAT  
WAS YOUR  
TRAINING.

# Kurssin tavoite

- Primääriinen tavoite on antaa osallistujille riittävät käsitteelliset ja tekniset valmiudet toimia *Ohjelmistotuotantoprojektissa*
- Suoritettuaan kurssin opiskelija
  - Tuntee ohjelmistoprosessin, erityisesti ketterän prosessin vaiheet
  - Tietää miten vaatimuksia hallitaan ketterässä ohjelmistotuotantoprosessissa
  - Ymmärtää suunnittelun, toteutuksen ja testauksen vastuut ja luonteen ketterässä ohjelmistotuotannossa
  - Ymmärtää ohjelmiston laadunhallinnan perusteet
  - Osaa toimia ympäristössä, jossa ohjelmistokehitys tapahtuu hallitusti ja toistettavalla tavalla
- Oppimismatriisi ei ole tällä hetkellä ajan tasalla
  - Matriiseista luovutaan syksyllä, joten sitä ei tulla edes päivittämään
- Aihepiirin hallitseville ihmisille suuri tarve, ks:
  - <http://www.projectmanagement.com/blog/Agility-and-Project-Leadership/6293/>

# Sisältö ja kurssimateriaali

- Sisältö ks kurssisivu  
<https://github.com/mluukkai/ohtu2017/wiki/Ohjelmistotuotanto-2017>
- "teoria" perustuu mm. seuraaviin lähteisiin
  - Henrik Kniberg: Scrum and XP from the trenches (ilmainen pdf)
  - James Shore: The Art of Agile development (osittain online)
  - Jonathan Rasmusson: The Agile Samurai
- Oleelliset luvut tullaan listaamaan kurssisivulla
- Näiden lisäksi paljon web-lähteitä, jotka myös tullaan mainitsemaan kurssisivulla
- Teoria-asiaa tulee myös laskaritehtävien yhteydessä
- **HUOM: pelkästään luentokalvoja lukemalla ei esim. kurssikokeessa tule pärjäämään kovin hyvin**

# Opetus ja suoritustapa

- **Luentoja** 2\*2h viikossa, ma 14-16 ja ti 12-14
  - Viikon 1 tiistain luento poikkeuksellisesti klo 14-16
  - Viikoilla 5 ja 6 pidetään ainoastaan maanantain luento
  - Viikolla 7 ei todennäköisesti luentoa
- **Laskarit:**
  - Ohjelmointi/versionhallinta/konfigurointitehtäviä
  - Laskareista yhteensä 10 kurssipistettä
- **Miniprojekti** viikoilla 3-6
  - Tehdään 3-5 hengen ryhmissä
  - yhteensä 10 kurssipistettä
- **Koe** yhteensä 20 kurssipistettä
- **Läpipääsyyn vaaditaan hyväksytty miniprojekti, puolet koepistemäärästä ja puolet koko kurssin pistemäärästä**

# Laskarit, ekstranopat

- **Laskarit**
  - Viikon tehtävien deadline sunnuntai klo 23.59
  - Ohjausajat kurssisivulla
  - Tehtävien palautus: ks. ensimmäinen laskari
- **Ylimääräiset opintopisteet**
  - **Versionhallinta 1 op:** jos et ole tehnyt kurssia, saat kurssin suoritetuksi tekemällä **kaikki** ohton versiohallintatehtävät ja suorittamalla hyväksytysti miniprojektiin
  - Tekemällä 90% kurssin muista kuin versionhallintaa käsittelevistä laskaritehtävistä, saat kurssista normaalilin viiden opintopisteen sijaan **kuusi opintopistettä**

# Miniprojekti

- **Viikolta 3 alkaen kurssilla siis ”miniprojekti”**
- **Kurssin läpäisyyn edellytyksenä on hyväksytysti suoritettu miniprojekti**
- miniprojektissa ohjelmoidaan hiukan, mutta pääosassa on ohjelmistoprosessin kurinalainen noudattaminen
- Projektit tehdään 3-5 hengen ryhmissä. Projektit tapaa asiakkaan viikoilla 3-6
  - Kaikkien ryhmäläisten tulee olla asiakastapaamisissa (30-60 min) paikalla
- Viimeisellä viikolla miniprojektien demotilaisuus (2h)
- Lisää miniprojektista parin viikon päästä
- Miniprojektiin osallistuminen ei ole välttämätöntä jos täytät työkokemuksen perusteella tapahtuvan Ohjelmistotuotantoprojektiin hyväksiluvun edellyttävät kriteerit ks. kohta ”Laaja suoritus” sivulta <http://www.cs.helsinki.fi/opiskelu/tietotekniikka-alan-ty-kokemus-opintosuorituksena>
  - jos ”hyväksiluet” miniprojektiin työkokemuksella, kerro asiasta välittömästi emailitse

# Luennot – laskarit - miniprojekti

- Kurssin luennolla keskitytään pääosin ohjelmistokehityksen teoriaan
- Laskarit taas ovat luonteeltaan melko tekniset sisältäen paljon versionhallintaa, ohjelmistojen konfigurointia, testausta ja ohjelmointia
- Osaa luentojen teoriasta ei käsitellä laskareissa ollenkaan, ja vastaavasti osaa laskareiden teknisemmistä asioista ei käsitellä luennolla
- Miniprojektiin ideana on yhdistää luentojen teoria ja laskareissa käsitellyt teknisemmät asiat, ja soveltaa niitä käytännössä pienessä ohjelmistoprojektissa
- Kokeessa suurin paino tulee olemaan teoriassa ja sen soveltamisessa käytäntöön
  - Laskareiden teknisimpiä asioita, kuten versionhallintaa ei kokeessa tulla kysymään
  - Tarkemmin kokeesta ja siihen valmistautumisesta kurssin viimeisellä luennolla

# Ohjelmistotuotanto engl. Software engineering

- The IEEE Computer Society defines software engineering as:  
**"The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software".**
- Lähde SWEBOK eli Guide to the Software Engineering Body of Knowledge <http://www.swebok.org>
- Mikä on SWEBOK:
  - Ison komitean yritys määritellä mitä ohjelmistotuotannolla tarkoitetaan ja mitä osa-alueita siihen kuuluu
  - Uusin versio vuodelta 2004 eli paikoin jo vanhentunut

# Ohjelmistotuotannon osa-alueet

- SWEBOK:in mukaan ohjelmistotuotanto jakautuu seuraaviin osa-alueisiin:
  - Software requirements eli *vaatimusmäärittely*
  - Software design eli *suunnittelu*
  - Software construction eli *toteutus/ohjelmointi*
  - Software testing
  - Software maintenance eli *ylläpito*
  - Software configuration management
  - Software engineering management
  - Software engineering process eli *ohjelmistotuotantoprosessi*
  - Software engineering tools and methods
  - Software quality
- Näiden osa-alueiden eritasoinen läpikäynti on myöskin tämän kurssin tavoite

# Ohjelmiston elinkaari (software lifecycle)

- Riippumatta tyylistä ja tavasta, jolla ohjelmisto tehdään, käy ohjelmisto läpi seuraavat vaiheet
  - Vaatimusten analysointi ja määrittely
  - Suunnittelu
  - Toteutus
  - Testaus
  - Ohjelmiston ylläpito ja evoluutio
- Eri vaiheiden sisältöön palaamme myöhemmin tarkemmin, jos asia on unohtunut, kertaa esim. OTM:n materiaalista
- Miten ja kenen toimesta vaiheet on suoritettu, on vaihdellut aikojen saatossa

# Alussa (ja osin edelleen) code'n'fix

- Tietokoneiden historian alkuaikoina laitteet maksoivat paljon, ohjelmat olivat laitteistoihin nähdyn "triviaaleja"
  - Ohjelmointi tapahtui konekielellä
  - Usein sovelluksen käyttäjä ohjelmoi itse ohjelmansa
- Vähitellen ohjelmistot alkavat kasvaa ja kehitettiin korkeamman tason ohjelmointikieliä (Fortran, Cobol, Algol)
- Pikkuhiljaa homma alkaa karata käsistä (wikipediaan ohjelmistotuotanoon historiaa käsitleväästä artikkelista):
  - Projects running over-budget
  - Projects running over-time
  - Software was very inefficient
  - Software was of low quality
  - Software often did not meet requirements
  - Projects were unmanageable and code difficult to maintain
  - Software was never delivered

# Kriisi

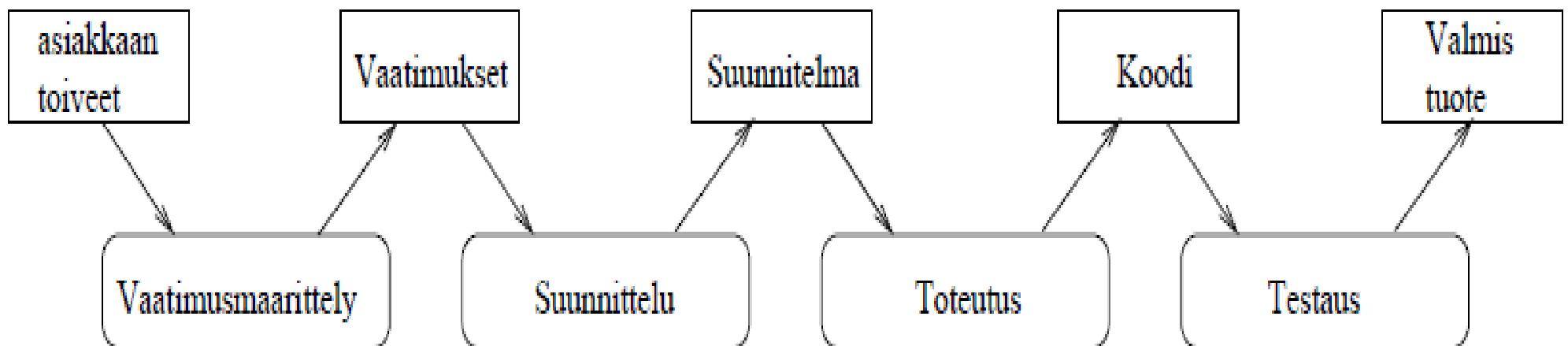
- Termi **Software crisis** lanseerataan kesällä 1968
  - The term was used to describe the impact of rapid increases in computer power and the complexity of the problems that could be tackled. In essence, it refers to **the difficulty of writing correct, understandable, and verifiable computer programs**. The roots of the software crisis are complexity, expectations, and change.
- Edsger Dijkstra:
  - The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.
- [http://en.wikipedia.org/wiki/Software\\_crisis](http://en.wikipedia.org/wiki/Software_crisis)

# Software development as Engineering

- Termi Software engineering määritellään ensimmäistä kertaa 1968:
  - The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.
- Syntyy idea siitä, että ohjelmistojen tekemisen tulisi olla kuin mikä tahansa muu insinöörityö
- Eli kuten esim. siltojen rakentamisessa, tulee ensin rakennettava artefakti määritellä (requirements) ja suunnitella (design) aukottomasti, tämän jälkeen rakentaminen (construction) on triviaali vaihe

# Vesiputousmalli eli lineaarinen malli eli Plan based process tai Big Design Up Front

- Winston W. Royce: Management of the development of Large Software, 1970
  - [www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf](http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf)
- Artikkelin sivulla 2 Royce esittelee yksinkertaisen prosessimallin (eli ohjeiston työvaiheiden jaksottamiseen), jossa ohjelmiston elinkaaren vaiheet suoritetaan lineaarisesti peräkkäin:



# Vesiputousmalli eli lineaarinen malli eli Plan based process tai Big Design Up Front

- Paradoksaalista kyllä, Royce *ei suosittele* artikkelissaan suoraviivaisen lineaarisen mallin käyttöä, vaan esittelee mallin, jossa järjestelmästä tehdään ensin prototyppi ja lopullinen määrittely ja suunnittelu tehdään vasta prototyppiin perustuen
- Suoraviivainen lineaarinen malli, jota ruvettiin kutsumaan *vesiputousmalliksi*, saavutti nopeasti suosiota
  - Taustalla osittain se, että Yhdysvaltain puolustusministerö rupesi vaatimaan kaikilta alihankkijoiltaan prosessin noudattamista (Standardi DoD STD 2167)
  - Muutkin ohjelmistoja tuottaneet tahot ajattelivat, että koska DoD vaatii vesiputousmallia, on se hyvä asia ja tapa kannattaa omaksua itselleen

# Vesiputousmalli eli lineaarinen malli eli Plan based process tai Big Design Up Front

- Vesiputousmalli perustuu vahvasti siihen, että eri vaiheet ovat erillisten tuotantotiimien tekemiä
  - Tämän takia jokainen vaiheen tulokset dokumentoidaan tarkoin
  - Ohjelmisto suunnitellaan tyhjentävästi ennen ohjelmointivaiheen aloittamista eli tehdään "Big Design Up Front" (BDUF)
- Vesiputousmallin mukainen ohjelmistoprosessi on yleensä tarkkaan etukäteen suunniteltu, resursoitu ja aikataulutettu
  - tästä johtuu joskus käytetty nimike *plan based process*
- Vesiputousmallin mukainen ohjelmistotuotanto ei ole osoittautunut erityisen onnistuneeksi
- Jo vesiputousmallin "isä" Royce suositti artikkelissaan ohjelmien tekemistä kahdessa *iteraatiossa*
  - Roycen mukaan ensin kannattaa tehdä prototyppi ja vasta siitä saatujen kokemusten valossa suunnitellaan ja toteutetaan lopullinen ohjelmisto

# Lineaarisen mallin ongelmia

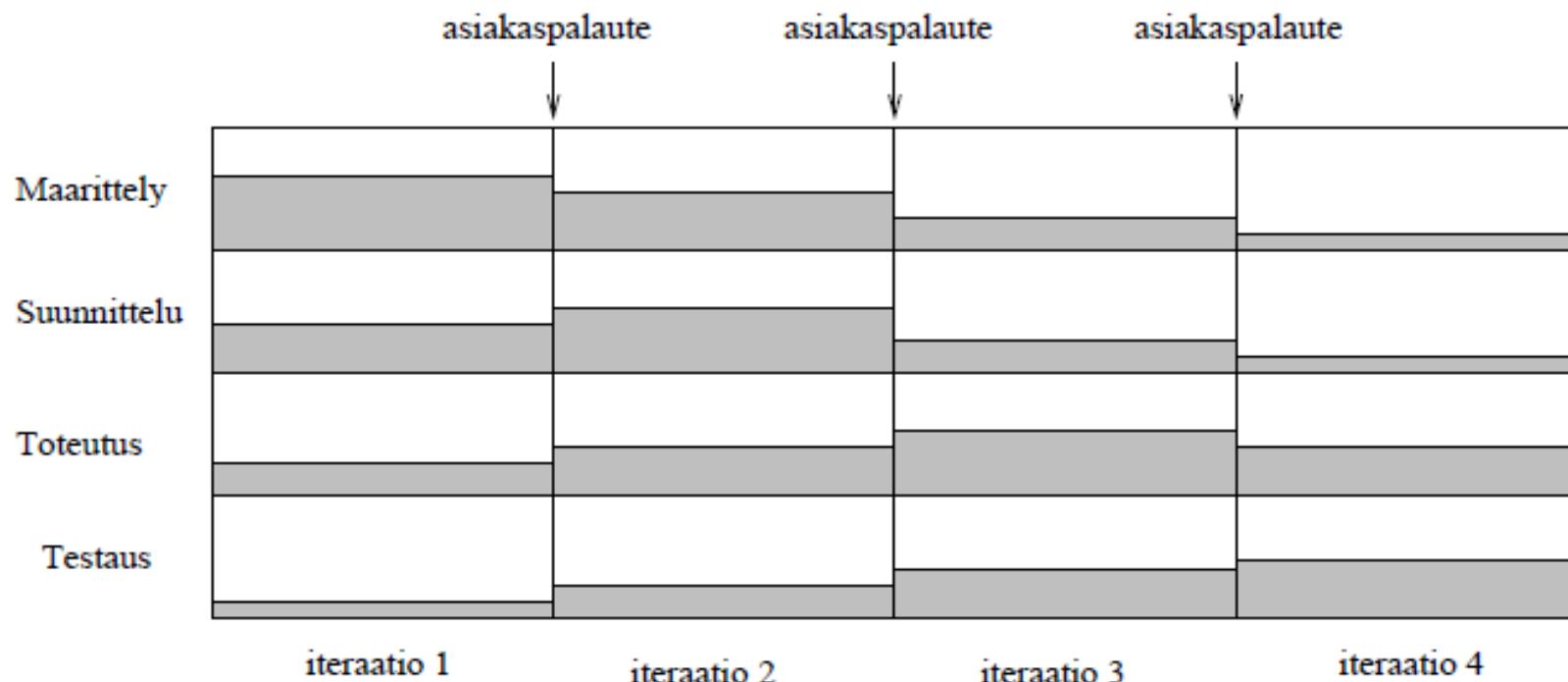
- Lineaarinen malli olettaa että ohjelmistotuotannon vaiheet tapahtuvat peräkkäin ja jokainen vaihe ainakin isoissa projekteissa eri ihmisten toimesta
- Vaatimukset kuitenkin usein muuttuvat matkan varrella:
  - Asiakas ei tiedä tai osaa sanoa mitä haluaa/tarvitsee
  - Asiakkaan tarve muuttuu projektin kuluessa
  - Asiakas alkaa haluta muutoksia kun näkee lopputuotteen
- Vaatimusmäärittelyn ja suunnittelun ja toteutuksen erottaminen on mahdotonta
  - Valittu arkkitehtiuri ja käytössä olevat toteutusteknologiat vaikuttavat suuresti määriteltyjen ominaisuuksien hintaan
  - Ohjelmaa on mahdotonta suunnitella siten, että toteutus on suoraviivaista, osa suunnittelusta tapahtuu pakosti vasta ohjelmointivaiheessa
- Vasta lopussa tapahtuva laadunhallinta paljastaa ongelmat liian myöhään
  - Vikojen korjaaminen tulee todella kalliaksi sillä testaus voi paljastaa ongelmia jotka pakottavat muuttamaan ohjelmiston vaatimuksia
- Martin Fowlerin artikkeli The New Methodology käsittelee laajalti lineaarisen mallin ongelmia
  - ks. <http://martinfowler.com/articles/newMethodology.html>

# Lineaarisen mallin ongelmia

- Kuten jo mainittiin, ohjelmistotuotannon takana on pitkälti analogia muihin insinööritieteisiin:
  - rakennettava artefakti tulee ensin määritellä ja suunnitella (design) aukottomasti, tämän jälkeen rakentaminen (construction) on triviaali vaihe
  - Perinteisesti ohjelointi on rinnastettu triviaalina pidettyyn "rakentamisvaiheeseen" ja kaiken haasteen on ajateltu olevan määrittelyssä ja suunnittelussa
    - Tätä rinnastusta on kuitenkin ruvettu kritisointaan, sillä ohjelmistojen suunnittelu sillä tarkkuudella, että suunnitelma voidaan muuttaa suoraviivaisesti koodiksi on osoittautunut mahdottomaksi
  - Onkin esitetty että perinteisen insinööritiedeanalogian triviaali rakennusvaihe ei ohjelmistoprosessissa olekaan ohjelointi, vaan ohjelmakoodin käänäminen eli että **ohjelmakoodi on itseasiassa ohjelmiston lopullinen suunnitelma** siinä mielessä kuin insinööritieteet käsittevät suunnittelun (design)
    - ks.  
<http://www.bleading-edge.com/Publications/C++Journal/Cpjour2.htm>

# Iteratiiviset prosessimallit

- Lineaarisen mallin ongelmiin reagoinut *iteratiivinen* tapa tehdä ohjelmistoja alkoi yleistyä 90-luvulla (mm. spiraalimalli, prototyppimalli, Rational Unified process)
- Iteratiivisessa mallissa ohjelmistotuotanto jaetaan jaksoihin, eli *iteraatioihin*
  - Jokaisen iteraation aikana määritellään, suunnitellaan toteutetaan ja testataan ohjelmista, eli ohjelmisto kehittyy vähitellen
  - Asiakasta tavataan jokaisen iteraation välissä, asiakas näkee sen hetkisen version ohjelmasta ja pystyy vaikuttamaan seuraavien iteraatioiden kulkuun



# Iteratiiviset prosessimallit

- Yhdysvaltojen puolustusministeriön vuonna 2000 julkaisema standardi (MIL-STD-498) alkaa suositella iteratiivista ohjelmistoprosessia:
  - "There are two approaches, evolutionary [iterative] and single step [waterfall], to full capability. **An evolutionary approach is preferred.** ... [In this] approach, the ultimate capability delivered to the user is divided into two or more blocks, with increasing increments of capability...software development shall follow an iterative spiral development process in which continually expanding software versions are based on learning from earlier development. It can also be done in phases"
- Itseasiassa iteratiivinen ohjelmistokehitys on paljon vanhempi idea kun lineaarinen malli
  - Esim. NASA:n ensimmäisen amerikkalaisen avaruuteen vieneen *Project Mercury*n ohjelmisto kehitettiin iteratiivisesti (50-luvun lopussa)
  - *Avaruussukkuloiden* ohjelmisto tehtiin vesiputousmallin valtakaudella 70-luvun lopussa, mutta sekin kehitettiin lopulta iteratiivista prosessia käyttänen (8 viikon iteraatioissa, 31 kuukauden aikana)
  - Lisää aiheesta osoitteesta <http://wiki.c2.com/?HistoryOfIterative>

# Ketterien menetelmien synty

- 1980- ja 1990-luvun prosessimalleissa korostettiin huolellista projektisuunnittelua, formaalia laadunvalvontaa, yksityiskohtaisia analyysi- ja suunnittelumenetelmiä ja täsmällistä, tarkasti ohjattua ohjelmistoprosessia
- Prosessimallit tukivat erityisesti laajojen, pitkäikäisten ohjelmistojen kehitystyötä, mutta pienien ja keskisuurten ohjelmistojen tekoon ne osoittautuivat usein turhan jäykiksi
- Perinteissä prosessimalleissa (myös iteratiivisissa) on pyritty työtä tekevän yksilön merkityksen minimoimiseen
  - Ajatuksena on ollut että yksilö on "tehdastyöläinen", joka voidaan helposti korvata toisella ja tällä ei ole ohjelmistoprosessin etenemiselle mitään vaikutusta
- Ristiriidan seurauksena syntyi joukko *ketteriä prosessimalleja* (agile process models), jotka korostivat itse ohjelmistoa sekä ohjelmiston asiakkaan ja toteuttajien merkitystä yksityiskohtaisen suunnittelun ja dokumentaation sijaan

# Agile manifesto 2001

- <http://agilemanifesto.org/>
- We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
  - **Individuals and interactions** over processes and tools
  - **Working software** over comprehensive documentation
  - **Customer collaboration** over contract negotiation
  - **Responding to change** over following a plan
- That is, while there is value in the items on the right, we value the items on the left more
- Manifestin laativat ja allekirjoittivat 17 ketterien menetelmien varhaista pioneeria, mm:
  - Kent Beck, Robert Martin, Ken Schwaber ja Martin Fowler
- Manifesti sisältää yllä olevan lisäksi 12 ketterää periaatetta, jotka on lueteltu seuraavilla sivuilla

# Ketterät periaatteet, osa 1

- Our highest priority is to satisfy the customer through **early and continuous delivery** of valuable software
- **Welcome changing requirements**, even late in development. Agile processes harness change for the customer's competitive advantage
- **Deliver working software frequently**, from a couple of weeks to a couple of months, with a preference to the shorter timescale
- Business people and developers **must work together** daily throughout the project.
- Build projects around **motivated individuals**. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is **face-to-face conversation**.

# Ketterät periaatteet, osa 2

- **Working software** is the primary **measure of progress**.
- Agile processes promote **sustainable development**. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to **technical excellence and good design** enhances agility.
- Simplicity – the art of **maximizing the amount of work not done** – is essential.
- The best architectures, requirements, and designs emerge from **self-organizing teams**.
- At regular intervals, the **team reflects on how to become more effective**, then tunes and adjusts its behavior accordingly.

# Ketterät menetelmät ja uusi metafora

- Ketterät menetelmät on sateenvarjotermi useille ketterille prosessimalleille
- Näistä tunnetuimpia ovat:
  - Extreme programming eli XP
  - Scrum
- Molempien, erityisesti Scrumiin tutustutaan kurssin aikana
- Ketterä ohjelmistotuotanto on ottanut vaikutteliaita myös Toyota production systemin taustalla olevasta *lean*-ajattelusta
- Viime aikoina on puhuttu paljon siitä, että ohjelmistojen tekemisen rinnastaminen perinteiseen insinöörityöhön eli koko termi *Software engineering* on metaforana väärä
- Ohjelmistojen tekemistä on alettu rinnastamaan käsityöläisyteen ja on syntynyt ns. **Software craftsmanship** "liike", jolla on jopa oma manifestinsä, ks.
  - <http://manifesto.softwarecraftsmanship.org/>

# Ohjelmistokehityksen työkalut

- Nyt on aika siirtyä teoriasta käytäntöön ja tarkastella alustavasti muutamaa ohjelmistokehityksen käytännön työkalua
  - Versionhallinta: git
  - Testaus: JUnit
  - Projektin riippuvuuksienhallinta ja käänäminen: gradle
  - CI- ja Build-palvelinohjelmisto: Travis-ci

# Versionhallinta – Git

- Versionhallinta välttämätön oikeastaan kaikissa projekteissa:
  - Koodi, dokumentaatio ym. löytyvät yksiselitteisestä paikasta, projektin "repositorystä"
  - Mahdollistaa tiedostojen rinnakkaisen editoinnin
    - Rinnakkainen editointi voi toki aiheuttaa *konflikteja* jos tiedoston samaa kohtaa editoidaan samaan aikaan usealta koneelta
  - Mahdollisuus palata historiassa taaksepäin
    - voidaan palauttaa tiedostosta sen edellisen päivän tilanne
  - Ohjelmistosta voi olla olemassa useita versioita yhtä aikaa
    - Voi olla esim. tarve tehdä bugikorjauksia johonkin ohjelman jo aiemmin julkaistuun versioon
- Oikeastaan yhden hengen pienäkään ohjelointiprojekteja ei ole järkevää tehdä ilman versionhallintaa
- Mitä versionhallintaan talletetaan?
  - Jos mahdollista, **kaikki ohjelmistoon liittyvä**: ohjelmakoodi, dokumentit, kirjastot, konfiguraatiotiedostot, jopa työkalut
- ks. [http://jamesshore.com/Agile-Book/version\\_control.html](http://jamesshore.com/Agile-Book/version_control.html)

# Versionhallinta – Git

- Kurssilla käytössä Git
  - Linus Torvaldsin kehittämä hajautettu versionhallinta
  - Tämän hetken eniten käytetty versionhallinta, syrjäyttänyt jo vuosia sitten vanhan ykkösen SVN:n
- Repositoriot talletetaan pääosin GitHub:iin
  - Internetissä oleva ”sosiaalinen” ohjelmistojen talletuspaikka
  - Ilmaiset repositoriot ovat koko maailmalle julkisia
  - Akateemisen ohjelman kautta mahdollista saada maksuttomia privaattirepositorioita
- Tutustumme Git:iin pikkuhiljaan laskareissa, muutama hyvä lähtökohta
  - <https://we.riseup.net/debian/git-development-howto>
  - <http://www.ralfebert.de/tutorials/git/>
  - <https://git-scm.com/book/en/v2>
- Git saattaa tuntua aluksi sekavalta. Peruskäyttö on kuitenkin hetken totuttelun jälkeen helppoa

# Testaus – JUnit

- Ohjelmiston kehittämisessä lähes tärkein vaihe on laadunvarmistus, laadunvarmistuksen tärkein keino taas on testaaminen
- Testaus on syytä automatisoida mahdollisimman pitkälle, sillä ohjelmistoja joudutaan testaamaan paljon, ja samat testit on erityisesti iteratiivisessa/ketterässä ohjelmistokehityksessä suoritettava uudelleen aina ohjelman muuttuessa
- xUnit-testauskehys on useille kielille saatavissa oleva lähinnä yksikkötestien automatisointiin tarkoitettu työkalu
  - Java-versio kehyksestä on nimeltään JUnit
- Tulemme tekemään automatisoitua testausta kurssin aikana paljon. Jos JUnit ei ole entuudestaan tuttu, kannattaa tutustumisen aloittaa välittömästi
  - <https://github.com/mluukkai/OTM2016/wiki/JUnit-ohje>

# Projektiin riippuvuuksien hallinta ja käänäminen – gradle

- ks. [http://jamesshore.com/Agile-Book/ten\\_minute\\_build.html](http://jamesshore.com/Agile-Book/ten_minute_build.html)
- Ohjelmistoprojektissa ohjelman käänäminen, testaaminen, paketointi suorituskelpoiseksi ja jopa ”deployaus” eli siirto testaus- tai tuotanto-ympäristöön tulee onnistua helposti
  - Kenen tahansa toimesta, miltä tahansa koneelta
  - ”nappia painamalla” tai yksi skripti ajamalla
- Ohjelmiston käänäminen edellyttää yleensä että ohjelman tarvitsemat kirjastot, eli ulkoiset riippuvuudet (Javassa yleensä Jar-tiedostoja) ovat käänösprosessin aikana saatavilla
- Riippuvuuksien hallinnan ja käänöksen suorittava skripti on käytännössä talletettava versionhallintaan ohjelmakoodin yhteyteen
- Hyvin toimivan käänös/testaus/paketointi-ympäristön konfigurointi ei ole välttämättä helppoa
- Aikojen saatossa on kehitetty asiaa helpottavia työkaluja
  - mm. make, Apache Ant, Maven
- Tällä kurssilla tutustumme Gradleen

# Projektiin riippuvuuksienhallinta ja käänäminen – Gradle

- Gradlen esittely projektin github-sivulta:
  - Gradle is a build tool with a focus on build automation and support for multi-language development. If you are building, testing, publishing, and deploying software on any platform, Gradle offers a flexible model that can support the entire development lifecycle from compiling and packaging code to publishing web sites. Gradle has been designed to support build automation across multiple languages and platforms including Java, Scala, Android, C/C++, and Groovy, and is closely integrated with development tools and continuous integration servers including Eclipse, IntelliJ, and Jenkins.
- Olet todennäköisesti käyttänyt Ohjelmoinnin harjoitystyössä "builtaustyökaluna" mavenia.
- Gradle on uuden generaation builtaustyökalu, jonka on tarkoitus korvata maven.
- Gradle toimii pitkälti samojen periaatteiden mukaan kuin maven, mutta on kuitenkin huomattavasti helpommin konfiguroitavissa ja myös nopeampi kuin edeltäjänsä.
- Tutustumme gradleen pikkuhiljaan laskareissa

# CI- ja Build-palvelinohjelmisto: Travis-ci

- Käännöksen automatisoinnin jälkeen seuraava askel on suorittaa käännösprosessi myös erillisillä **käännöspalvelimella** (build server)
- Ideana on, että ohjelmistokehittäjä noudattaa seuraavaa sykliä
  - Uusin versio koodista haetaan versionhallinnan keskitetystä repositoriosta ohjelmistokehittäjän työasemalle
  - Lisäykset ja niitä testaavat testit tehdään paikalliseen kopioon
  - Käännös ja testit ajetaan paikalliseen kopioon ohjelmistokehittäjän työasemalla
  - Jos kaikki on kunnossa, paikalliset muutokset lähetetään keskitettyyn repositorioon
  - Käännöspalvelin seuraa keskitettyä repositoriota ja kun siellä huomataan muutoksia, käänää käänöspalvelin koodin ja suorittaa sille testit
  - Käännöspalvelin raportoi havaitusta virheistä
- Erillisen käännöspalvelimen avulla varmistetaan, että ohjelmisto toimii muuallakin kuin muutokset tehneen ohjelmistokehittäjän koneella
- Kurssilla käytämme pilvessä toimivaa Travis-nimistä build-palvelinohjelmistoa
  - Keskitetyn build-palvelimen käyttöön liittyy käsite **jatkuva integraatio** (engl. Continuous integration), palaamme tähän tarkemmin myöhemmin kurssilla

Ohjelmistotuotanto

Luento 2

14.3.2017

# Ohjelmiston tuottaminen ei ole kontrolloitu prosessi

- Vesiputousmallin suurimmat ongelmat ovat seuraavat
  - Yleensä mahdotonta määritellä vaatimukset tyhjentävästi projektin alkuvaiheessa
    - Asiakas ei ymmärrä vielä alussa mitä haluaa
    - Bisnesympäristö muuttuu projektin kuluessa
  - Suunnittelu sillä tasolla, että ohjelointi on triviaali ja ennustettava rakennusvaihe, rinnastettavissa esim. talon rakennukseen, on mahdotonta
    - Ohjelointi on osa suunnitteluprosessia, ohjelmakoodi on tuotteen lopullinen suunnitelma
    - Suunnittelu on teknisesti haastavaa ja riskejä sisältävää toimintaa
  - 90-luvun iteratiiviset prosessimallit korjaavat monia näistä epäkohdista
  - Kuitenkin 90-luvun mallit olivat vielä vahvasti "plan-based" ja olettivat että ohjelmistotuotanto on jossain määrin kontrolloitavissa oleva prosessi
    - Tarkka projektisuunnitelma ja sen noudattaminen
    - Selkeä roolijako: projektipäälliköt, analyyttikot, arkkitehdit, ohjelmoijat, testaajat

# Ketterien menetelmien perusolettamuksia

- Useimmat ohjelmistoprojektit ovat laadultaan uniikkeja
  - Vaatimukset erilaiset kuin millään jo tehdyllä ohjelmistolla
  - Uusi tekijätähti, omanlaisilla kompetensseilla ja persoonallisuksilla varustettu
  - Toteutusteknologiat kehittyvät, joten tehdään todennäköisesti tavalla, joka ei ole kaikille tunnettu
- Järkevää lähteä oletuksesta että kyseessä ei ole kontrolloitu prosessi, joka voidaan tarkkaan etukäteen suunnitella (eli ei "plan-based")
- Parempi ajatella *tuotekehitysprojektina*, näiden kontrollointiin sopii paremmin ns. "empiriinen prosessi"
  - Toiminnan periaatteina *transparency, inspection, adaption*
- Tekijät yksilöitä, oletus että yksilöt toimivat paremmin kun heihin luotetaan ja annetaan tiimille vapaus organisoida itse toimintansa
  - "The whole team"-periaate: tiimi kollektiivina vastuussa aikaansaannoksesta
  - Oletuksena että perinteinen command-and-control ja jako eri vastuualueisiin (suunnittelija, ohjelmoija, testaaja) ei tuota optimaalista tulosta

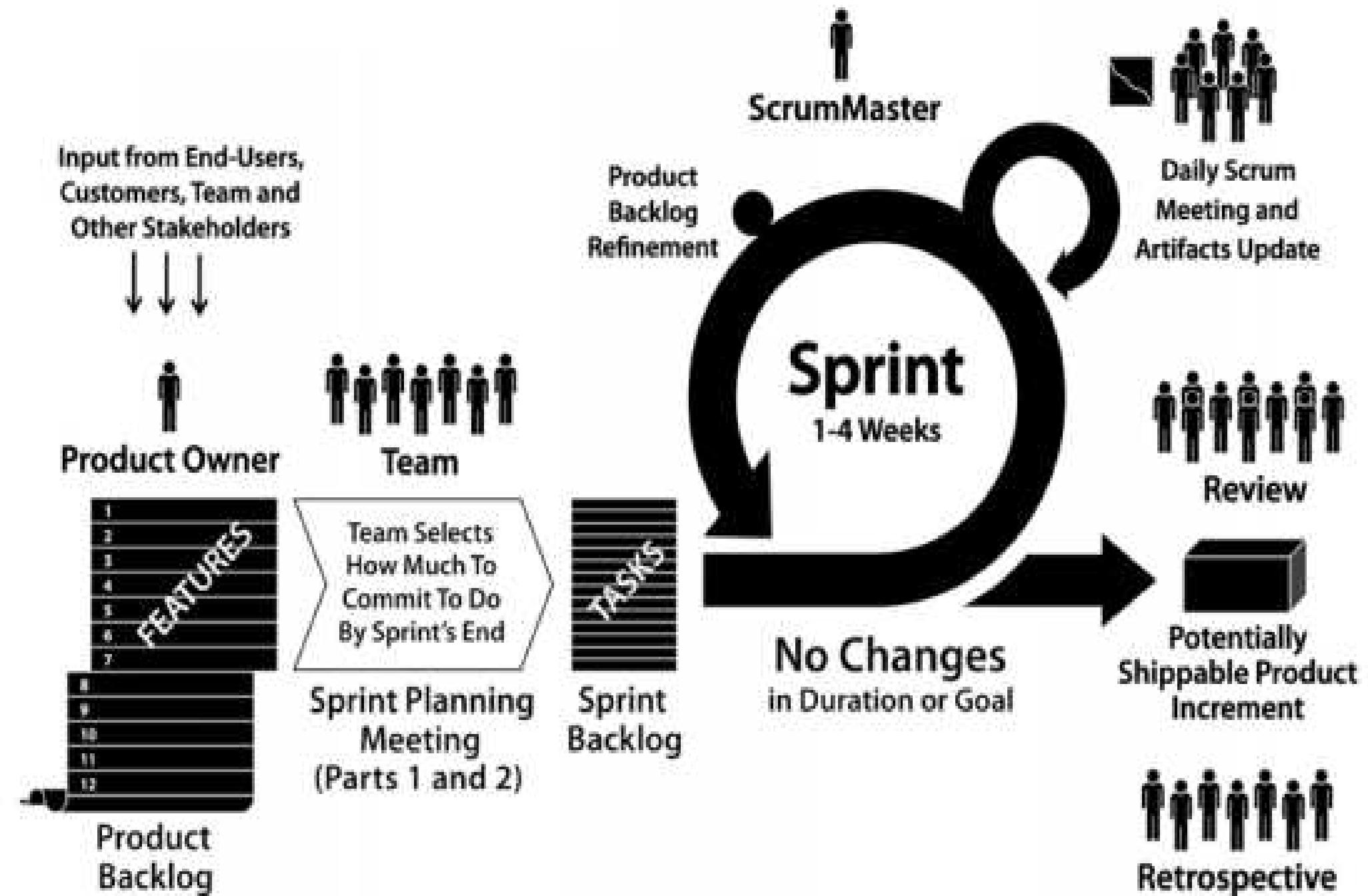
# Scrum

- Tutustumme kurssin aikana suhteellisen tarkasti Scrumiin, joka on tällä hetkellä selvästi suosituin ketterä menetelmä/prosessimalli
- [Schwaber, Sutherland: The Scrum Guide]
  - Scrum is a framework within which people can address complex adaptive problems, while productively and creatively delivering products of the highest possible value
  - Scrum is:
    - Lightweight
    - Simple to understand
    - Extremely difficult to master
  - Scrum is a process framework that has been used to manage complex product development since the early 1990s
  - Scrum is not a process or a technique for building products; rather, it is a framework within which you can employ various processes and techniques.
  - Scrum makes clear the relative efficacy of your product management and development practices so that you can improve

# Scrum lyhyesti

- Iteratiivinen ja inkrementaalinen menetelmä (tai kehittäjiensä mukaan framework eli menetelmäkehys)
- Kehitys tapahtuu 1-4 viikon iteraatioissa, joita Scrumissa kutsutaan **sprinteiksi**
- **Scrum-tiimi** koostuu 3-10:stä kehittäjästä
- **Scrum master** toimii tiimin apuna ohjaten mm. prosessin noudattamisessa sekä toimien rajapintana yrityksen hallintoon
- **Product owner** eli tuotteen omistaja hallinnoi projektin backlogia
  - **backlog** sisältää priorisoidussa järjestyksessä projektissa toteutettavan ohjelmiston ominaisuudet/vaatimukset/toiminnot
- Jokaisen sprintin alussa tiimi valitsee projektin backlogista sprintin aikana toteutettavat vaatimukset
- Sprintin aikana scrum-tiimi toteuttaa itseorganisoidusti sprinttiin valitut vaatimukset lopputuloksena vaatimusten osalta toimiva ohjelmisto

# Scrum: roles, artifacts and events



# Scrum: roles, artifacts and events

- Käydään vielä läpi hieman seikkaperäisemmin scrumin terminologiaa
- Scrum määrittelee 3 erilaista **roolia**:
  - Kehittäjä
  - Scrum master
  - Product owner
- Scrumiin kuuluvat **artefaktit** eli "konkreettiset asiat" ovat
  - Product backlog eli projektin kehitysjono
  - Sprint backlog eli sprintin tehtävälistä
  - Työn alla oleva ohjelmisto
- Scrumissa tekeminen rytmittyy sprintteihin eli 1-4 viikon mittaisiin iteraatioihin. Sprintteihin kuuluu muutamia **standardipalaverejä** (events):
  - Sprintin suunnittelupalaveri
  - Daily scrum -palaverit
  - Sprintin katselointi
  - Retrospektiivi

# Product backlog

- Product backlog on siis priorisoitu lista asiakkaan tuotteelle asettamista vaatimuksista eli toivotuista ominaisuuksista ja toiminnnoista
  - Voi sisältää myös esim. isompia bugikorjauksia
- Hyvänen käytänteenä pidetään sitä, että backlogissa olevat vaatimukset ovat asiakkaan tasolla olevia mielekkäitä toiminnallisuksia
- Backlogin kärjessä olevat vaatimukset valitaan toteutettavaksi seuraavan sprintin aikana
  - Tämän takia kärjessä olevat vaatimukset on yleensä kirjattu tarkemmin kuin backlogin häntäpäään vaatimukset
- Usein on tarkoituksena myös *estimoida* eli arvioida backlogissa olevien vaatimusten toteuttamisen vaatima työmääärä
  - Työmääräarviot tekee kehittäjätäimi
- Scrum ei määrittele missä muodossa backlog ja siinä olevat vaatimukset esitetään
  - Viime vuosina on yleistynyt käytäntö, jossa tehtävät esitetään ns. *User Storyinä*, tutustumme tähän tekniikkaan ensi viikolla

# Product owner

- Scrumin mukaan kuka vaan voi milloin tahansa lisätä backlogiin vaatimuksia
- Backlogia priorisoii ainoastaan **Product owner** eli tuotteen omistaja
- Product owner on yksittäinen henkilö
  - Priorisointiin voi toki olla vaikuttamassa useampikin henkilö, mutta Product owner tekee lopulliset päätökset prioriteettien suhteen
- Product owner on vastuussa backlogista
  - Priorisoi vaatimukset maksimoiden asiakkaan tuotteesta saaman hyödyn
  - Varmistaa että kehittäjät iimi ymmärtää toteutettavaksi valitut vaatimukset

# Kehittäjätiimi

- Kehittäjätiimi koostuu noin 3-10:stä henkilöstä, kaikista käytetään nimikettä *developer*
  - Vaikka kaikilla nimike developer, voivat jotkut tiimin jäsenistä oват erikoistuneet omaan osa-alueeseensa (esim. testaaminen), koko tiimi kuitenkin kantaa aina yhteisen vastuun kehitystyöstä
  - Oletuksena on että tiimin jäsenet työskentelevät 100%:sti tiimissä
  - Koko tiimin tulee oletusarvoisesti työskennellä samassa paikassa, mieluiten yhteisessä tiimille varattussa avokonttorissa
- Tiimi on "cross-functional", eli tiimin jäsenten tulisi sisältää kaikki tarvittava kompetenssi järjestelmän suunnitteluun ja toteuttamiseen
- Pääperiaatteena on että kehitystiimiä **ei johdeta** ulkopuolelta
  - Tiimi päättää mihin tavoitteisiin se kussakin sprintissä sitoutuu, eli miten paljon vaatimuksia backlogista valitaan sprintissä toteutettavaksi
  - päättää (tiettyjen reunaehtojen puitteissa) itse sen miten sprintin tavoite toteutetaan
- Tiimi on siis **itseorganisoituva** (self organizing)

# Scrum master

- Jokaisella Scrum-tiimillä on **Scrum master**, eli henkilö joka vastaa siitä että Scrumia noudatetaan kehitystyössä
- Ei perinteinen projektipääällikkö vaan "servant-leader" rooli
  - Rohkaisee ja auttaa tiimiä itseorganisoitumisessa
  - Opastaa hyvien käytänteiden noudattamisessa
  - Järjestää Scrumiin liittyvät palaverit
  - Pyrkii poistamaan kehitystyön esteitä
    - Esteenä voi olla jokin tiimistä riippumaton asia, jonka poistamiseksi Scrum master joutuu neuvottelemaan esim. yrityksen hallinnon kanssa
    - "Este" voi myös liittyä ryhmän työtapoihin, tällöin Scrum master opastaa ryhmää toimimaan siten, että tuottavuutta haittaava este poistuu
  - Suojaa tiimiä esim. ulkopuolisten yrityksiltä puuttua sprintin aikaiseen toimintaan
  - Auttaa tuotteenomistajaa eli product owneria product backlogin ylläpitämisessä
- Eli Scrum master tekee kaikkensa, jotta tiimillä olisi optimaaliset olosuhteet kehittää tuotetta

# Sprintti

- Scrumissa kehitystyö siis jakautuu 1-4 viikon mittaisiin iteraatioihin eli sprintteihin
  - Sprintin kesto on projektissa tyypillisesti aina sama, nykyään suosituin sprintin pituus lienee 2 viikkoa
  - Sprintti on "time-boxed", eli sprinttiä ei missään olosuhteissa pidennetä
- Jokaisen sprintin alussa tiimi valitsee projektin backlogista sprintin aikana toteutettavat vaatimukset
  - Backlog on priorisoitu ja vaatimukset valitaan aina priorisoidun listan kärjestä
- Tiimi valitsee sprinttiin ainoastaan sen verran toteutettavaa minkä valmistumiseen se uskoo kykenevänsä
- Scrumissa periaatteena on, että jokaisen sprintin lopuksi tuotteesta on oltava olemassa **toimiva versio**
- Sprintin aikana scrum-tiimi toteuttaa itseorganisoidusti sprinttiin valitut ohjelmiston ominaisuudet
- Sprintin aikana tiimille ei esitetä uusia vaatimuksia

# Definition of done

- Scrum kuten kaikki muutkin ketterät menetelmät asettavat suuren painoarvon tuotetun ohjelmiston laadulle
- Jokaisessa sprintissä siis tulee lopputuloksena olla toimiva, valmiiksi tehty osa ohjelmistoa
- Scrumissa on määriteltävä projektitasolla **definition of done** eli se mitä tarkoittaa, että jokin vaatimus on toteutettu valmiiksi
- Valmiiksi tehty määritellään yleensä tarkoittamaan sitä, että vaatimus on
  - analysoitu, suunniteltu, ohjelmoitu, testattu, testaus automatisoitu, dokumentoitu, integroitu muuhun ohjelmistoon ja viety tuotantoypäristöön
- Eli kun sprintin lopussa tavoitteena on olla toimiva ohjelma, tarkoitetaan sillä nimenomaan definition of done:n tasolla toimivia ja valmiiksi tehtyjä vaatimuksia
  - Jos joitain ohjelman osia on tehty huolimattomasti, Scrum master hylkää ne ja siirtää toteutettavaksi seuraavaan sprinttiin
  - Jos sprintin aikana osoittautuu että tiimi ei ehdi toteuttamaan kaikein se sitoutui, **ei ole hyväksytävää tinkiä laadusta**, vaan osa vaatimuksista jätetään seuraavaan sprinttiin

# Sprint planning

- Ennen jokaista sprinttiä järjestetään sprintin suunnittelukokous
  - Aiemmin Scrum määritteli, että kokous on kaksiosainen, nykyään puhutaan ainoastaan kokouksen kahdesta aiheesta (engl. topic)
- Ensimmäisen aihe on selvittää **mitä sprintin aikana tehdään**
  - Product owner esittelee product backlogin kärjessä olevat vaatimukset
  - Tiimin on tarkoitus olla riittävällä tasolla selvillä siitä, mitä vaatimuksilla tarkoitetaan
  - Tiimi arvioi kuinka monta tehtävälistan vaatimuksista se kykenee sprintin aikana toteuttamaan (Definition of doneen määrittelemällä laadulla)
- Sprintin aikana toteutettavien vaatimusten lisäksi asetetaan **sprintin tavoite** (sprint goal)
  - Tavoite on yksittäisiä vaatimuksia geneerisempi ilmaus siitä mitä tulevassa sprintissä on tarkoitus tehdä

# Sprint planning

- Suunnittelukokouksen toisena aiheena on selvittää **miten sprintin tavoitteet saavutetaan**
- Tämä yleensä edellyttää että tiimi suunnittelee toteutettavaksi valitut vaatimukset tarvittavalla tasolla
- Aikaansaannoksesta on usein lista tehtävistä (**task**), jotka sprintin aikana on toteutettava, jotta sprinttiin valitut vaatimukset saadaan toteutettua
- Suunnittelun aikana identifioidut tehtävät kirjataan **sprintin backlogiin** eli sprintin tehtävälistaan
- Sprint planningin maksimikesto on 8 tuntia jos sprinttien pituus on 4 viikkoa ja muuten 4 tuntia
- Palaamme sprintin suunnitteluun tarkemmin ja konkreettisten esimerkkien kanssa ensi viikolla

# Daily scrum – päiväpalaveri

- Jokainen päivä sprintin aikana aloitetaan **daily scrumilla** eli korkeintaan 15 minuutin mittaisella palaverilla
- Aina samaan aikaan, samassa paikassa, kaikkien kehittäjien oltava paikalla
- Jokainen tiimin jäsen vastaa vuorollaan kolmeen kysymykseen
  - Mitä sain aikaan edellisen tapaamisen jälkeen?
  - Mitä aion saada aikaan ennen seuraavaa tapaamista?
  - Mitä esteitä etenemiselläni on?
- Kuka tahansa saa olla seuraamassa daily scrumia, mutta vain tiimin jäsenillä on puheoikeus
- Palaverin on tarkoitus olla lyhyt ja muu keskustelu ei ole sallittua
  - Jos jollakin on ongelmia, Scrum master keskustelee asianomaisen kanssa daily scrumin jälkeen
- Jos muuhun palaverointiin, esim. suunnittelun tai vaatimusten tarkentamiseen on tarvetta, tulee palaverit järjestää daily scrumista erillään
  - Scrum ei ota kantaa muihin palavereihin

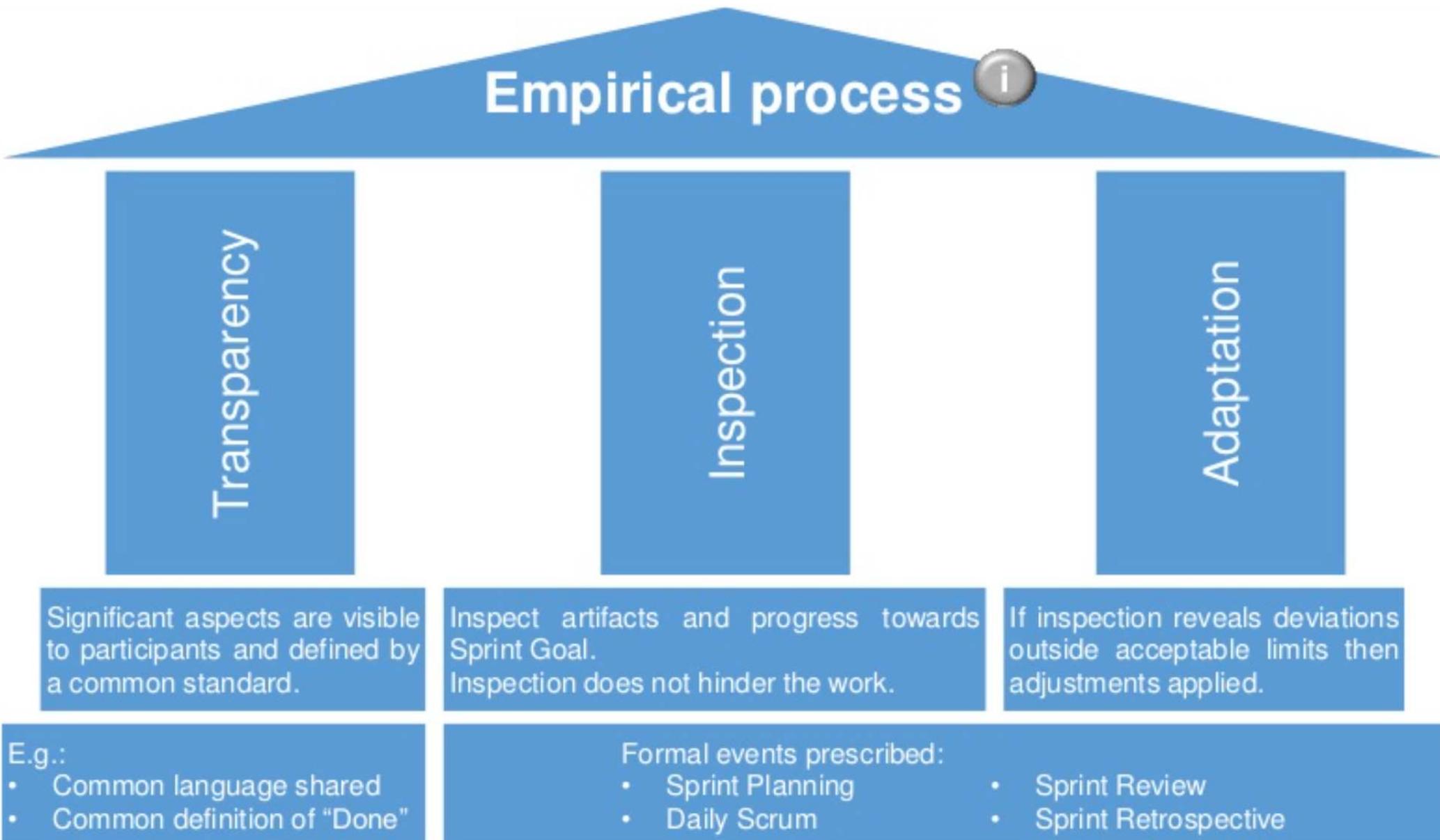
# Sprintin katselointi

- Sprintin pääteeksi järjestetään sprint review eli katselointi
- Katselointiin voi osallistua kuka tahansa
- Informaali tilaisuus, jonka aikana tiimi esittelee sprintin aikaansaannoksia
  - Katselmoinnissa tarkastellaan/demotaan toteutettua, toimivaa ohjelmistoa, powerpoint-kalvojen näytäminen katselmoinnissa on kielletty!
- Scrum master huolehtii, että ainoastaan niitä ominaisuuksia demonstroidaan jotka on toteutettu "kokonaan" eli definition of doneen mukaisesti
- Product owner varmistaa, mitkä vaatimuksista toteutettiin hyväksyttävällä tavalla. Ne vaatimukset joita ei hyväksytä toteutetuksi siirretään takaisin product backlogiin
- Katselmoinnin aikana kuka tahansa saa antaa palautetta tuotteesta ja esim. ehdottaa uusia vaatimuksia lisättäväksi product backlogiin
- Katselointi aiheuttaa usein myös tarpeen product backlogin osittaiseen uudelleenpriorisointiin
- Myös katselmoinnin kesto on rajoitettu (4h tai 2h riippuen sprintin kestosta)

# Retrospektiivi – transparency inspection, adaption

- Sprintin katselmoinnin ja seuraavan sprintin alun välissä pidettävä palaveri, jonka aikana tiimi tarkastelee omaa työskentelyprosessiaan
  - Identifioidaan mikä meni hyvin ja missä asioissa on parantamisen varaa
  - Mietitään ratkaisuja joihinkin ongelmakohtiin, joita pyritään korjaamaan seuraavan sprintin aikana
- Retrospektiivien ja koko scrumin tärkeimmät taustaperiaatteet ovatkin **transparency, inspection ja adaption**:
  - Lyhyt kehityssykli mahdollistaa vaatimusten uudelleenpriorisoinnin ja muuttamisen ymmärryksen kasvaessa tai bisnesympäristön muuttuessa
  - Retrospektiivi kannustaa tiimiä jatkuvasti parantamaan työprosessiaan
  - Daily scrumit tuovat esiin projektin tilanteen päivittäisellä tasolla kaikille tiimin jäsenille
  - Jokaisen sprintin yhteydessä järjestetään uusi sprintin suunnittelu, joka mahdollistaa kehitystyön aikana opitun huomioimisen priorisoinnissa ja uusien ominaisuuksien suunnittelussa

# Taustalla olevat periaatteet transparency -inspection – adaption



# No silver bullet...

- Scrum on osoittautunut monin paikoin paremmaksi tavaksi ohjelmistojen tuottamiseen kuin vesiputousmalli tai muut suunnitelmavetoiset mallit
- Scrum ei kuitenkaan ole mikään "silver bullet" ja Scrumin käytön yleistyessä myös epäonnistuneiden Scrum-projektien määrä kasvaa
- Yksi ongelmista on ns. **scrumbut**
  - "we are doing scrum but ...", ks,  
<https://www.scrum.org/resources/what-scrumbut>
- Toisin kuin esim. eXtreme Programming eli XP, Scrum ei määrittelee mitään teknisiä käytänteitä vaan luottaa itseorganisoidun tiimin kykyyn tuottaa laadukasta jälkeä. Läheskään aina tämä ei toteudu ja lupaavan alun jälkeen tiimi saattaa joutua vaikeuksiin, ks:
  - ks. <http://www.martinfowler.com/bliki/FlaccidScrum.html>
  - Hajautettu ohjelmistotuotanto, alihankkijoiden käyttö ja massiivista kokoluokkaa olevat projektit aiheuttavat edelleen haasteita Scrumille ja muillekin ketterille menetelmille vaikkakin asiaan on viime vuosina kiinnitetty huomiota
  - Robert Martinin Scrum-kriitikkiä (Martin on yksi agile manifestin allekirjoittajista)
    - <http://tech.groups.yahoo.com/group/scrumdevelopment/message/44851>
  - Päätetään alustava Scrumiin tutustumisemme menetelmän kehittäjien sanoihin "*Scrum is easy to understand but extremely difficult to master*"

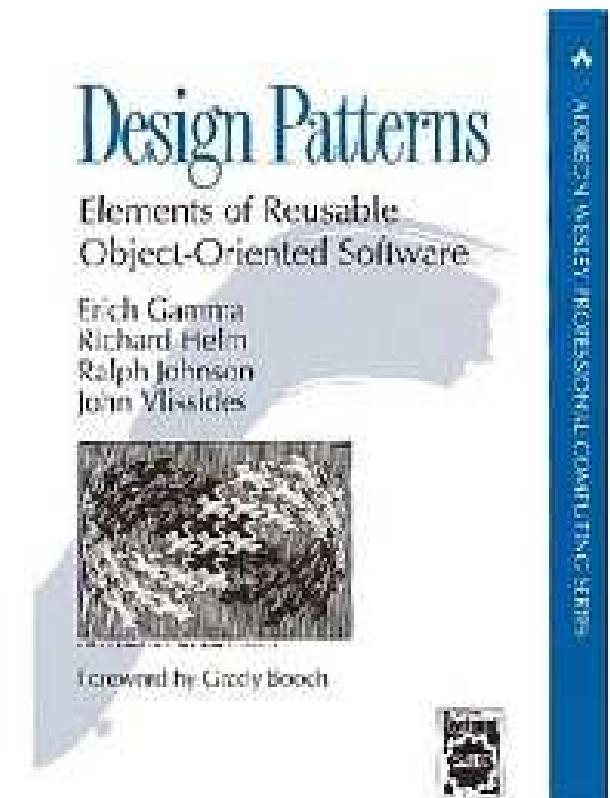
# Design patterns – suunnittelumallit

- Ohjelmistotuotannon yksi paha ongelma on se, että pyörä keksitää jatkuvasti uudelleen
- Design patternit eli **suunnittelumallit** tarjoavat pieni lääkkeen tähän ongelmaan
- Mistä on kysymys? Annetaan "Gang of four:in" kertoa:

A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems.

It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem.

The solution is customized and implemented to solve the problem in a particular context



# Design patterns – suunnittelumallit

- Suunnittelumallit siis dokumentoivat hyväksi tunnettuja suunnitteluratkaisuja
- Gang of Fourin vuonna 1994 julkaisema kirja lanseerasi suunnittelumallien käsitteen tietotekniikkaan
  - Siitä lähtien aiheesta on ilmestynyt lukematon määrä julkaisuja
- Suunnittelumallit jakautuvat useisiin ”aliluokkiin”
  - Olioiden luomista helpottavat mallit
  - Ohjelman oliorakennetta ohjaavat mallit
  - Ohjelman suoritusta ja laskentaa ohjaavat mallit
- On olemassa myös muita ”patterneja”, mm:
  - Architectural patterns: arkkitehtuurimallit
  - Project management patterns: esim. Scrumin voidaan ajatella olevan tälläinen
  - Myös muille kuin olioparadigmaa noudattaville kielille on olemassa omat patterninsa
- Tutustumme kurssin aikana muutamiin suunnittelumalleihin

# Design pattern of the day – dependency injection

- Kurssin ensimmäinen suunnittelumalli **dependency injection**, eli riippuvuuksien injektiointi ei löydy juuri miltään perinteiseltä suunnittelumallilistalta
- Kyseessä on perimmiltään hyvin yksinkertainen menetelmä, jonka avulla *luokkien riippuvuuksia toisista luokista pyritään minimoimaan*
  - Yksi DI:n suurista motivoivista tekijöistä on yksikkötestauksen helpottaminen
- Lyhyt ja simpeli selitys asiasta:  
<http://jamesshore.com/Blog/Dependency-Injection-Demystified.html>
- Luennolla esitetyt koodit löytyvät viikon 2 laskareiden tehtävänannosta
- Riippuvuuksien injektiointi yhdessä ns. *Inversion of Control* -periaatteen kanssa on noussut viime vuosina suosituksi rakenneperiaatteeksi sovelluskehysissä, esim. Spring:issä
  - ks. <http://martinfowler.com/articles/injection.html>

Ohjelmistotuotanto

Luento 3

20.3.

# Vaatimusmäärittely

- Ehkä keskeisin ongelma ohjelmistotuotantoprosessissa on määritellä asiakkaan **vaatimukset** (requirements) rakennettavalle ohjelmistolle
- Vaatimukset jakaantuvat kahteen luokkaan
  - **Toiminnalliset vaatimukset** (functional requirements)
    - Ohjelman toiminnot: mitä ohjelma tekee ja mitä sillä voidaan tehdä
  - **Ei-toiminnalliset vaatimukset** (nonfunctional requirements)
    - Koko ohjelmistoa koskevat "laatuvaatimukset" ja
    - ohjelmiston toimintaympäristön asettamat rajoitteet
- Vaatimusten selvittämistä, dokumentoimista ja hallinnointia kutsutaan **vaatimusmäärittelyksi**, engl. **requirements engineering**
- Käytettävästä prosessimallista riippumatta vaatimusmäärittelyn tulee ainakin alkaa ennen ohjelmiston suunnittelua ja toteuttamista
  - Lineaarississa prosessimalleissa vaatimusmäärittely tehdään kokonaisuudessaan ennen ohjelmiston suunnittelua ja toteutusta
  - Iteratiivisessa ohjelmistokehityksessä vaatimusmäärittelyä tapahtuu paloittain ohjelmiston rakentamisen edetessä

# Vaatimusmäärittelyn vaiheet

- Vaatimusmäärittelyn luonne vaihtelee paljon riippuen kehitettävästä ohjelmistosta, kehittäjäorganisaatiosta ja ohjelmistokehitykseen käytettävästä prosessimallista
- Joka tapauksessa asiakkaan tai asiakkaan edustajan on oltava prosessissa aktiivisesti mukana
- Vaatimusmäärittely jaotellaan yleensä muutamaan työvaiheeseen
  - Vaatimusten kartoitus (engl. elicitation)
  - Vaatimusanalyysi
  - Vaatimusten validointi
  - Vaatimusten dokumentointi
  - Vaatimusten hallinnointi
- Useimmiten työvaiheet limittyyt ja vaatimusmäärittely etenee spiraalimaisesti tarkentuen
  - Ensin kartoitetaan, analysoidaan ja dokumentoidaan osa vaatimuksista
  - Prosessia jatketaan kunnes haluttu määrä vaatimuksia on saatu dokumentoitua tarvittavalla tarkkuudella

# Vaatimusten kartoituksen menetelmää

- Selvitetään järjestelmän **sidosryhmät** (stakeholders) eli tahot, jotka ovat suoraan tai epäsuorasti tekemisissä järjestelmän kanssa
- Käytetään ”kaikki mahdolliset keinot” vaatimusten esiin kaivamiseen, esim.:
  - Haastatellaan sidosryhmien edustajia
  - Pidetään brainstormaussessioita asiakkaan ja kehittäjien kesken
- Alustavien keskustelujen jälkeen kehittäjät iimi voi strukturoida vaatimusten kartoitusta
  - Mietitään järjestelmän *kuviteltuja käyttäjiä* ja keksitään käyttäjille tyypillisiä käytöskenaarioita
  - Tehdään paperiprototyypejä ja käyttöliittymäluonnonoksia
- Skenaarioita ja prototyypejä läpikäymällä asiakas voi tarkentaa näkemystäään vaatimuksista
- Jos kehitettävän järjestelmän on tarkoitus korvata olemassa oleva järjestelmä, voidaan vaatimuksia selvittää myös havainnoimalla loppukäyttäjän työskentelyä (ethnografia)

# Vaatimusten analysointi, validointi ja dokumentointi

- Vaatimusten keräämisen lisäksi vaatimuksia täytyy *analysoida*:
  - Onko vaatimuksissa keskinäisiä ristiriitoja
  - Ovatko vaatimukset riittävän kattavat
  - Ovatko vaatimukset sellaisia että niiden toteutuminen on mahdollista, taloudellisesti järkevää ja testattavissa
- Vaatimukset on myös pakko *dokumentoida* muodossa tai toisessa
  - Ohjelmistokehittäjiä varten: mitä tehdään
  - Testaajia varten: toimiiko järjestelmä kuten vaatimukset määrittelevät
  - Usein vaatimusdokumentti toimii oleellisena osana asiakkaan ja ohjelmistotuottajatiimin välisessä sopimuksessa
- Ja *validoida*:
  - Onko asiakas vielä sitä mieltä että kirjatut vaatimukset edustavat asiakkaan mielipidettä, eli kuvaavat sellaisen järjestelmät mitä asiakas tarvitsee

# Vaatimusten luokittelu – toiminnalliset vaatimukset

- Vaatimukset jakaantuvat toiminnallisiin ja ei-toiminnallisiin vaatimuksiin
- **Toiminnalliset vaatimukset** (functional requirements) kuvaavat mitä toimintoja järjestelmällä on
  - Esim.
    - *Asiakas voi lisätä tuotteen ostoskoriin*
    - *Onnistuneen luottokorttimaksun yhteydessä asiakkaalle vahvistetaan ostotapahtuman onnistuminen sähköpostitse*
- Toiminnallisten vaatimusten dokumentointi voi tapahtua esim.
  - "feature-listoina"
  - UML-käyttötapaiksina (ks. OTM)
  - Ketterissä menetelmissä yleensä **User storyinä**, joihin tutustumme kohta tarkemmin

# Ei-toiminnalliset vaatimukset

- Ei-toiminnalliset vaatimukset (nonfunctional requirements) jakautuvat kahteen luokkaan: **laatuvaatimuksiin ja toimintaympäristön rajoitteisiin**
- Laatuvaatimukset (quality attributes), ovat koko järjestelmän toiminnallisuutta rajoittavia/ohjaavia tekijöitä, esim.
  - Käytettävyys
  - Testattavuus
  - Laajennettavuus
  - Suorituskyky
  - Skaalautavuus
  - Tietoturva
  - [http://en.wikipedia.org/wiki/List\\_of\\_system\\_quality\\_attributes](http://en.wikipedia.org/wiki/List_of_system_quality_attributes)
- Toimintaympäristön rajoitteita (constraints) ovat esim:
  - Toteutusteknologia (esim. tulee toteuttaa Ruby on Railsilla)
  - Integroituminen muihin järjestelmiin (esim. kirjautuminen google-tunnuksilla)
  - Mukautuminen standardeihin
- Ei-toiminnalliset vaatimukset vaikuttavat yleensä ohjelman arkkitehtuurin suunnitteluun

# Vaatimusmäärittely 1900-luvulla

- Vesiputousmallin hengen mukaista oli, että vaatimusmäärittelyä pidettiin erillisenä ohjelmistoprosessin vaiheena, joka on tehtävä kokonaisuudessaan ennen suunnittelun aloittamista
  - Ideana oli että suunnittelun ei pidä vaikuttaa vaatimuksiin ja vastaavasti vaatimukset eivät saa rajoittaa tarpeettomasti suunnittelua
- Asiantuntijat korostivat, että vaatimusten dokumentaation on oltava kattava ja ristiriidaton
  - Pidettiin siis ehdottoman tärkeänä että heti alussa kerätään ja dokumentoitiin *kaikki* asiakkaan vaatimukset
  - mielessään luonnollisen kielen sijaan formaalilla kielellä (matemaattisesti) tehty jotta esim. ristiriidattomuuden osoittaminen olisi mahdollista
- Tiedetään että jos määrittelyvaiheessa tehdään virhe, joka huomataan vasta testauksessa, on muutoksen tekeminen kallista
  - Tästä loogisena johtopäätöksenä oli tehdä vaatimusmäärittelystä erittäin järeä ja huolella tehty työvaihe
- Vaatimusdokumenttipohjia standardoitiin
  - *IEEE Recommended Practice for Software Requirements Specifications ks.*  
<http://ieeexplore.ieee.org>

# Vaatimusmäärittely 1900-luvulla – ei toimi

- Ideaali jonka mukaan vaatimusmäärittely voidaan irrottaa kokonaan erilliseksi, huolellisesti tehtäväksi vaiheeksi on osoittautunut utopiaksi
- Vaatimusten muuttumien on väistämätöntä
  - Ohjelmistoja käyttävien organisaatioiden toimintaympäristö muuttuu nopeasti, mikä on relevanttia tänään, ei ole välttämättä sitä enää 3 kuukauden päästä
  - Asiakkaiden on mahdotonta ilmaista tyhjentävästi tarpeitaan etukäteen
  - Ja vaikka asiakas osaisikin määritellä kaiken etukäteen, tulee mielipide muuttumaan kun asiakas näkee lopputuloksen
  - Huolimatta huolellisesta vaatimusmäärittelystä, ohjelmistokehittäjät eivät osaa tulkita kirjattuja vaatimuksia samoin kuin vaatimukset kertonut asiakas
- Vaatimusmäärittelyä ei ole mahdollista/järkevää irrottaa suunnittelusta
  - Suunnittelu auttaa ymmärtämään ongelma-aluetta syvällisemmin ja generoi muutoksia vaatimuksiin
  - Ohjelmia tehdään maksimoiden valmiiden ja muualta, esim. open sourcena saatavien komponenttien käyttö, tämä on syytä ottaa huomioon vaatimusmäärittelyssä
  - Jos suunnittelu ja toteutustason asiat otetaan huomioon, on vaatimusten priorisointi helpompaa: helpompi arvioida vaatimusten toteuttamisen hintaa

# Vaatimusmäärittely 2000-luvulla

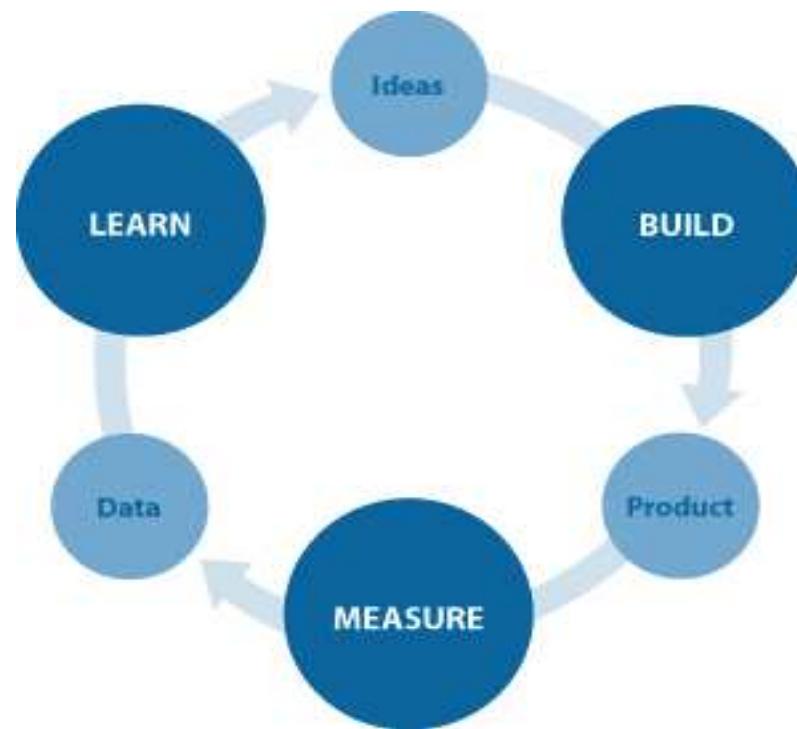
- Nykyään vallitsee laaja konsensus siitä, että useimmissa tilanteissa vaatimusmäärittelyä ei ole järkevä tehdä kokonaan suunnittelusta ja toteutuksesta irrallaan
- Syitä tälle
  - **Time to market:** ohjelmistotuotteet halutaan markkinoille nopeasti ja perinpohjiselle, kuukausia kestävällä vaatimusmäärittelylle ei ole aikaa
  - Tämän takia kaikkia vaatimuksia ei edes teoriassa ehditä kartoittamaan ja siitä taas seuraa **muuttuvat vaatimukset**
  - **Uusikäyttö, ohjelmistojen koostaminen palveluista:** ohjelmistoja tehdään enenevissä määrin räätälöimällä valmiista komponenteista ja verkossa/pilvessä olevista palveluista, vaatimukset riippuvat näin enenevissä määrin muustakin kuin asiakkaan tahdosta
- Ilman suunnittelun ja toteutuksen huomioimista vaikea tietää vaatimusten toteuttamisen hintaa
  - Riskinä että asiakas haluaa vaatimuksen muodossa, joka moninkertaistaa toteutuksen hinnan verrattuna periaatteessa asiakkaan kannalta yhtä hyvään, hieman eri tavalla muotoiltuun vaatimukseen

# Ohjelmiston suunnitteluun ja toteutukseen integroitu vaatimusmäärittely

- 2000-luvun iteratiivisen ja ketterän ohjelmistotuotannon tapa on integroida kaikki ohjelmistotuotannon vaiheet yhteen
- Ohjelmistoprojektiin alussa määritellään vaatimuksia tarkemmassa tasolla ainakin yhden iteraation tarpeiden verran
- Ohjelmistokehittäjät arvioivat vaatimusten toteuttamisen hintaa
- Asiakas priorisoi vaatimukset siten, että iteraatioon valitaan toteutettavaksi vaatimukset, jotka tuovat mahdollisimman paljon liiketoiminnallista arvoa
- Jokaisen iteraation aikana tehdään määrittelyä, suunnittelua, ohjelmointia ja testausta
- Jokainen iteraatio tuottaa valmiin osan järjestelmää
  - Edellisen iteraation tuotos toimii syöteenä seuraavan iteraation vaatimusten määrittelyyn
- Ohjelmisto on mahdollista saada tuotantoon jo ennen kaikkien vaatimusten valmistumista
- *Kattavana teemana tuottaa asiakkaalle maksimaalisesti arvoa*

# Vaatimusmäärittely 2010-luvulla: Lean startup

- Eric Riesin vuonna 2011 julkaisema kirja Lean startup kuvailee formalisoili systemaattisen tavan kartoittaa vaatimuksia erityisen epävarmoissa konteksteissa, kuten startup-yrityksissä
- Malli perustuu kolmiosaisen *build-measure-learn*-syklin toistamiseen



# Vaatimusmäärittely 2010-luvulla: Lean startup

- Esim. internetpalveluja tai mobiilisovelluksia rakennettaessa asiakkaan tarpeista, eli järjestelmän vaatimuksista ei ole minkäänlaista varmuutta, voidaan vain olettaa mitä ihmiset tulisivat käyttämään
  - Alkuvaiheessahan järjestelmällä ei edes ole vielä asiakkaita/käyttäjiä, joilta voitaisiin kysyä mitä he haluavat
- Otetaan lähtökohdaksi jokin idea siitä, mitä asiakkaat haluavat ja tehdään **hypoteesi** miten asiakkaat käyttäytyisivät, jos kyseinen järjestelmä/toiminnallisuus/ominaisuusjoukko olisi toteutettu
- Rakennetaan nopeasti ns. **Minimal Viable Product** (MVP) joka toteuttaa ominaisuuden
  - A **minimum viable product** has just those core features that allow the product to be deployed, and no more. The product is typically deployed to a subset of possible customers, such as early adopters that are thought to be more forgiving, more likely to give feedback, and able to grasp a product vision from an early prototype or marketing information. [Wikipedia]

# Vaatimusmäärittely 2010-luvulla: Lean startup

- MVP laitetaan tuotantoon ja **mitataan** miten asiakkaat käyttäytyvät uuden ominaisuusjoukon suhteeseen
- Jos MVP koskee jotain järjestelmään toteutettua uutta ominaisuutta, käytetään usein **A/B-testausta**: uusi ominaisuus julkaistaan vain osalle käyttäjistä, loput jatkavat vanhan ominaisuuden käyttöä
- Käyttäjien oikeasta järjestelmästä mitattua käyttäytymistä verrataan sitten alussa asetettuun hypoteesiin ja näin pystytään **oppimaan** olivatko toteutetut vaatimukset sellaisia, joita asiakkaat halusivat
- Jos toteutettu idea ei osoittautunut hyväksi, voidaan palata järjestelmän edelliseen versioon ja jatkaa *build-measure-learn*-sykliä tekemällä hypoteesi jostain muusta ideasta
- Lean startup -”menetelmällä” on siis tarkoitus oppia systemaattisesti ja mahdollisimman nopeasti mitä asiakkaat haluavat
  - Jos idea ei osoittaudu menestyksekäksi, on parempi että **suuntaa vaihdetaan** (engl. pivot) nopeasti

# Vaatimusmäärittely ja projektisuunnittelu ketterässä prosessimallissa

# Taustaa

- Seuraavassa esitellään yleinen tapa vaatimustenhallintaan ja projektisuunnitteluun ketterässä ohjelmistotuotantoprojektissa
- Tapa pohjautuu Scrumin ja eXtreme Programingin eli XP:n eräiden käytänteiden soveltamiseen
- Lähteenä on käytetty mm. seuraavia:
  - Kniberg Scrum and XP from the trenches, sivut 9-55
  - Shore: Art of agile development, osa luvusta 8
  - Rasmussen: The Agile Samurai, luvut 6-8
- Kaikissa edellisissä käydään läpi suunnilleen samat asiat, terminologia ja painotukset hieman vaihtelevat (Kniberg käyttää Scrumin ja muut XP:n terminologiaa). Tärkeimmät erot terminologiassa
  - Scrumin sprinttiä kutsutaan XP:ssä iteraatioksi
  - XP:n *on-site customer* on suunnilleen sama kuin Scrumin *Product owner*
  - XP:ssä ei ole selvää vastinetta Scrum Masterille, koko tiimi jakaa vastuun prosessin noudattamisesta
- Erittäin kattavan kuvan asioihin antavat Mike Cohnin loistavat kirjat *Agile Estimation and Planning* ja *User stories applied*

# User story

- Ketterän vaatimusmäärittelyn tärkein työväline on **User story**
  - Käsitteelle ei ole vakiintunutta käänöstä, joten käytämme jatkossa englanninkielistä termiä
- Alan suurimman auktoriteetin Mike Cohnin mukaan:

A user story describes functionality that will be valuable to either user or purchaser of software. User stories are composed of three aspects:

- 1) A written **description** of the story, used for planning and reminder
- 2) **Conversations** about the story to serve to flesh the details of the story
- 3) **Tests** that convey and document details and that will be used to determine that the story is complete

- Mitä ylläoleva kuvaus tarkoittaa? Jatketaan user storyihin tutustumista käymällä samalla läpi esimerkkijärjestelmää Kumpula beershop:
  - <https://github.com/mluukkai/BeerShop>
  - <http://kumpulabeershop.herokuapp.com/>

# User story

- User Storyt kuvaavat **loppukäyttäjän kannalta arvoa tuottavia toiminnallisuksia**
- US:n ”määritelmän” alakohdat 1 ja 2 antavat ilmi sen, että User story on karkean tason tekstuaalinen kuvaus **ja** lupaus/muistutus siitä, että toiminnallisuuden vaatimukset on selvitettävä asiakkaan kanssa
- Seuraavat voisivat olla biershopin User storyjen tekstuaalisia kuvaauksia:
  - Asiakas voi lisätä oluen ostoskoriin
  - Asiakas voi poistaa ostoskorissa olevan oluen
  - Asiakas voi maksaa luottokortilla ostoskorissa olevat oluet
- User story ei siis ole perinteinen vaatimusmääritelmä, joka ilmaisee tyhjentävästi miten joku toiminnallisuus tulee toteuttaa
  - User story on ”placeholder” vaatimukselle, muistilappu ja lupaus, että toiminnallisuuden vaatimukset tulee selventää tarvittavalla tasolla ennen kuin se toteutetaan
- Usein on tapana kirjoittaa User storyn kuvaus pienelle noin 10-15 cm pahvikortille tai postit-lapulle

# User story

- Kun User story päätetään toteuttaa, on pakko selvittää tyhjentävästi, mitkä ovat Storyn kirjaaman toiminnon vaatimukset
- User storyn henkeen siis kuuluu, että Story on lupaus kommunikoinnista asiakkaan kanssa vaatimuksen selvittämiseksi
  - *conversations about the story to serve to flesh the details of the story*
- "määritelmän" kolmas alikohta sanoo että Storyyn kuuluu "*Tests that convey and document details and that will be used to determine that the story is complete*"
- User storyyn liittyviä testejä kutsutaan yleensä **Storyn hyväksymästeiksi** (acceptance test) tai hyväksymäkriteereiksi (acceptance criteria)
- Hyväksymätesti tarkoittaa yleensä joukkoa konkreettisia testiskenaarioita joiden on toimittava, jotta User storyn kuvaaman toiminnallisuuden katsotaan olevan valmis
- Hyväksymätestien luonne vaihtelee projekteittain
  - Ne voitat olla Storyn kuvausen sisältävän kortin käänöpuolelle kirjoitettavia tekstuaalisia skenaarioita (varsinkin jos projektissa on käytettävissä on-site customer, joka voi suorittaa hyväksymätestauksen)
  - Tai parhaassa tapauksessa automaattisesti suoritettavia testejä

# Esimerkki

- Alla esimerkki pahvikortille kirjoitetusta User storystä
- Kortin etupuolella kuvaus, prioriteetti ja estimaatti
  - Estimaatilla tarkoitetaan kortin toiminnallisuuden toteuttamisen työmäääräärväriota. Palaamme estimointiin pian tarkemmin
- Kortin takapuolella suhteellisen informaalilla kielellä kirjoitettu hyväksymistesti

Front of Card

IB  
As a student I want to purchase  
a parking pass so that I can  
drive to school

Priority: ~~High~~ Should  
Estimate: 4

Back of Card

Conditions:

The student must pay the correct amount.  
One pass for one month is issued at a time.  
The student will not receive a pass if the payment  
isn't sufficient.

The person buying the pass must be a currently  
enrolled student.

The student may only buy one pass per month.

# Minkälainen on hyvä User Story

- Kuten jo mainittu, tulee User storyn kuvata asiakkaalle arvoa tuottavia toimintoja
  - Käytettävä asiakkaan kieltä, ei teknistä jargonia
- Hyväksi käytäntönä pidetään että User story kuvaaa järjestelmän kaikkia osia koskevaa (esim. käyttöliittymä, bisneslogiikka, tietokanta) eli "end to end"-toiminnallisuutta
  - Esim. "lisää jokaisesta asiakkaasta rivi tietokantatauluun customers" ei olisi suositeltava muotoilu User storylle
- Edellisen sivun esimerkki on formuloitu viimeaikaisen muodon mukaisessa muodossa
  - **As a <type of user>, I want <functionality> so that <business value>**
  - *As a student I want to purchase a parking pass so that I can drive to school*
- Näin muotoilemalla on ajateltu että User story kiinnittää huomion siihen kenelle kuvattava järjestelmän toiminto tuo arvoa
- Muoto ei oikein taivu suomenkielisiin kuvauksiin, joten sitä ei täällä kurssilla käytetä

# Minkälainen on hyvä User Story

- Bill Wake luettelee artikkelissa *INVEST in good User Stories* kuusi User storyille toivottavaa ominaisuutta:
  - **I**ndependent
  - **N**egotiable
  - **V**aluable to user or customer
  - **E**stimable
  - **S**mall
  - **T**estable
- **I**ndependent User storyjen pitäisi olla toisistaan mahdollisimman riippumattomia
  - Riippumattomuus mahdollistaa eri käyttötapausten toteuttamisen mahdollisimman riippumattomasti toisistaan
  - Tämä taas antaa asiakkaalle enemmän vapausasteita storyjen priorisointiin
- Esim. biershopin Storyjen *Lisää olut ostoskoriin* ja *Poista olut ostoskorista* välillä on riippuvuus, jota on vaikea välttää

# Minkälainen on hyvä User Story

- **Negotiable** hyvä User story ei ole tyhjentävästi kirjoitettu vaatimusmäärittely vaan lupaus siitä että asiakas ja toteutustiimi sopivat User storyn toiminnallisuuden sisällön ennen toteutusvaihetta
- **Estimatable** User storyn toteuttamisen vaatima työmäärä pitää olla arvioitavissa kohtuullisella tasolla
- **Small** Työmäärän arvointi onnistuu paremmin jos User storyt ovat riittävän pieniä. User storyä pidetään yleensä liian isona, jos se ei ole toteutettavissa noin viikon työpanoksella
- Liian suuret User storyt kannattaa jakaa osiin
  - Esim käyttötapaus *Olutkaupan ylläpitäjä voi kirjautua sivulle, lisätä ja päivittää oluiden tietoja sekä tarkastella asiakkaille tehtyjen toimitusten lista* kannattaa jakaa useaan osaan:
    - *Ylläpitäjä voi kirjautua sivulle*
    - *Ylläpitäjä voi lisätä ja päivittää oluiden tietoja*
    - *Ylläpitäjä voi tarkastella asiakkaille tehtyjen toimitusten listaa*
    - *Sivulle kirjautunut ylläpitäjä voi lisätä ja päivittää oluiden tietoja*
    - *Sivulle kirjautunut ylläpitäjä voi tarkastella asiakkaille tehtyjä toimituksia*

# Minkälainen on hyvä User Story

- **Testability** Kuudes toivottu ominaisuus on testattavuus, eli User storyjen pitää olla sellaisia, että niille on mahdollista tehdä testit tai laatia kriteerit, joiden avulla voi yksikäsitteisesti todeta onko Story toteutettu hyväksyttävästi
- Ei-toiminnalliset vaatimukset (esim. suorituskyky, käytettävyys) aiheuttavat usein haasteita testattavuudelle
  - Esim. käyttötapaus *Olutkaupan tulee toimia tarpeeksi nopeasti kovassakin kuormituksessa* voidaan muotoilla testattavaksi esim. seuraavasti:
  - *käyttäjän vasteaika saa olla korkeinaan 0.5 sekuntia 99% tapauksissa jos yhtäaikaisia käyttäjiä sivulla on maksimissaan 1000*
- Viime viikolla Scrumin yhteydessä puhuimme **product backlogista**, joka siis on priorisoitu lista asiakkaan tuotteelle asettamista vaatimuksista eli toivotuista ominaisuuksista ja toiminnoista
- Nykyään käytäntönä on, että product backlog koostuu nimenomaan User storyistä

# Alustava product backlog

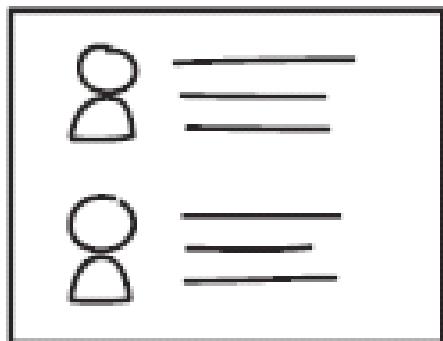
- Projektin aluksi kannattaa heti ruveta etsimään ja määrittelemään User storyja ja muodostaa näistä alustava Product Backlog
- Käytettäväissä ovat kaikki yleiset vaatimusten kartoitustekniikat
  - Haastattelut
  - Brainstormaus, story gathering workshopit
- Alustavan User storyjen keräämisvaiheen ei ole tarkoituksenmukaista kestää kovin kauaa, maksimissaan muutaman päivän
- User storyjen luonne (muistilappu ja lupaus, että vaatimus tarkennetaan ennen toteutusta) tekee niistä hyvän työkalun projektin aloitukseen
  - Turhiin detaljeihin ei puututa
  - Ei tavoitellakaan täydellistä ja kattavaa lista vaatimuksista, asioita tarkennetaan myöhemmin
- Kun alustava lista User storyistä on kerätty, ne priorisoidaan ja niiden vaatima työmäärä arvioidaan karkealla tasolla
  - Näin muodostuu alustava Product Backlog, eli priorisoitu lista vaatimuksista

# Story gathering workshop

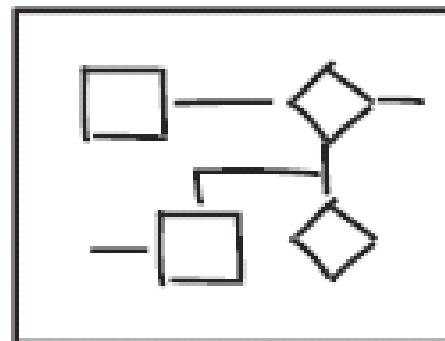
- Ennenkuin menemme tarkemmin User storyjen priorisointiin, esitellään nopeasti Johan Rasmussenin kirjassa Agile Samurai esittämä tapa Storyjen keräämiseen
- Step 1: **get a big room**
  - Huoneeseen kerääntyvät kaikki asianosaiset, asiakkaat ja ohjelmistotuotantotyöntekijät



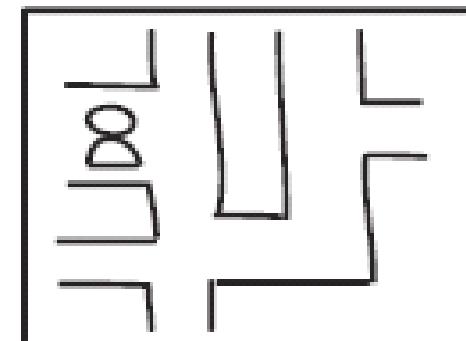
- Step 2: draw a lot of pictures



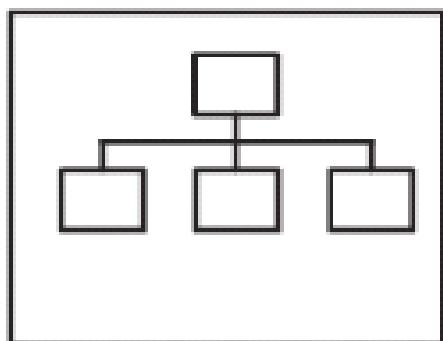
Personas



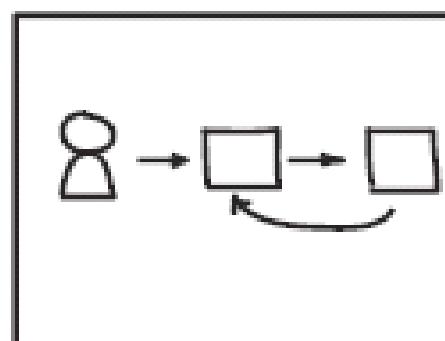
Flowcharts



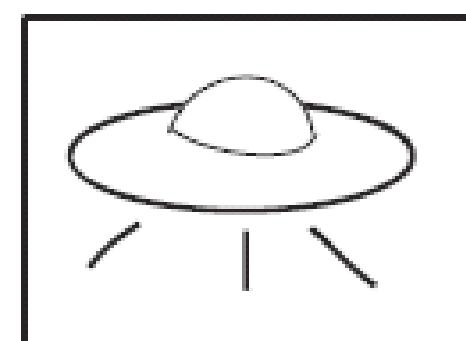
Scenarios



System maps



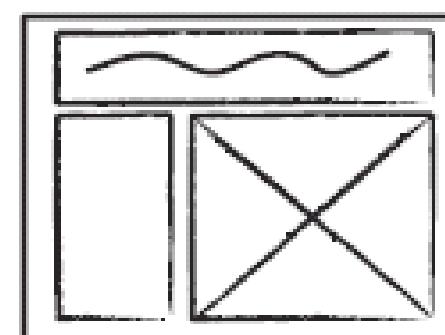
Process flows



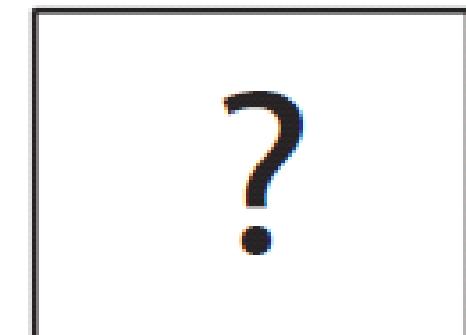
Concept designs



Storyboards

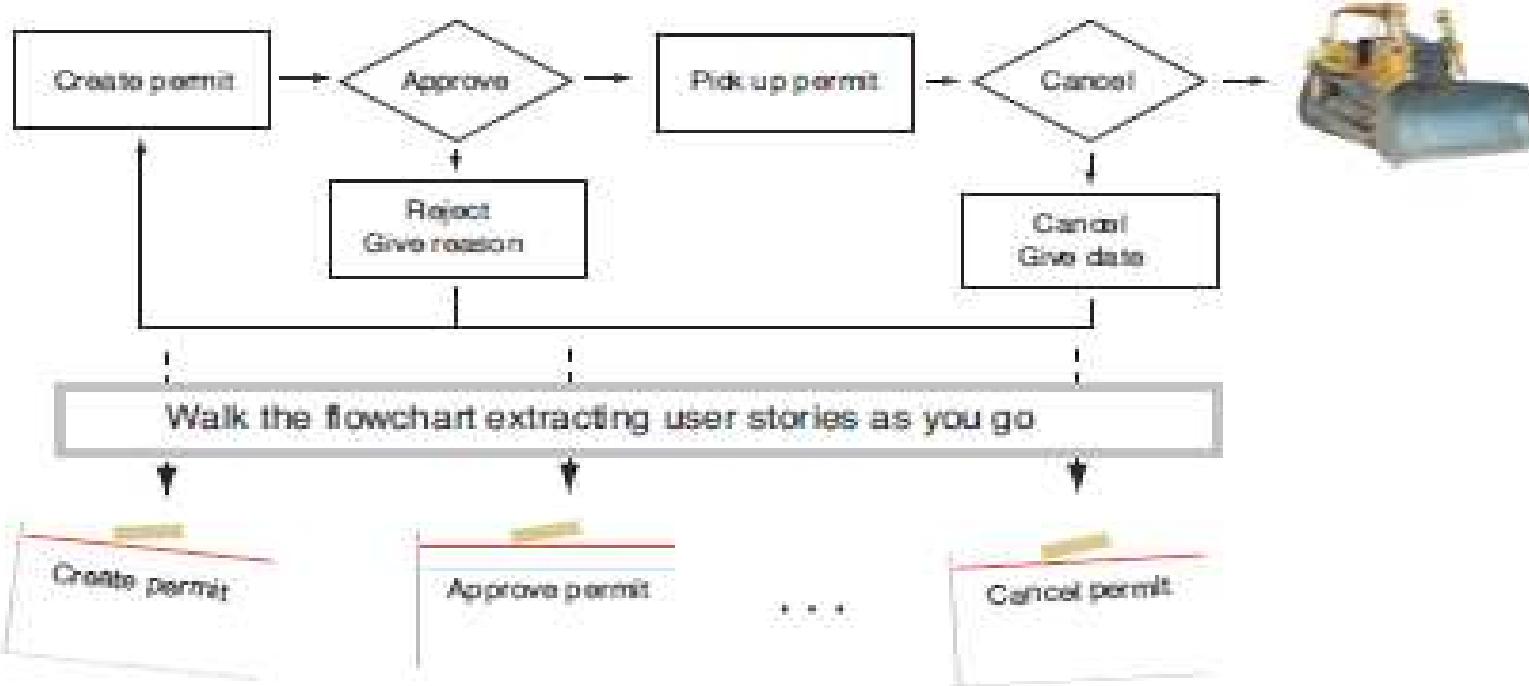


Paper prototypes



Your own

- Step 3: Write lots of stories



# Story gathering workshop

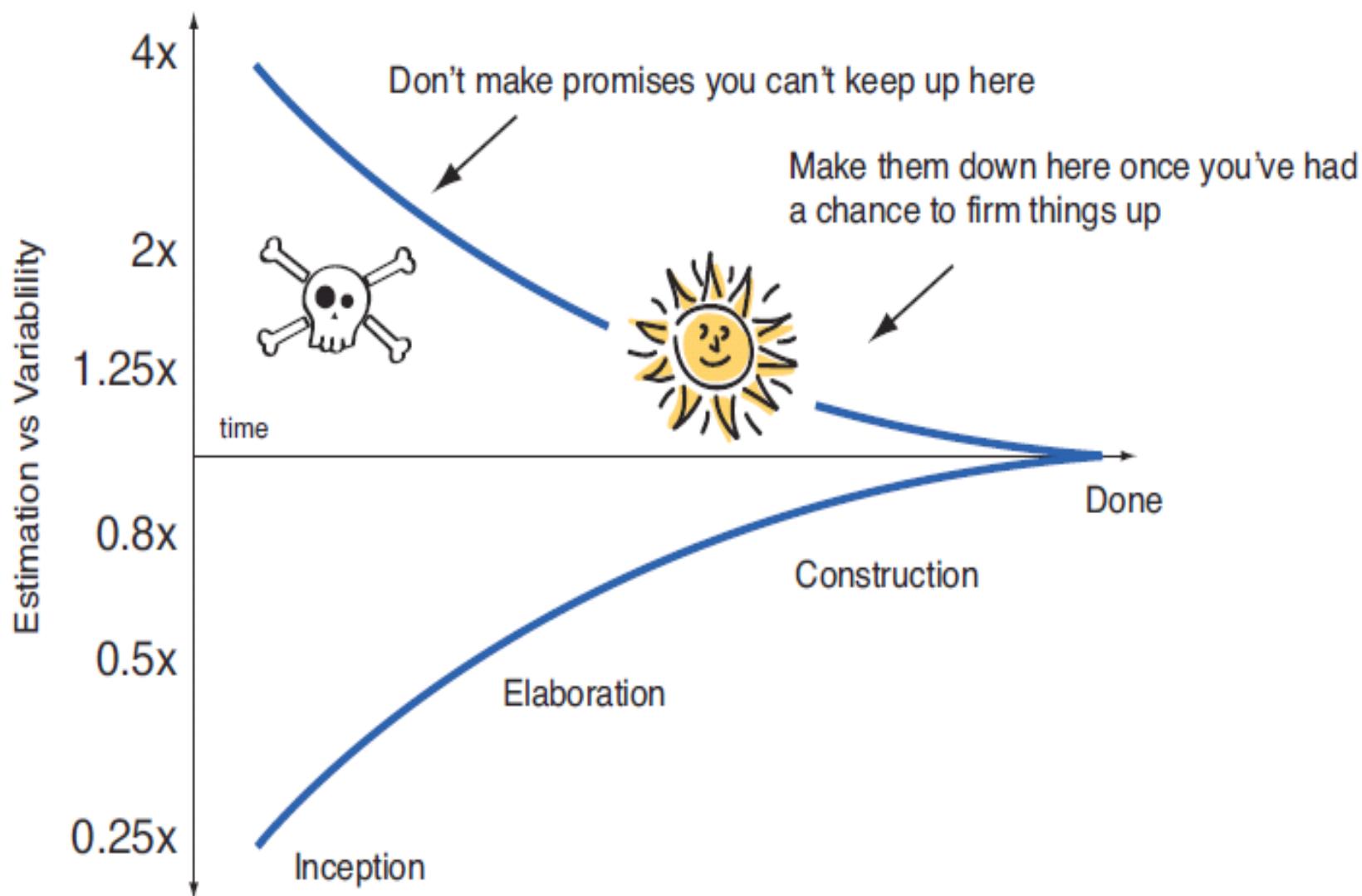
- **Step 4: Brainstorm everything else**
  - Kuvien piirtämisen ja siihen liittyvän brainstormauksen innoittamana saadaan yleensä kirjoitettua suuri joukko User storyjä
  - Kuvin ilmaistavien asioiden lisäksi mietitään muuta projektiin liittyvää ja kirjataan niitä vastaavat User storyt
- **Step 5: Scrub the list and make it shine**
  - Lopuksi siivotaan lista:
    - Poistetaan duplikaatit
    - Yhdistetään liian pienet toisiinsa liittyvät Storyt isommiksi
    - Kirjoitetaan User storyt koherentimpaan muotoon

# Backlogin priorisointi

- Product Backlog siis on priorisoitu lista User storyjä
  - Kuten muistamme priorisoinnin hoitaa *Product Owner*, XP:ssä käytetään suunnilleen samassa roolissa toimivasta henkilöstä nimitystä on-site customer
  - Prioriteetti määräää järjestyksen, missä ohjelmistokehittäjät toteuttavat ohjelmiston ominaisuuksia
  - Priorisoinnin motivaationa on pyrkiä maksimoimaan asiakkaan kehitettävästä ohjelmistosta saama hyöty/arvo
    - Tärkeimmät asiat halutaan toteuttaa mahdollisimman nopeasti ja näin saada tuotteesta alustava versio markkinoille niin pian kuin mahdollista
  - User storyjen priorisointiin vaikuttaa Storyn kuvaaman toiminnallisuuden asiakkaalle tuovan arvon lisäksi pari muutakin seikkaa
    - Storyn toteuttamiseen kuluva työmääärä
    - Storyn kuvaamaan ominaisuuteen sisältyvä tekninen riski
  - Ei ole siis kokonaistaloudellisesti edullista tehdä priorisointia välttämättä pelkästään perustuen asiakkaan User storyistä saamaan arvoon

# Estimointi eli User storyn toteuttamiseen kuluvan työmäärän arvointi

- User storyjen viemän työmäärän arvioimiseen on oikeastaan kaksi motivaatiota
  - Auttaa asiakasta priorisoinnissa
  - Mahdollistaa koko projektin viemän ajan summittainen arvointi
- Työmäärän arvioimiseen on kehitetty vuosien varrella useita erilaisia menetelmiä
- Kaikille yhteistä on se, että ne eivät toimi kunnolla, eli tarkkoja työmääräärvioita on mahdoton antaa
  - Joskus työmäärän arvioinnista käytetäänkin leikillisesti termiä *guesstimation*
- Mitä kauempana tuotteen/ominaisuuden valmistuminen on, sitä epätarkempia työmääräärviot ovat
  - Cone of uncertainty, ks. seuraava sivu



- Ketterät ohjelmistotuotantomenetelmät ottavat itsestäänselvyytenä sen, että estimointi on epävarmaa ja tarkentuu vasta projektin kuluessa
  - Koska näin on, pyritään vahvoja estimointiin perustuvia lupauksia aikatauluista olemaan tekemättä

# Suhteelliseen kokoon perustuva estimointi

- On huomattu, että vaikka ominaisuuksien toteuttamiseen menevän tarkan ajan arvioiminen on vaikeaa, osaavat ohjelmistokehittäjät jossain määrin arvioida eri tehtävien vaatimaa työmääriä suhteessa toisiinsa
- Esim.
  - User storyn *Tuotteen lisääminen ostoskoriin* toteuttaminen vie yhtä kauan kuin User storyn *Tuotteen poistaminen ostoskorista* toteuttaminen
  - User Storyn *Ostoskorissa olevien tuotteiden maksaminen luottokortilla* toteuttaminen taas vie noin kolme kertaa kauemmin kun edelliset
- Ketterissä menetelmissä käytetäänkin yleisesti suhteelliseen kokoon perustuvaa estimointia
- "yksikkönä" arvioinnissa on yleensä **Story point**
  - Ei yleensä vastaa mitään todellista tuntimääriä
  - Biershop-projektissa voitaisiin esim. kiinnittää että User storyn *Tuotteen lisääminen olutkoriin* estimaatti on 1 Story point, muita voidaan sitten verrata tähän, eli *Ostoskorissa olevien tuotteiden maksaminen luottokortilla* estimaatiksi tulisi 3 Story pointia

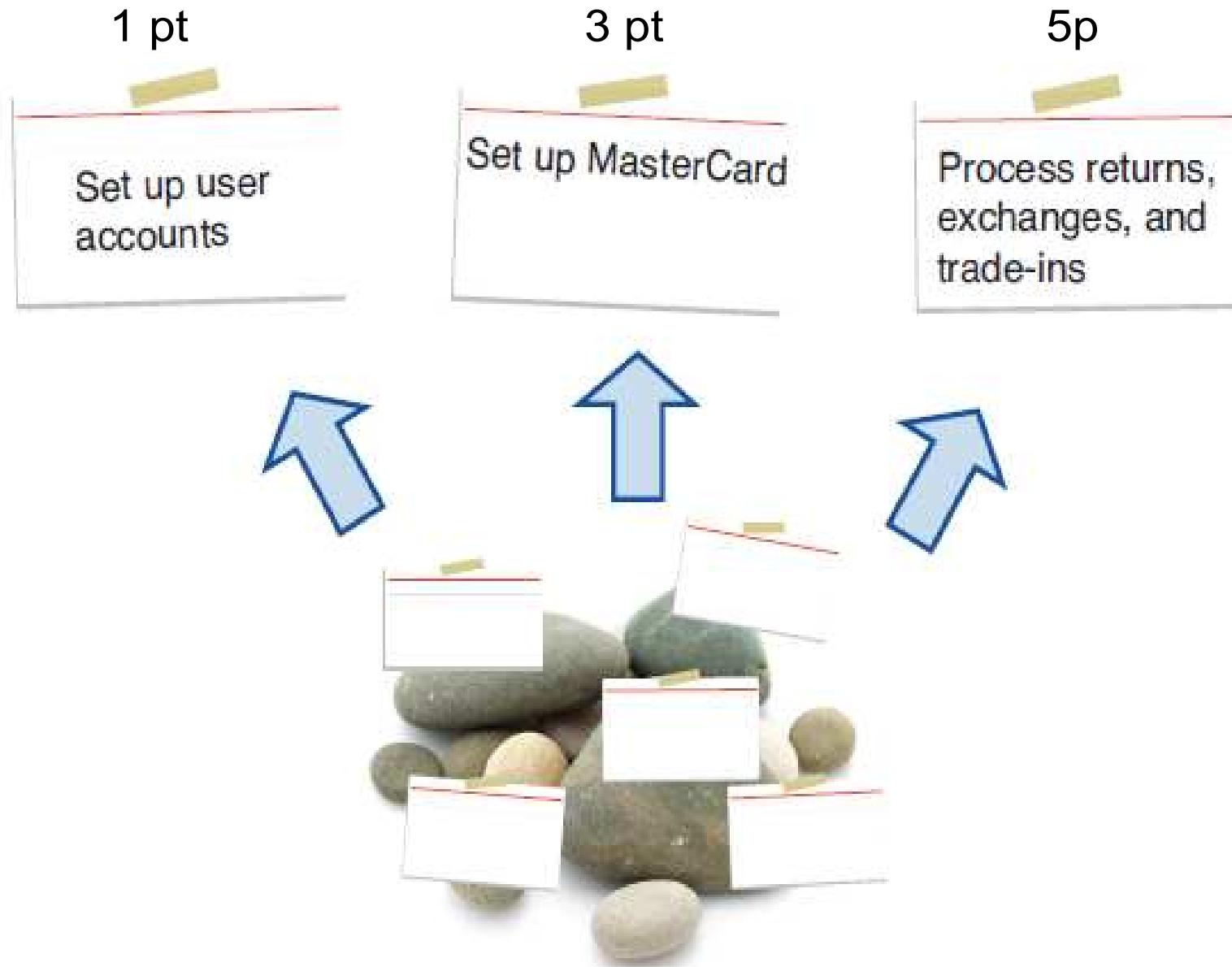
# Suhteelliseen kokoon perustuva estimointi

- Kun estimoitavana on suuri määrä User storyjä
  - Esimerkki Rasmussenin kirjasta Agile samurai



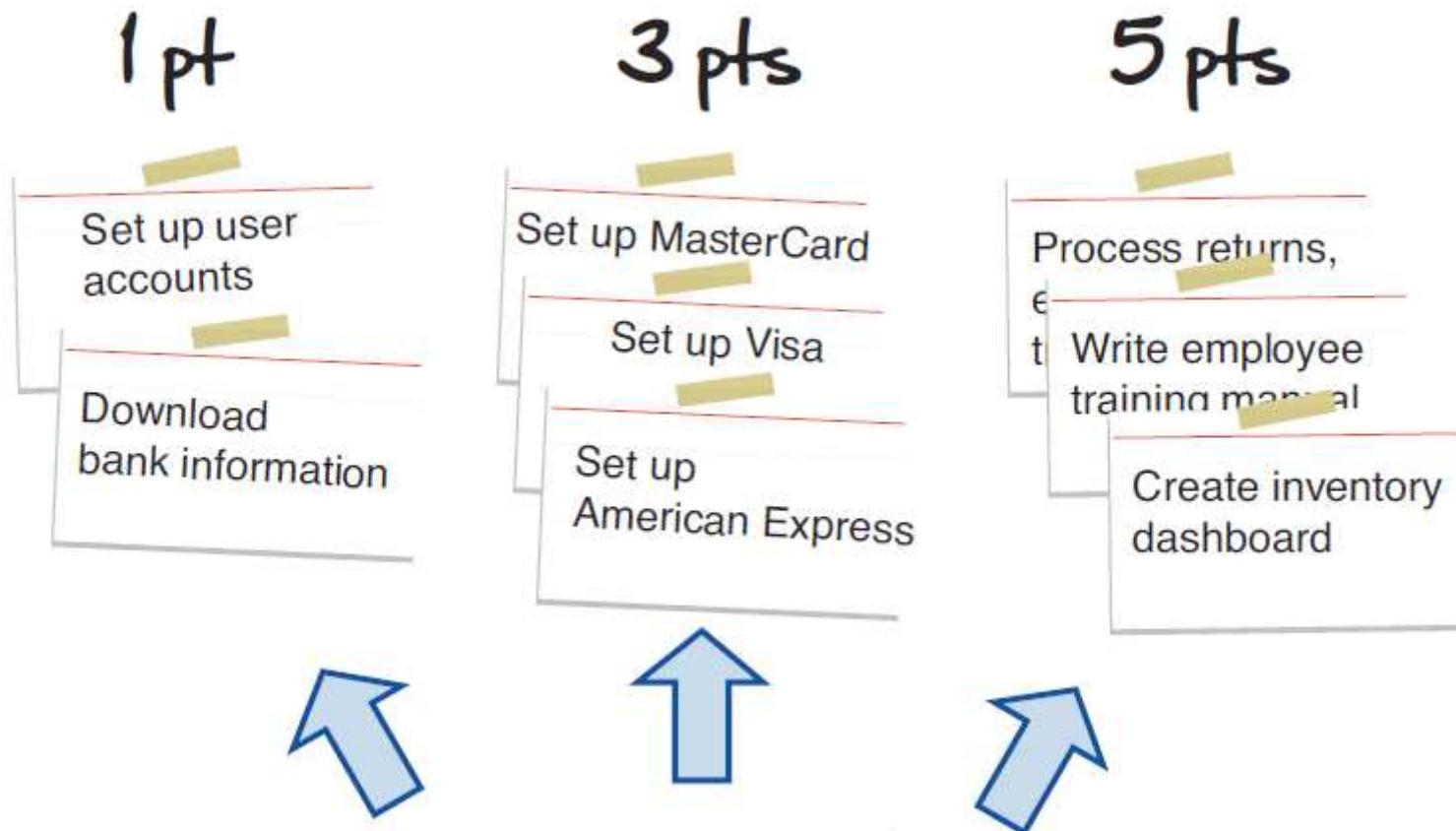
# Suhteelliseen kokoon perustuva estimointi

- saattaa olla kannattavaa arvioida ensin muutama hieman erikokoinen Story ja valita nämä referensseiksi



# Suhteelliseen kokoon perustuva estimointi

- Ja arvioida muut User storyt näiden suhteen



# Kuka suorittaa estimoinnin?

- Estimointi tapahtuu **aina ohjelmistokehitystiimin toimesta**
- Product ownerin on oltava läsnä tarkentamassa estimoitaviin User storeihin liittyviä vaatimuksia
- Usein estimointia auttaa User storyn pilkkomisen teknisiin työvaiheisiin
  - Esim. User story *Tuotteen lisääminen ostoskorioon*, voisi sisältää toteutuksen kannalta seuraavat tekniset tehtävät (task):
    - tarvitaan sessio, joka muistaa asiakkaan
    - domain-olio ostoskorin ja ostoksen esittämiseen
    - html-näkymää päivitettyvä tarvittavilla painikkeilla
    - Kontrolleri painikkeiden käsittelyyn
    - yksikkötestit kontrollerille ja domain-olioille
    - hyväksymätestien automatisointi
  - Työvaiheisiin pilkkomisen saattaa vaatia myös hieman suunnittelua, esim. täytyy miettiä, miten ohjelman rakennetta on muokattava, jotta uusi toiminnallisuus saadaan järkevästi toteutettua
  - Jos kyseessä on samantapainen toiminnallisuus kuin joku aiemmin toteutettu, voi estimointi tapahtua ilman User storyn vaatimien erillisten työvaiheiden miettimistä

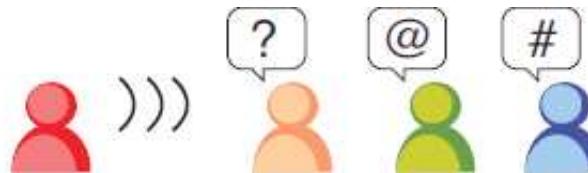
# Estimoinnista

- Estimointi on joka tapauksessa suhteellisen epätarkkaa, joten estimoinnin on tarkoitus tapahtua nopeasti
  - yhden User storyn estimointiin kannattaa käyttää aikaa korkeintaan 15 minuuttia, jos se ei riitä, on todennäköistä että Storya ei tunneta vielä niin hyvin että se kannattaisi estimoida
- Kuten viime viikolla mainitsimme, määritellään ketterissä projekteissa yleensä ns. "definition of done"
- Estimoinnissa tulee arvioida User storyn viemä aika **definition of doneen tarkkuudella**, tämä sisältää yleensä kaiken Storyn toteuttamiseen liittyvän:
  - määrittely, suunnittelu, toteutus, automatisoitujen tekstien tekeminen, testaus, integrointi ja dokumentointi
- Äsknen mainitsimme että Story point ei vastaa yleensä mitään aikayksikköä
  - Jotkut kuitenkin mitoittavat Story Pointin ainakin projektin alussa "ideal working day:n" suuruiseksi, eli työpäiväksi johon ei sisällä mitään häiriötekijöitä
  - Useimmat auktoriteetit suosittelevat olemaan sotkematta Story pointeja päiviin
  - ks. esim. <http://blog.crisp.se/2008/12/05/tomasbjorkholm/1228470417545>

# Planning poker

- Hyvänen periaatteena pidetään että kaikki tiimin jäsenet osallistuvat estimointiin
  - Tiimille syntyy yhtenäinen ymmärrys User storyn sisällöstä
- *Planning poker* on eräs suosittu tapa estimoinnin tekemiseen

1. Customer reads story.

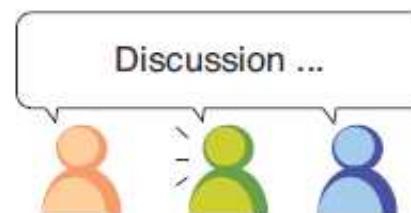


Development team asks questions

2. Team estimates.  
This includes testing.



3. Team discusses.



4. Team estimates again.  
Repeat until consensus reached.



# Planning poker

- Käydään läpi Backlogissa olevia User storyja yksi kerrallaan
- Asiakas lukee User storyn sisällön ja selittää tarkemmin Storyn luonnetta ja vaatimuksia
- Tiimi keskustelee Storystä, miettii kenties Storyn jakautumista teknisiin työvaiheisiin
- Kun kaikki kokevat olevansa valmiina arvioimaan, jokainen kertoo arvionsa (yksikkönä siis Story point)
- Usein tämä vaihe toteutetaan siten, että käytössä on pelikortteja, joilla on estimaattien arvoja, esim 1, 2, 5, 10, ... ja kukin estimointiin osallistunut näyttää estimaattinsa yhtä aikaa
- Jos estimaatit ovat suunnilleen samaa tasoa, merkataan estimaatti User storylle
- Jos seuraa eroavaisuutta, keskustelee tiimi eroavaisuuksien syistä
  - Voi esim. olla, että osa tiimin jäsenistä ymmärtää User storyn vaatimukset eri tavalla ja tämä aiheuttaa eroavaisuutta estimaatteihin
- Kun tiimi on keskustellut aikansa, tapahtuu uusi estimointikierros ja konsensus todennäköisesti saavutetaan pian

# Estimaattien skaala

- Koska estimointi on joka tapauksessa melko epätarkkaa, ei estimoinnissa ole tapana käyttää kovin tarkkaa skaalaa
- Yleistä on esim. käyttää ainostaan arvoja 1, 2, 3, 5, 10, 20, 40, 100 tai vastaavaa yläpäästä harvenevaa skaalaa
- Motivaationa se, että mitä suuremmasta kokonaisuudesta kyse, sitä vaikeampaa estimointi on, ja skaala yläpäässä on tarkoituksella harva, jotta estimaatit eivät antaisi valheellista kuvaa tarkkuudesta
- Joskus käytetään myös estimaattia *epic* jolla tarkoitetaan niin isoa tai huonosti ymmärrettyä User storyä että sitä ei voida vielä estimoida
- Alan suurin auktoriteetti Mike Cohn suosittlee käyttämään skaalaa 1, 2, 3, 5, 8 tai 1, 2, 4, 8 ja antamaan sitä suuremmille estimaatti *epic*



Ohjelmistotuotanto

Luento 4

21.3.

Vaatimusmäärittely ketterässä prosessimallissa  
nopea kertaus

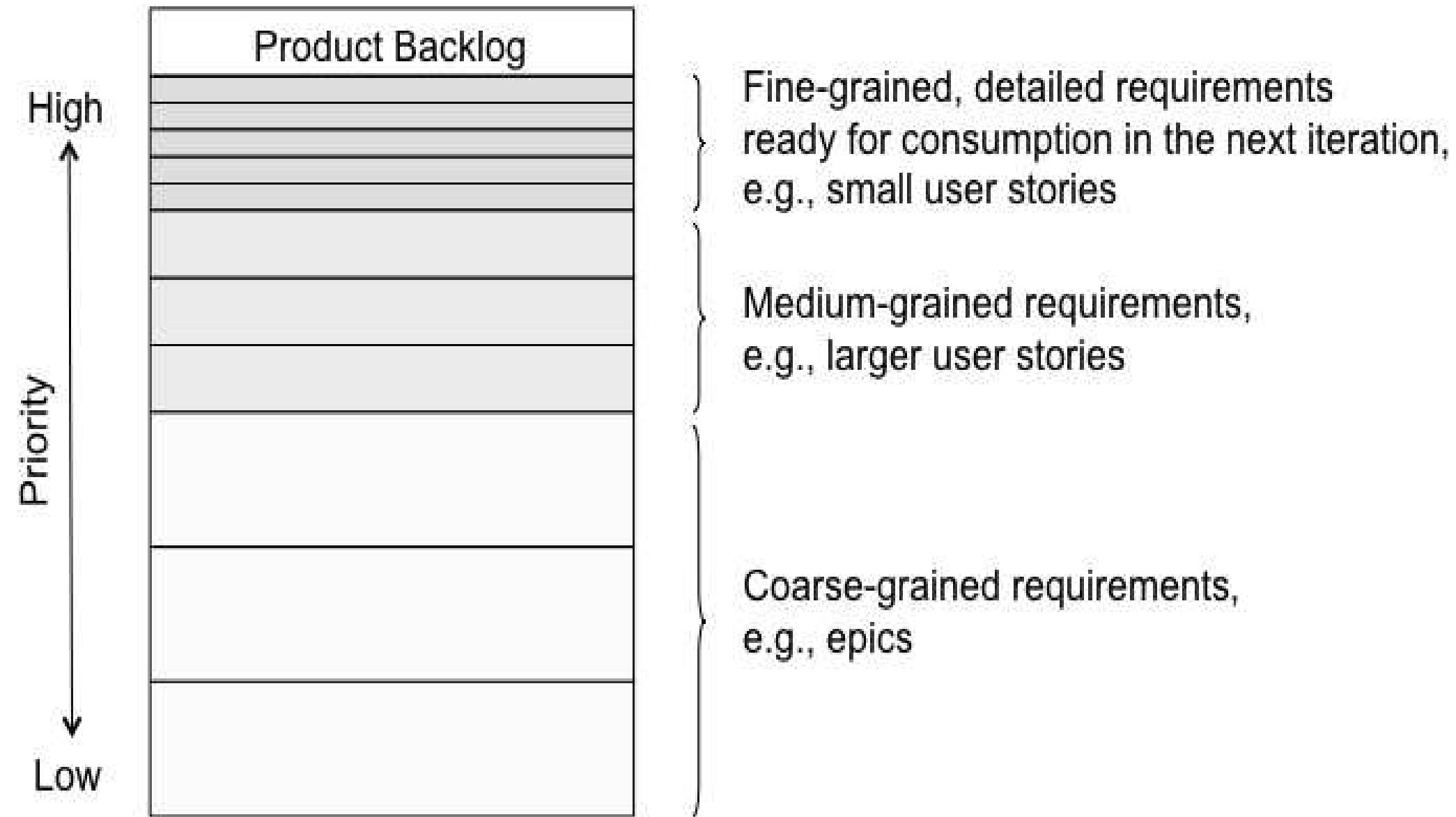
# Nopea kertaus eilisestä

- User story
  - ”Määritelmä”: CCC
    - card
    - conversation
    - confirmation
  - INVEST
- Estimointi
  - Miksi?
  - Miten?
  - Kuka?
- Product Backlog
  - Kuka vastuussa?
  - Miten saadaan projektin alussa muodostettua?

# Hyvä product backlog on DEEP

- <http://www.romanpichler.com/blog/product-backlog/making-the-product-backlog-deep/>
- Mike Cohn lanseerasi lyhenteen DEEP kuvaamaan hyvän backlogin ominaisuuksia
  - **Detailed** appropriately
  - **Estimated**
  - **Emergent**
  - **Prioritized**
- **Detailed** appropriately eli sopivan detaljoitu:
  - Backlogin prioriteeteiltaan korkeimpien eli pian toteutettavaksi otettavien User Storyjen kannattaa olla suhteellisen pieniä ja näin tarkemmin estimoituja
  - Alemman prioriteetin User Storyt voivat vielä olla isompia ja karkeammin estimoituja

# Hyvä product backlog on DEEP



# Hyvä product backlog on DEEP

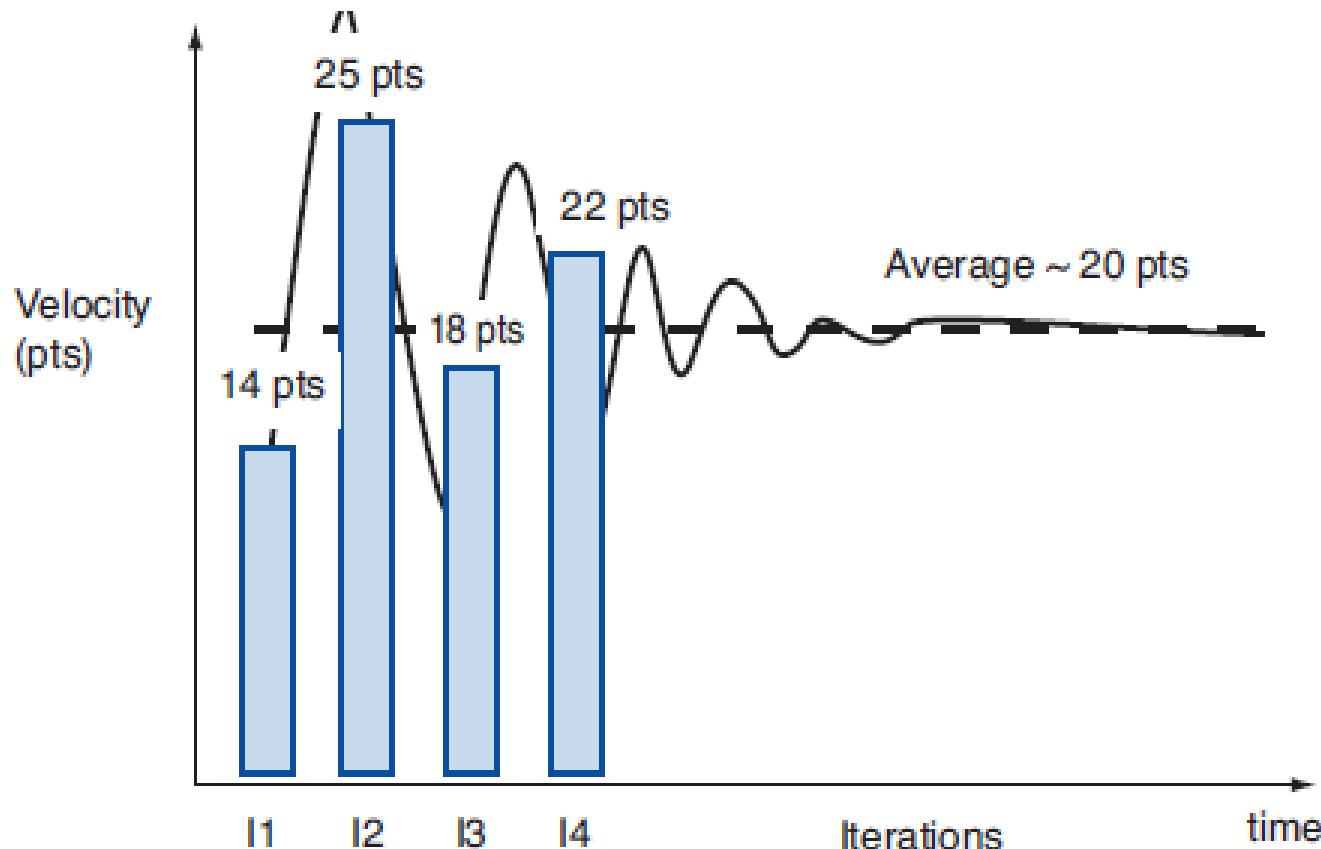
- DEEP ominaisuuksista **estimated** ja **prioritized** ovat meille tuttuja
- **Emergent** kuvaa backlogin muuttuvaa luonnetta:
  - The product backlog has an organic quality. It evolves, and its contents change frequently. New items emerge based on customer and user feedback, and they are added to the product backlog. Existing items are modified, reprioritized, refined, or removed on an ongoing basis.
- Muuttuvan luonteen takia *backlogia tulee hoitaa (**backlog grooming**)* projektin edetessä
  - Backlogiin lisätään uusia User storyja ja vanhoja tarpeettomaksi käyneitä poistetaan
  - Isoja User storyja pilkotaan tarpeentullen pienemmiksi (erityisesti prioriteetin kasvaessa täytyy isot Storyt pilkkoa pienemmiksi)
  - Backlogiin lisättäviä uusia User storyjä estimoidaan ja vanhojen Storyjen estimaatteja tarkastetaan ymmärryksen kasvaessa
  - Backlogin hoitamiseen osallistuu koko ohjelmistotuotantotiimi, pääasiallinen vastuu on Product Ownerilla
- <http://www.romanpichler.com/blog/product-backlog/grooming-the-product-backlog/>

# Julkaisun suunnittelu – release planning

- Estimoinnin toinen tarkoitus on, että se **mahdollistaa koko projektin viemän aikamäärän summittaisen arvioinnin** eli julkaisun suunnittelun (engl. release planning)
- Jos estimoinnin yksikkönä kuitenkin on abstrakti käsite *Story point*, miten estimaattien avulla on mahdollista arvioida projektin viemää aikamäärää?
- Kehitystiimin **velositeetti** (engl velocity) tarjoaa osittaisen ratkaisun tähän
- Velositeetilla tarkoitetaan Story pointtien määrää, minkä verran tiimi pystyy keskimäärin toteuttamaan yhden sprintin aikana
- Jos tiimin velositeetti on selvillä ja projektissa toteutettavaksi tarkoitetyt User storyt on estimoitu, on helppo tehdä alustava arvio projektin viemästä aikamäärästä
$$(\text{User storyjen estimaattien summa}) / \text{velositeetti} * \text{sprintin pituus}$$
- Projektin alkaessa velositeetti ei yleensä ole selvillä, ellei kyseessä ole jo yhdessä työskennellyt tiimi
- On kehitetty tapoja joiden avulla velositeetti voidaan yrittää etukäteen ennustaa
  - Hyvin epäluotettavia, emme käsittele niitä nyt

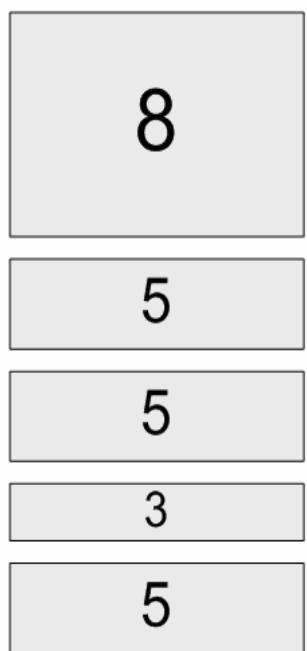
# Velositeetti

- Velositeetti vaihtelee tyypillisesti alussa melko paljon ja alkaa stabiloitumaan vasta muutaman sprintin päästä
  - Estimointi on aluksi vaikeampaa varsinkin jos sovellusalue ja käytetyt teknologiat eivät ole täysin tuttuja
- Tiimin velositeetti ja siihen perustuva projektin keston arvio alkaakin tarkentumaan pikkuhiljaa



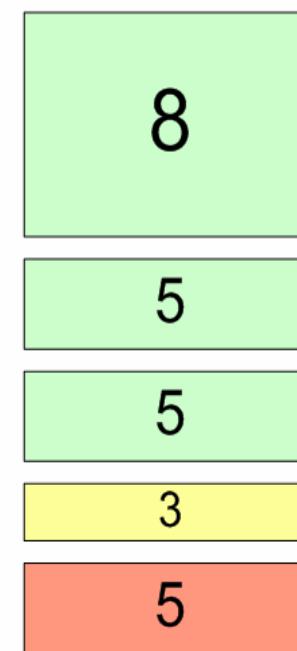
- Ketterissä menetelmissä on oleellista kuvata mahdollisimman realistisesti projektin etenemistä
- Tämän takia velositeettiin lasketaan mukaan ainoastaan **täysin valmiaksi** (eli Definition of Doneen mukaisesti) toteutettujen User storyjen Story pointit
  - "lähes valmiaksi" tehtyä työtä ei siis katsota ollenkaan tehdyksi työksi
  - [http://jamesshore.com/Agile-Book/done\\_done.html](http://jamesshore.com/Agile-Book/done_done.html)

Beginning of sprint



Estimated  
velocity = 26

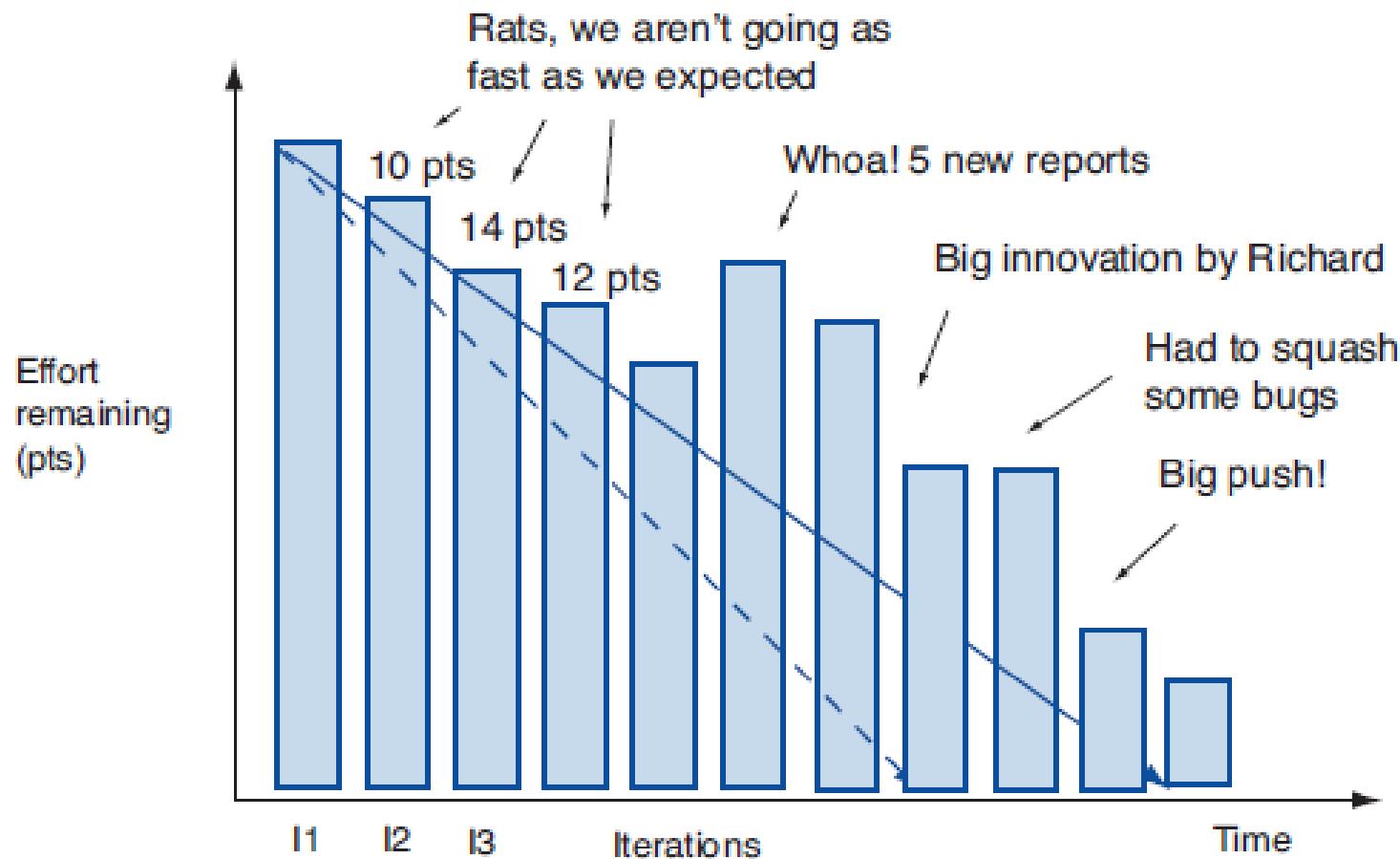
End of sprint



Actual  
velocity = 18

# Julkaisun suunnittelu – release planning

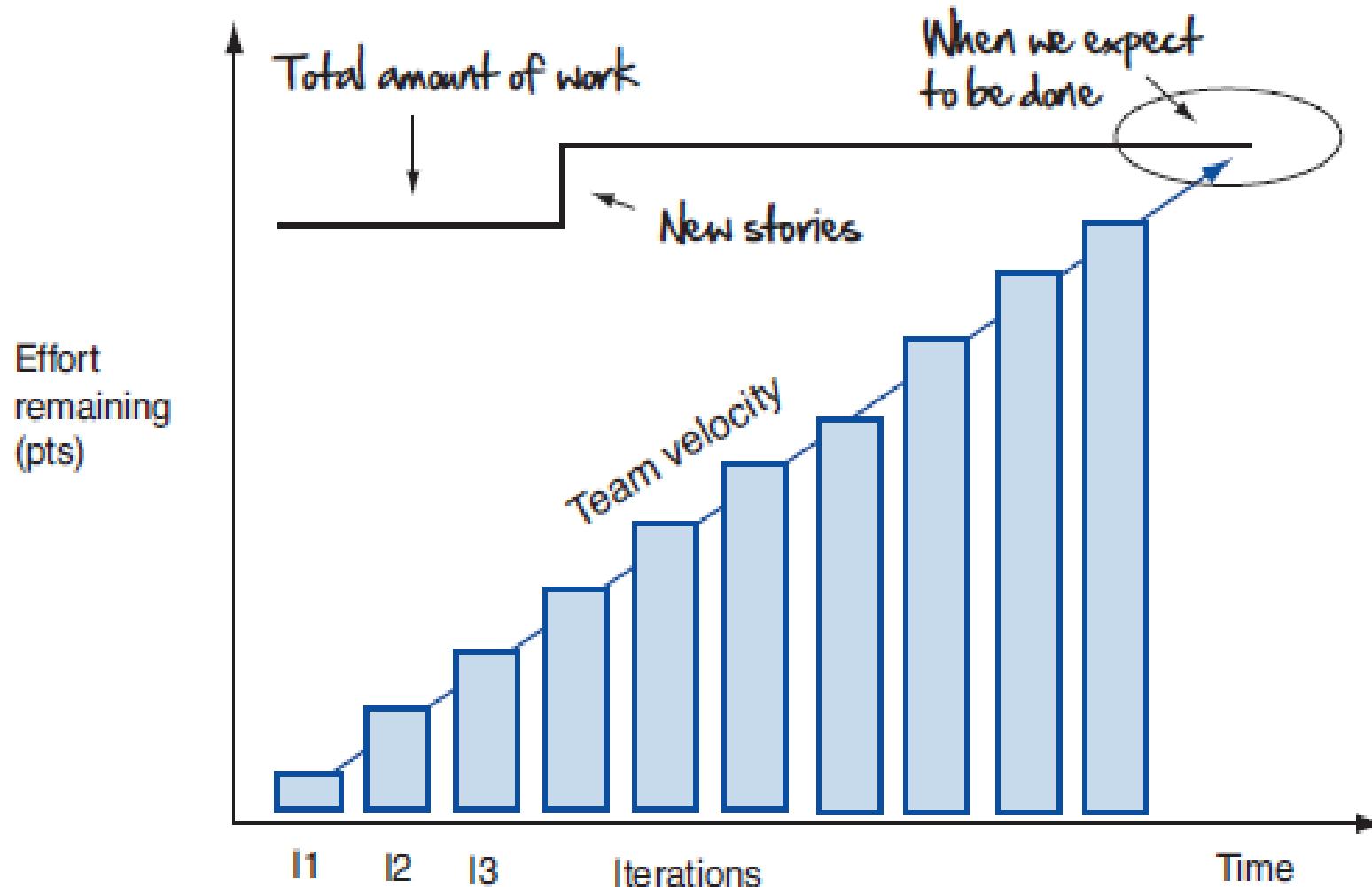
- Ketterän projektin etenemistä kuvataan yleensä Release Burndown -kaavion avulla
  - Aika etenee x-akselissa sprintti kerrallaan
  - y-akselilla on **jäljellä olevan työn määrä** Story pointteina mitattuna



- Ketterässä projektissa vaatimukset saattavat muuttua kehitystyön aikana, siksi jäljellä olevan työn määrä ei aina vähene

# Julkaisun suunnittelu – release planning

- Joskus käytetäänkin Burn Up -kaavioita joka tuo selkeämmin esiin kesken projektin etenemisen tapahtuvan työmääärän kasvun



# Sprintin suunnittelu

# Sprintin/iteraation suunnittelu

- Kertauksena viime viikolta: *Scrum määrittelee pidettäväksi ennen jokaista sprinttiä suunnittelupalaverin*
- Palaverin ensimmäinen tavoite on selvittää **mitä sprintin aikana tehdään**
  - Product Owner esittelee Product backlogin kärjessä olevat vaatimukset
  - Tiimin on tarkoitus olla riittävällä tasolla selvillä mitä vaatimuksilla tarkoitetaan
  - Tiimi valitsee tehtäväksi niin monta Backlogin vaatimukseen kuin se arvioi kykenevänsä sprintin aikana toteuttamaan Definition of Doneen määrittelemällä laatutasolla
- Sprintin aikana toteutettavien vaatimusten lisäksi asetetaan sprintin tavoite
- Suunnittelukokouksen toisena tavoitteena **miten sprintin tavoitteet saavutetaan**
  - Tiimi suunnittelee toteutettavaksi valitut vaatimukset tarvittavalla tasolla
  - Tarkennetaan nyt Sprintin suunnittelun ja läpivientiin liittyviä asioita
  - Lähteenä Kniberg Scrum and XP From the Trenches, luvut 3-6

# Sprintin suunnittelu

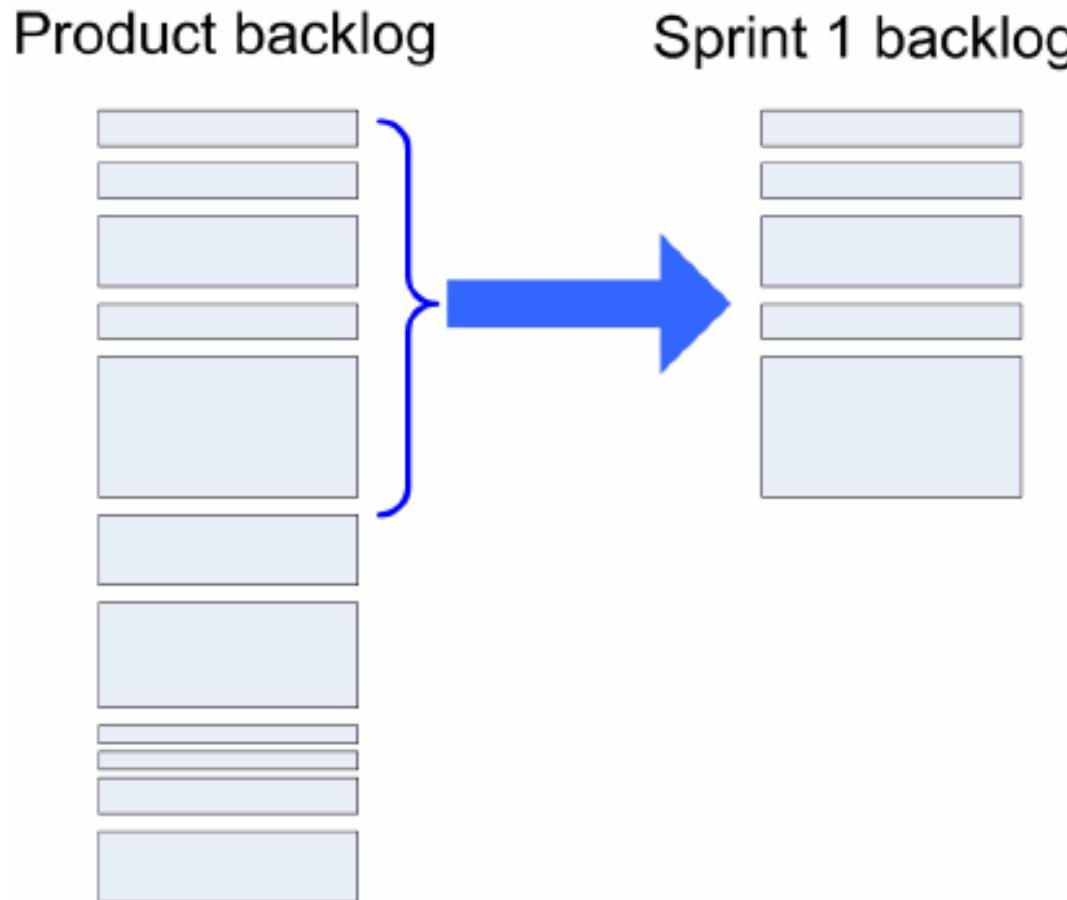
- Suunnittelun osallistuu Product Owner ja kehittäjät
- Lähtökohtana on sopivassa tilassa oleva eli DEEP Product backlog
  - Priorisoitu ja estimoitu
  - Korkeimman prioriteetin omaavat User storyt tarpeeksi pieniä ja Product Ownerin hyvin ymmärtämiä
- Suunnittelun yhteydessä määritellään **sprintin tavoite** (sprint goal)
  - Tavoite on jotain geneerisempää kuin yksittäisten backlogissa olevien User storyjen toteuttaminen
- Scrumin kehittäjä Ken Schwaber mainitsee 2002 kirjoitetussa kirjassaan asettavansa usein ensimmäisen sprintin tavoitteeksi: *"demonstrate a key piece of user functionality on the selected technology"*
- Seuraavalla sivulla Mike Cohnin määritelmä sprintin tavoitteesta

# Sprintin tavoite [Mike Cohn]

- A sprint goal is a short, one- or two-sentence, description of what the team plans to achieve during the sprint
- It is written collaboratively by the team and the product owner
- The following are typical sprint goals on an eCommerce application:
  - Implement basic shopping cart functionality including add, remove, and update quantities
  - The checkout process—pay for an order, pick shipping, order gift wrapping, etc.
- The sprint goal can be used for quick reporting to those outside the sprint
  - There are always stakeholders who want to know what the team is working on, but who do not need to hear about each product backlog item (User story) in detail
- The success of the sprint will later be assessed during the Sprint Review Meeting against the sprint goal, rather than against each specific item selected from the product backlog
- <http://www.mountaingoatsoftware.com/scrum/sprint-planning-meeting>

# Toteutettavien user storyjen valinta

- Sprintin tavoitteen asettamisen lisäksi tulee valita backlogista sprintin aikana toteutettavat User storyt
- Pääperiaate on valita ”sopiva määrä” backlogin korkeimmalle priorisoituja Storyjä



- Valituksi tulevat Storyt siirretään **sprintin backlogiin**

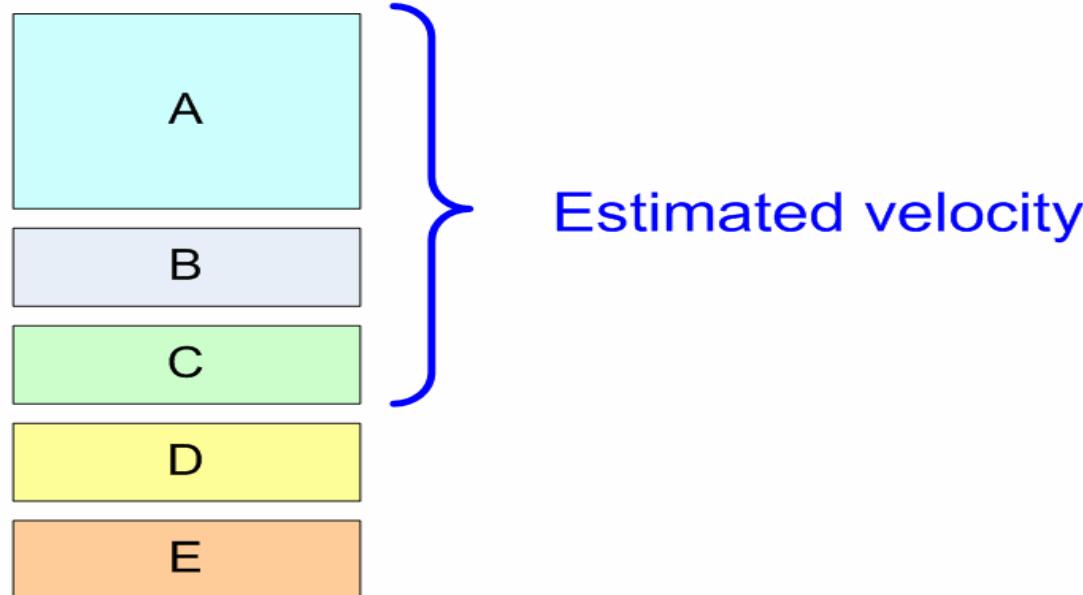
# Sprinttiin otettavien User storyjen määrä

- Kehitystiimi siis päättää kuinka monta User storyä sprinttiin otetaan toteutettavaksi
- Tapoja päättää kuinka monta User storyä sprinttiin otetaan on muutamia:
  - "perstuntuma": otetaan niin monta korkeimman prioriteetin Storyä kuin miin kaikki tiimiläiset tuntevat voivansa sitoutua
  - Jos storyt on estimoitu ja tiimin velositeetti tunnetaan, otetaan sprinttiin velositeetin verran storyjä
  - Edellisten yhdistelmä
- Jos User storyjä ei ole estimoitu tai velositeetti ei ole tiedossa, "perstuntumamenetelmä" lienee ainoa jota voidaan käyttää
  - Tässäkin menetelmässä tiimi saa valita vain sellaiseen määräänstoryjä, jotka se kokee voivansa toteuttaa kunnolla eli "definition of done" määrittelemän (eli suunnittelu, toteutus, automaattiset testit, testaus, integrointi, dokumentointi) mukaan valmiiksi
  - Velositeetin käsite ja estimaatihan huomioivat "definition of done"

# Toteutettavien user stroyjen valinta

- Jos tiimin velositeetti on tiedossa ja User storyt on estimoitu, otetaan Storyjä mukaan maksimissaan velositeetin verran
- Product ownerilla on mahdollisuksia vaikuttaa sprinttiin mukaan otettaviin User storyihin tekemällä uudelleenpriorisointia

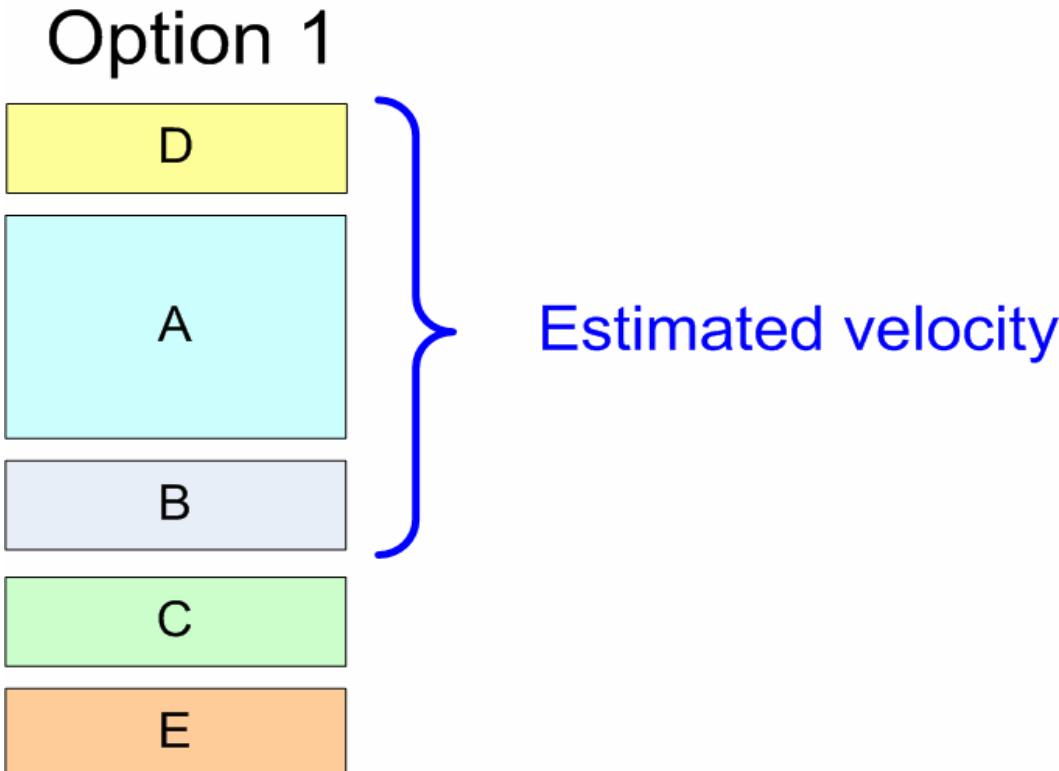
Product backlog



- Entä jos Product Owner haluaa storyn D mukaan sprinttiin?

# Uudelleenpriorisointi

- Product Owner nostaa D:n prioriteettia, C tippuu pois sprinttiin valittavien User Storyjen joukosta

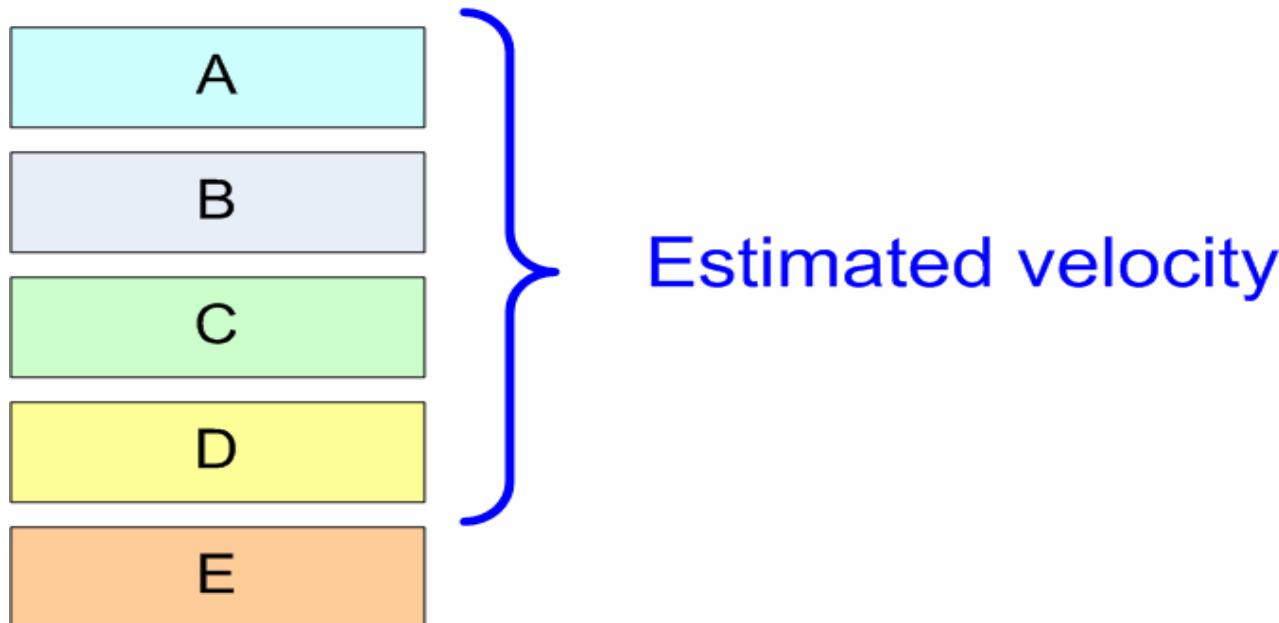


- Entä jos Product Owner haluaa Sprinttiin mukaan kaikki User Storyt A-D?

# User Storyn scopen pienentäminen

- *Jostain on luovuttava:* Product Owner pienentää User storyn A määrittelemää toiminnallisuutta, kehitystiimi estimoi pienennetyn A:n ja nyt A-D mahtuvat sprinttiin:

Option 2

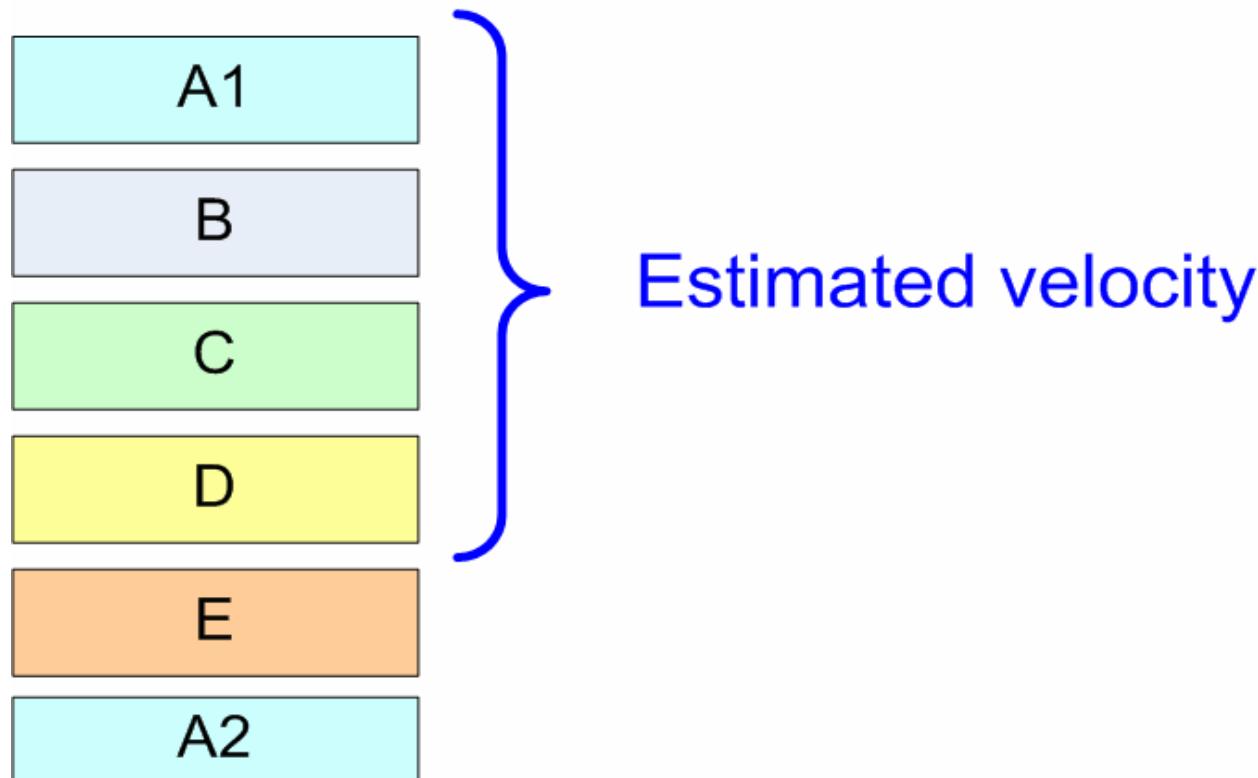


- Entä jos A:n toiminnallisuutta ei saa karsia ja silti Product Owner haluaa A-D:n mukaan sprinttiin?

# User Storyn jakaminen

- Ratkaisu on jakaa User story A kahteen pienempään osaan A1:n ja A2:n
  - A1 sisältää A:n tärkeimmät piirteet ja otetaan mukaan sprinttiin
  - A2 saa alempaan prioriteetin ja jää sprintin ulkopuolelle

Option 3



# User storyjen jakaminen

- Storyjen jakaminen pienemmiksi ei ole aloittelijalle, eikä aina ammattilaisellekaan helppoa
- Seuraavassa Richard Lawrencen ohjeita
  - <http://www.richardlawrence.info/2009/10/28/patterns-for-splitting-user-stories/>
- Good user stories follow Bill Wake's INVEST model. They're Independent, Negotiable, Valuable, Estimable, Small, and Testable
- Many new agile teams attempt to split stories by architectural layer: one story for the UI, another for the database, etc.
  - This may satisfy small, but it fails at independent and valuable.
- How small should stories be?
  - I recommend 6-10 stories per iteration, so how small is small enough depends on your team's velocity.
- Over my years with agile, I've discovered nine patterns for splitting user stories into good, smaller stories.

# User storyjen jakaminen

- Pattern #1: **Workflow Steps**

- As a content manager, I can publish a news story to the corporate website.

==>

- ... I can write and save a news story.
- ... I can edit a saved news story.
- ... I can publish a news story directly to the corporate website.
- ... I can publish a news story with editor review.
- ... I can view a news story on a editor review site.
- ... I can publish a news story from the editor review site to production

- Pattern #2: **Business Rule Variations**

- As a user, I can search for flights with flexible dates.

==>

- ... as “between dates x and y.”
- ... as “a weekend in December.”
- ... as “± n days of dates x and y.”

# User storyjen jakaminen

- Pattern #3: **Major Effort**
  - As a user, I can pay for my flight with VISA, MasterCard, Diners Club, or American Express.

==>

    - ... I can pay with VISA
    - ... I can pay with all four credit card types (VISA, MC, DC, AMEX) (given one card type already implemented).
- Pattern #4: **Simple/Complex**
  - As a user, I can search for flights between two destinations.

==>

    - ... when only direct flights used.
    - ... specifying a max number of stops.
    - ... including nearby airports.
    - ... using flexible dates.

# User storyjen jakaminen

- Pattern #6: **Data Entry Methods**
  - As a user, I can search for flights between two destinations.  
==>
  - ... using simple date input.
  - ... with a fancy calendar UI.
- Pattern #7: **Operations**
  - As a user, I can manage my account.  
==>
  - ... I can sign up for an account.
  - ... I can edit my account settings.
  - ... I can cancel my account

# User storyjen jakaminen

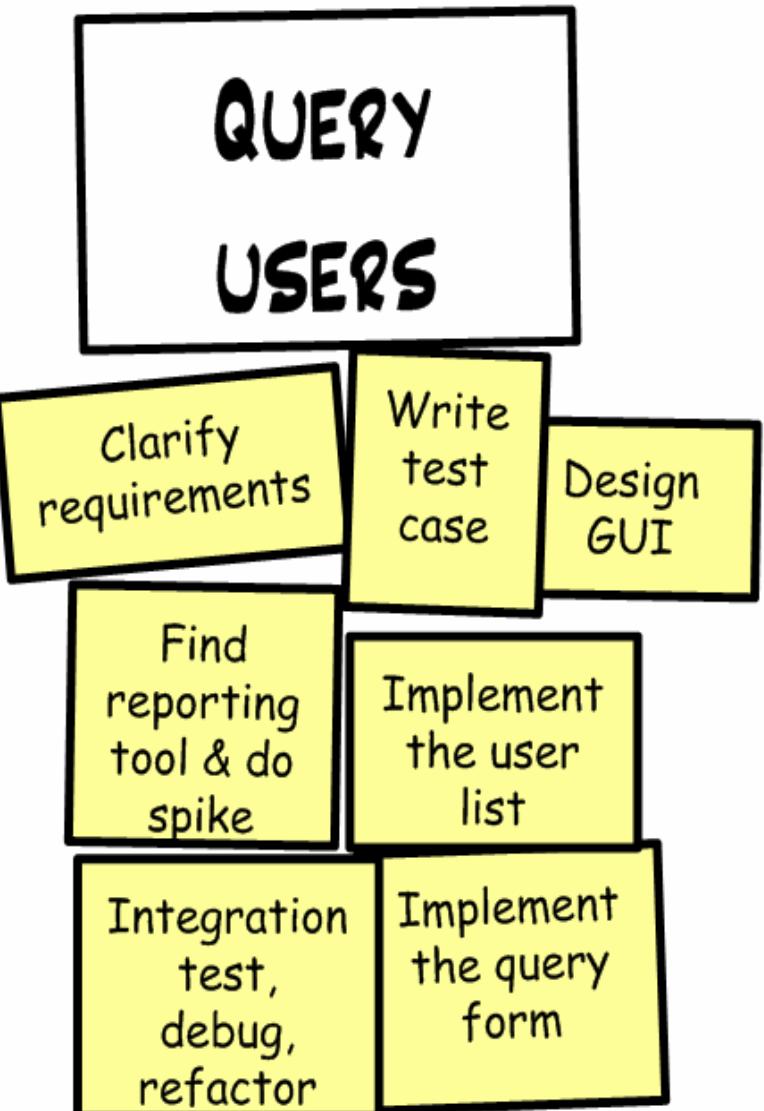
- Pattern #8: **Defer Performance**
  - As a user, I can search for flights between two destinations.

==>
  - ... (slow—just get it done, show a “searching” animation).
  - ... (in under 5 seconds).
- Pattern #9: **Break Out a Spike**
  - A story may be large not because it's necessarily complex, but because the implementation is poorly understood. In this case, no amount of talking about the business part of the story will allow you to break it up.  
**Do a time-boxed spike first to resolve uncertainty around the implementation.** Then, you can do the implementation or have a better idea of how to break it up.
  - As a user, I can pay by credit card.

==>
  - Investigate credit card processing.
  - Implement credit card processing.

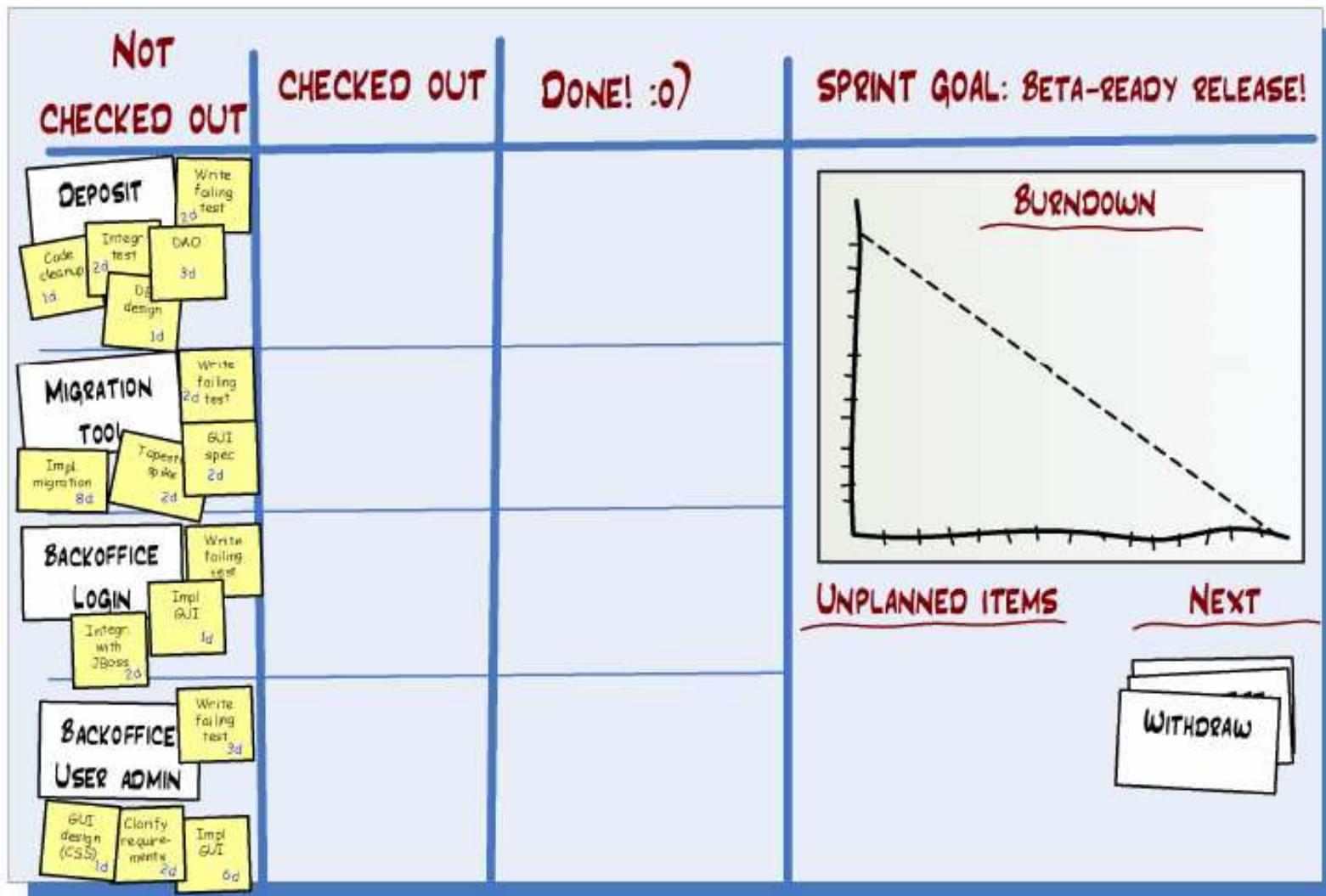
# Sprintin suunnittelun toinen vaihe

- Sprintin suunnittelun yhteydessä sprinttiin valituille User storyille tehdään karkean tason suunnittelu
  - Mietitään mitä teknisen **tason tehtäviä** (**Task**) on toteutettava, jotta User story saadaan valmiiksi
  - Suunnitellaan komponentteja ja rajapintoja karkealla tasolla
  - Huomioidaan User storyn aiheuttamat muutokset olemassa olevaan osaan sovelluksesta
  - Kaikkia Storyyn liittyviä taskeja ei sprintin suunnittelun aikana välttämättä löydetä
  - Uusia taskeja generoidaan tarvittaessa sprintin edetessä



# Sprint backlog

- Sprintin tehtävälistä eli Sprint backlog koostuu sprinttiin valitusta User storeista ja Storeihin liittyvistä tehtävistä eli Taskeista
- Backlog voi olla organisoitu "taskboardiksi":

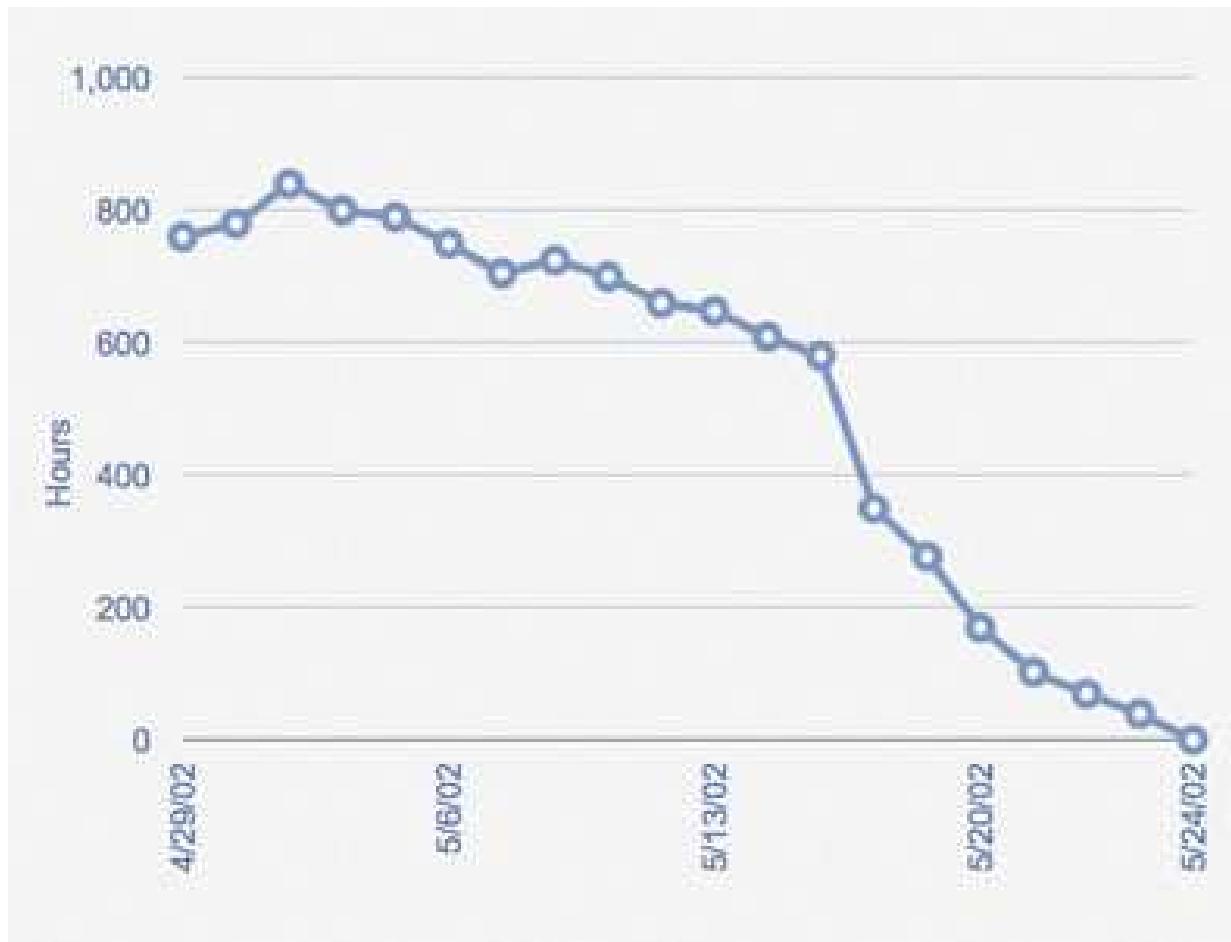


# Taskboard

- Kuvassa sprinttiin on valittu 4 User storyä ja taskboard on jaettu neljään kaistaan (swimlane), jokaiselle Storylle oma kaista
- Kuten arvata saattaa, jokaisen taskin on tarkoitus siirtyä sarakkeesta "not checked out" sarakkeeseen "done"
- **Sprintissä arvioidaan päivittäin jäljellä olevaksi arvioitua työmääriä**
- Taskien **jäljellä oleva työmääriä** arvioidaan yleensä **tunteina**
- Jokaiseen taskiin kirjataan sen arvioitu vielä jäljellä oleva työmääri
  - Jos käytössä on "manuaalinen" taskboard, kirjoitetaan arvio suoraan taskia edustavaan postit-lappuun
  - Arviota päivitetään joka päivä
  - Arvio voi nousta jos taski huomataankin työläämmäksi mitä alun perin ajateltiin
- On varsin tavallista, että uusia taskeja keksitään kesken sprintin
  - Uudet taskit saavat olla ainoastaan kehittäjätiiimin itse identifioimia menossa olevaan sprinttiin liittyviä töitä
- Eli sprintissä jäljellä oleva työaika-arvio voi kasvaa kesken sprintin!

# Sprintissä jäljellä olevan työmääärän arvointi

- Jokaisen taskin jäljellä olevan työn määrä arvioidaan esim. päivittäisessä scrum-palaverissa eli Daily Scrumissa
- Jäljellä olevaa työmääriää (tunteina mitattuna) visualisoidaan sprintin etenemistä kuvaavalla **burndown-käyrällä**
  - Tätä sprintin burndown:ia ei pidä sekottaa projektin burndown-käyrään!



Stuff that nobody is working on today

Stuff that somebody is working on today

Stuff that nobody will work on any more

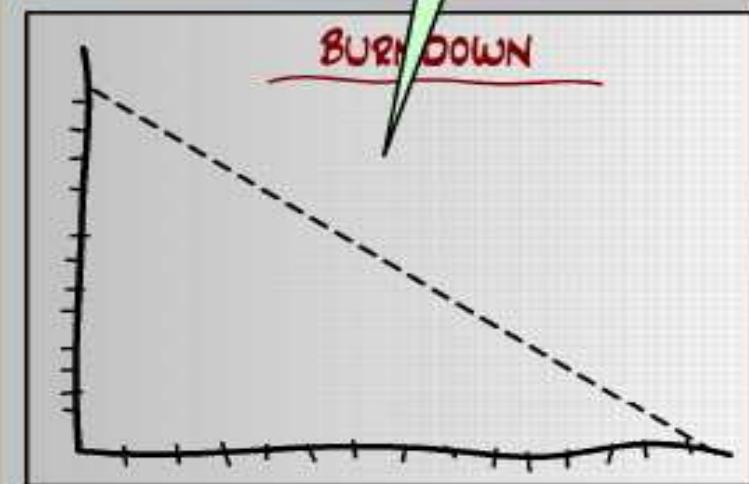
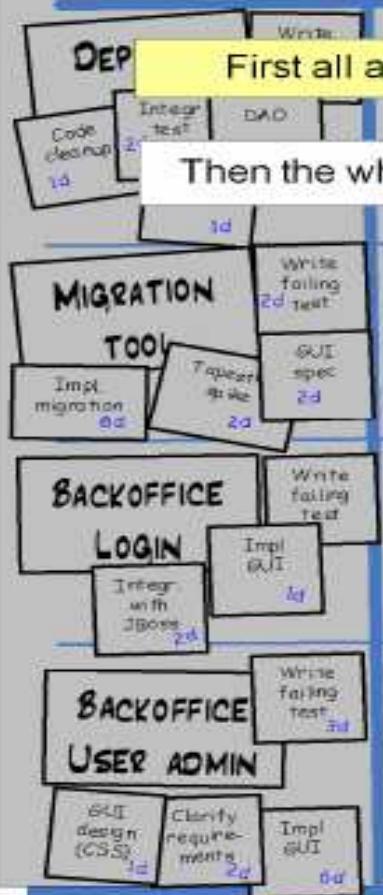
Manually plot a new point on the burndown every day after the daily scrum.

NOT  
CHECKED OUT

CHECKED OUT

DONE! :)

SPRINT GOAL: BETA READY RELEASE!



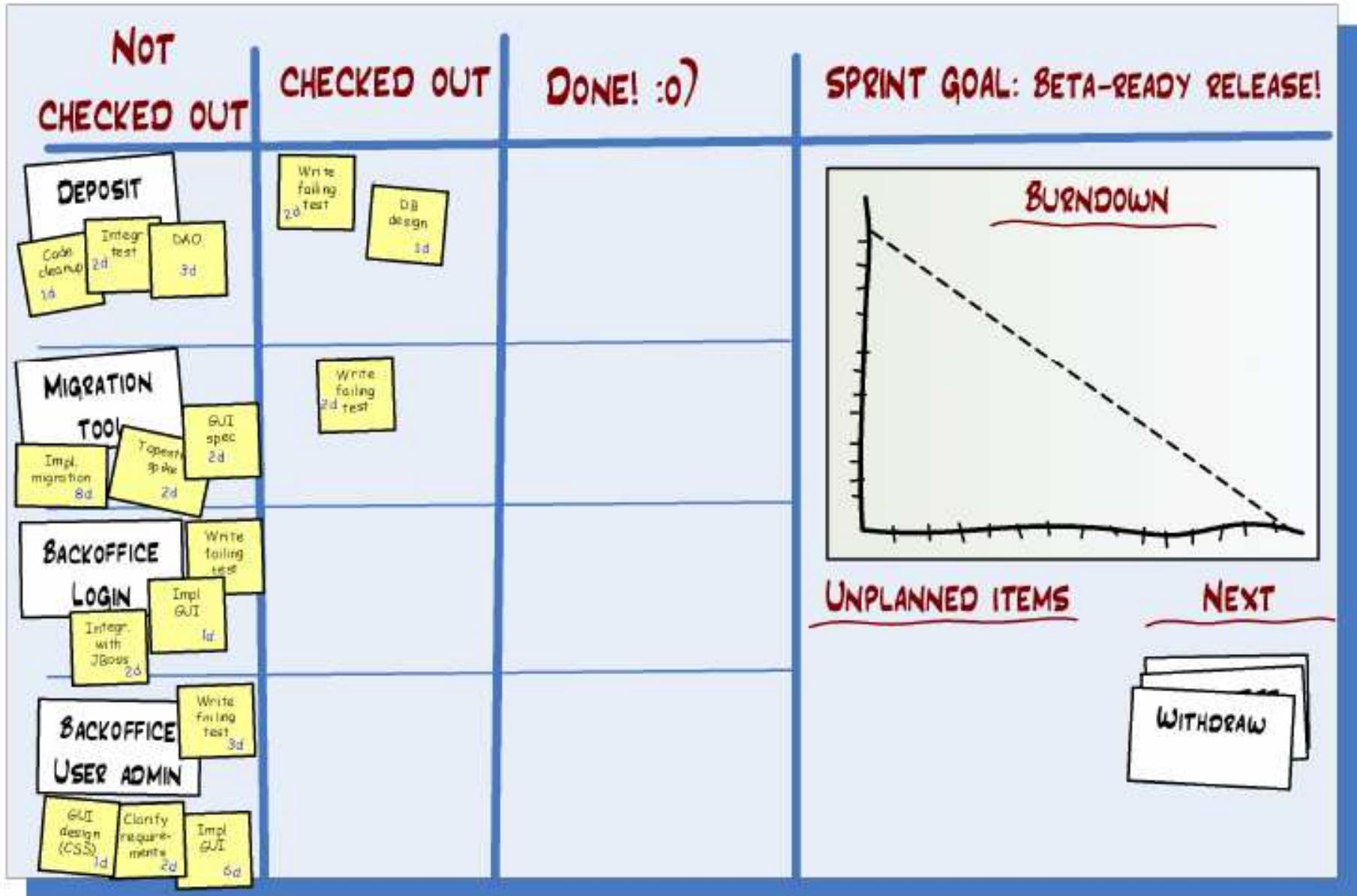
UNPLANNED ITEMS

NEXT

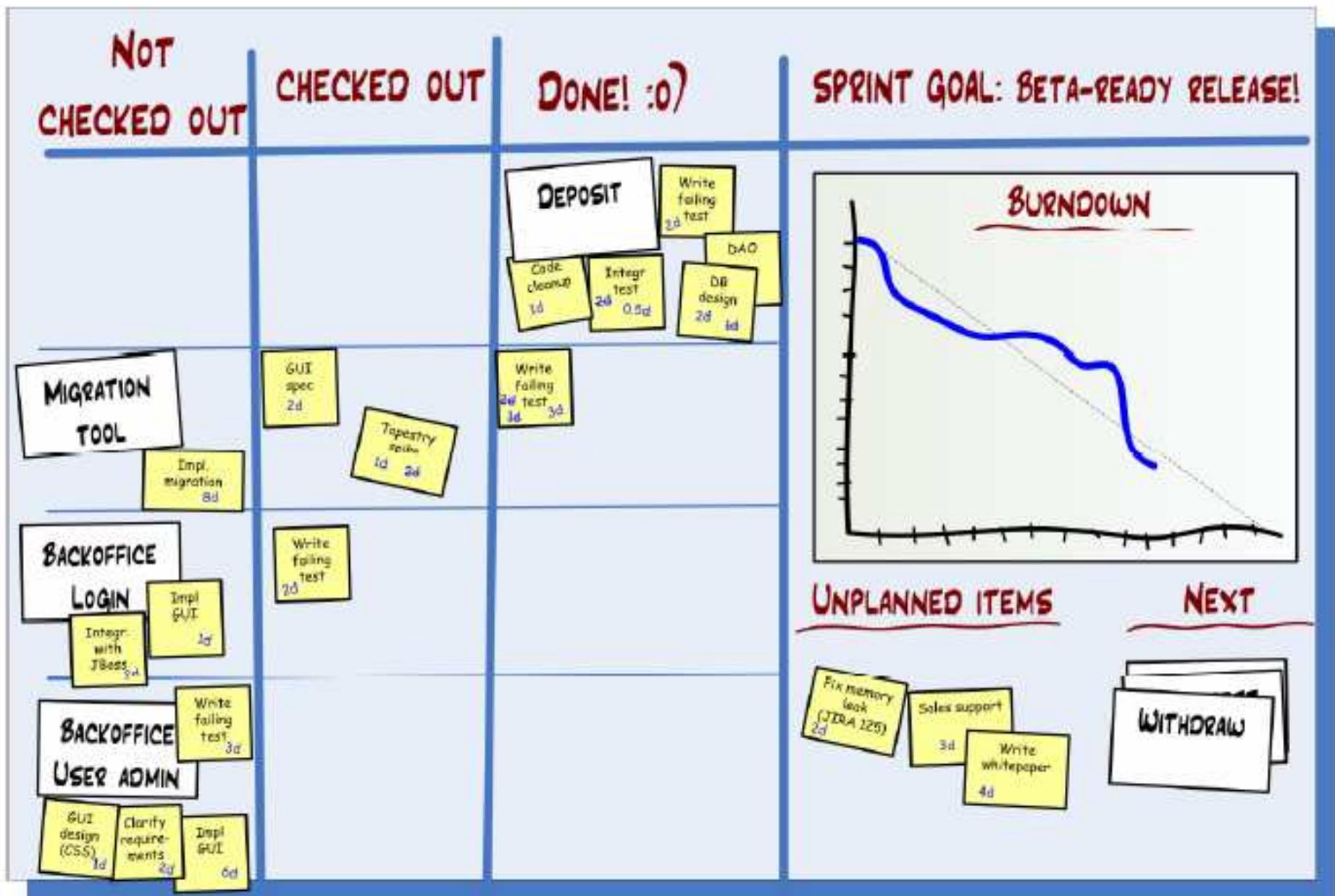


If all backlog items are completed before the sprint ends, add new ones from here.

# Tilanne sprintin alussa



# Ja puolen välin jälkeen



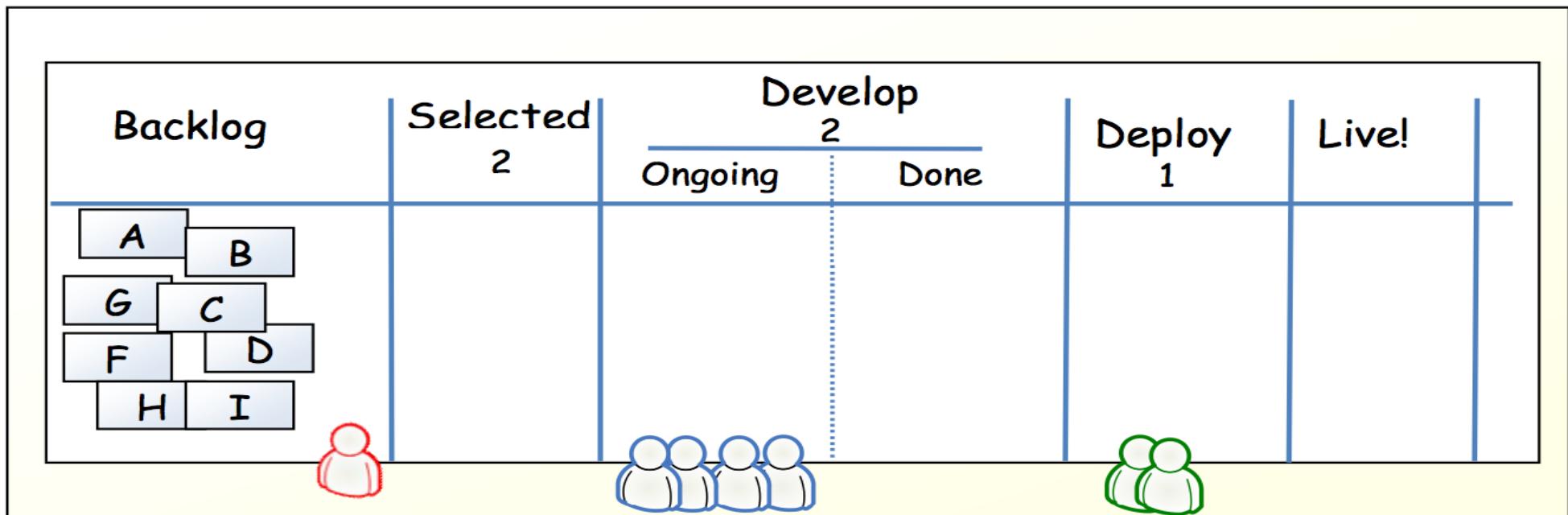
# Taskboardissa voi olla merkattu useampiakin työvaiheita

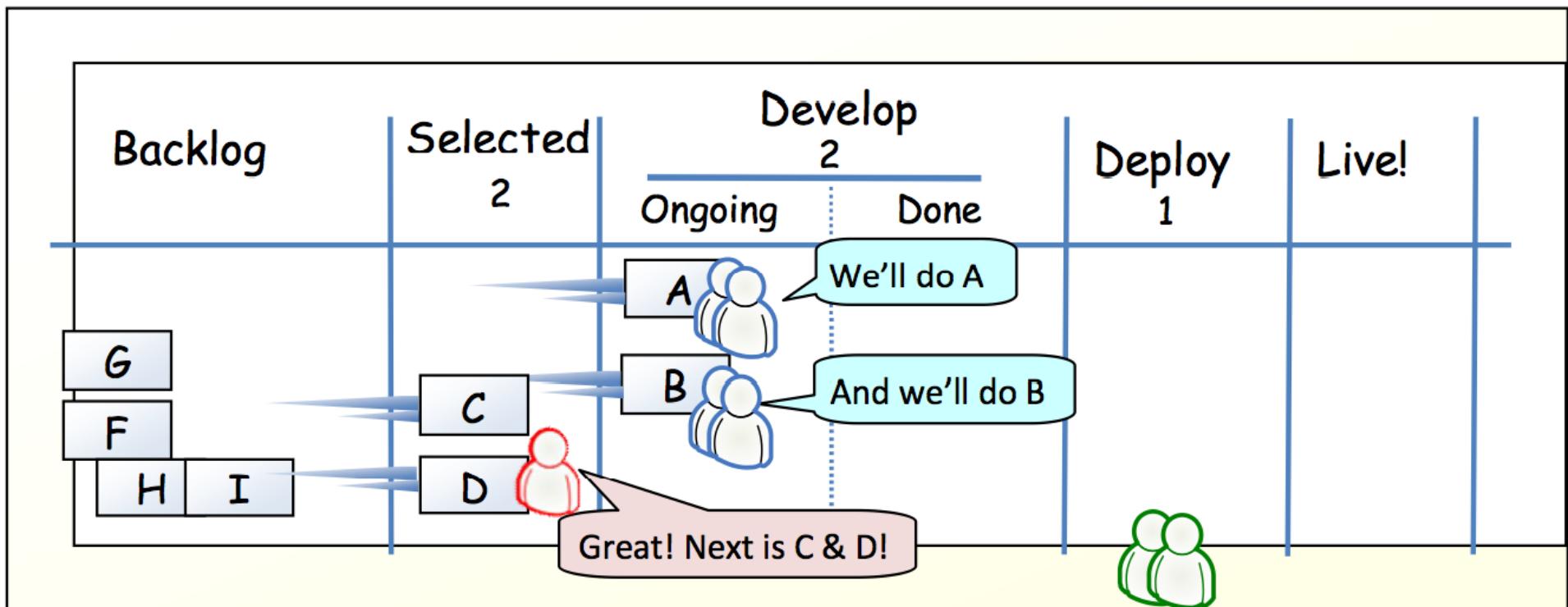
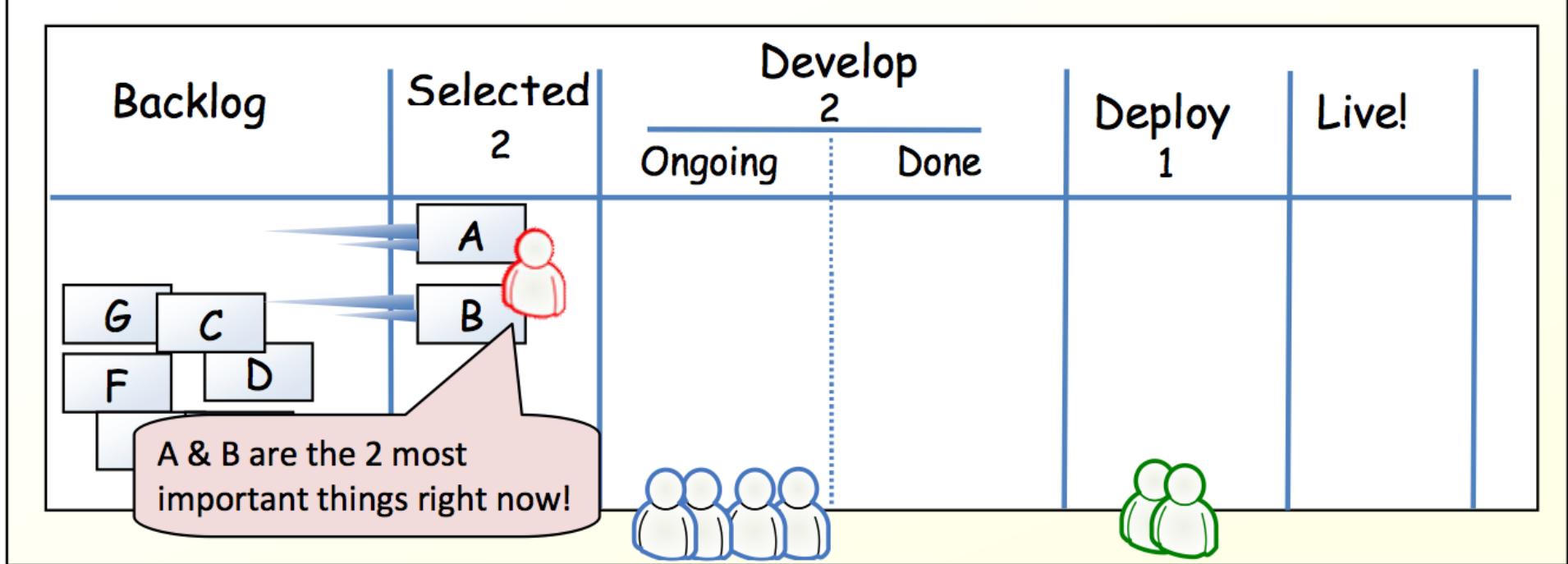
Story	To Do		In Process	To Verify	Done
As a user, I... 8 points	Code the... 9	Test the... 8	Code the... DC 4	Test the... SC 6	Code the... 8 Test the... SC 4 Test the... SC 8 Test the... SC 6
As a user, I... 5 points	Code the... 8	Test the... 8	Code the... DC 8		Test the... SC 4 Test the... SC 8 Test the... SC 6

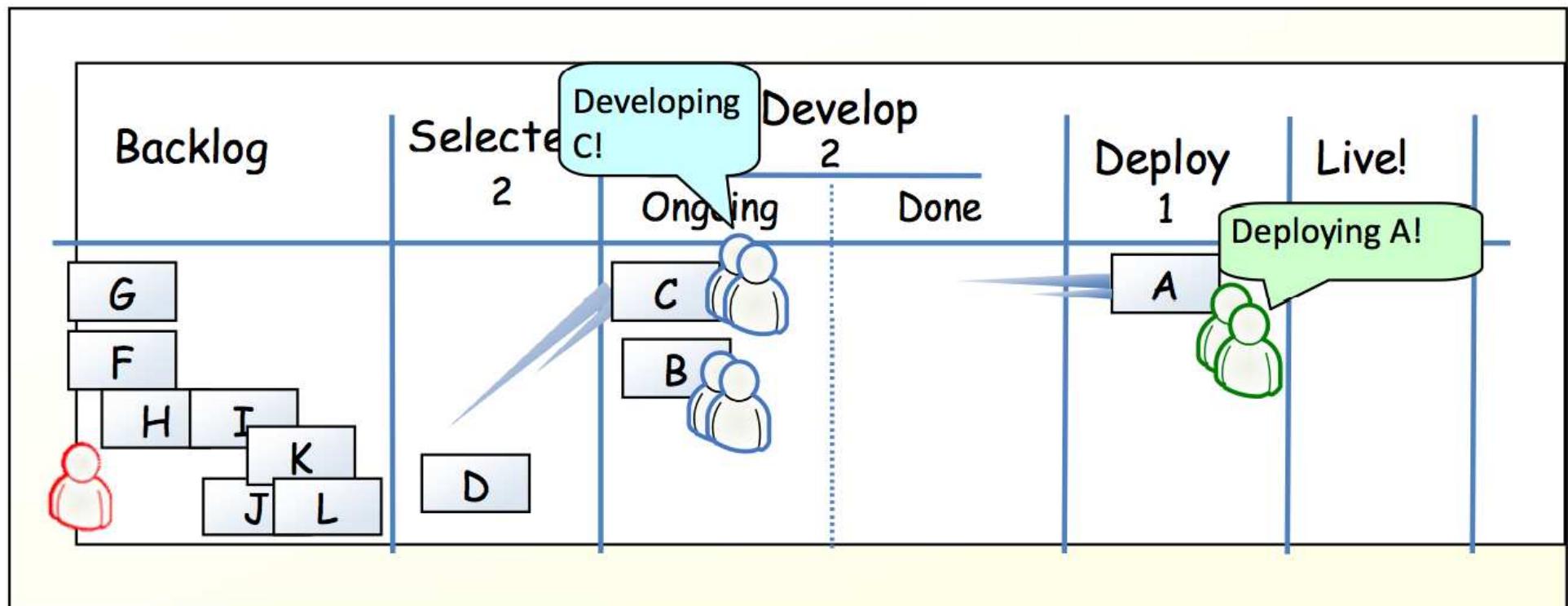
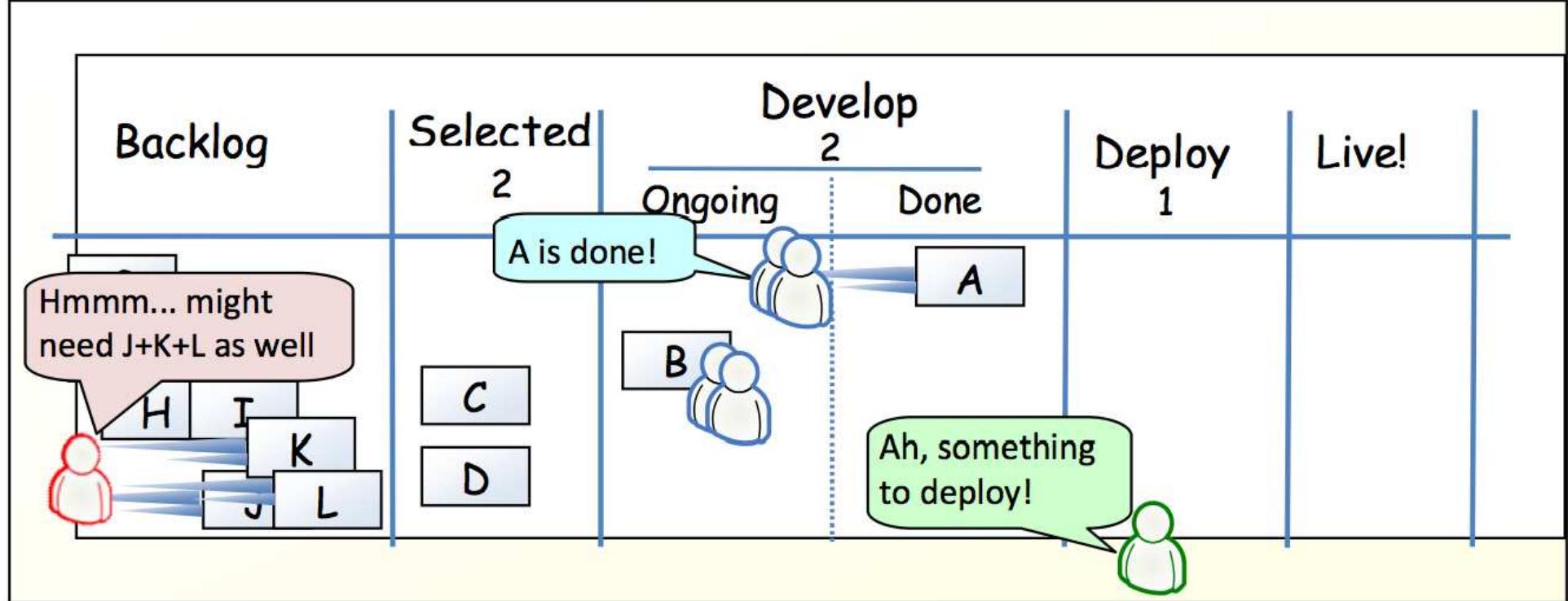
- Yhtä aikaa työn alla olevien taskien suuri määrä voi koitua Scrumissa ongelmaksi, sillä riski sille, että sprintin päätyttyä on paljon osittain valmiita User storyja kasvaa

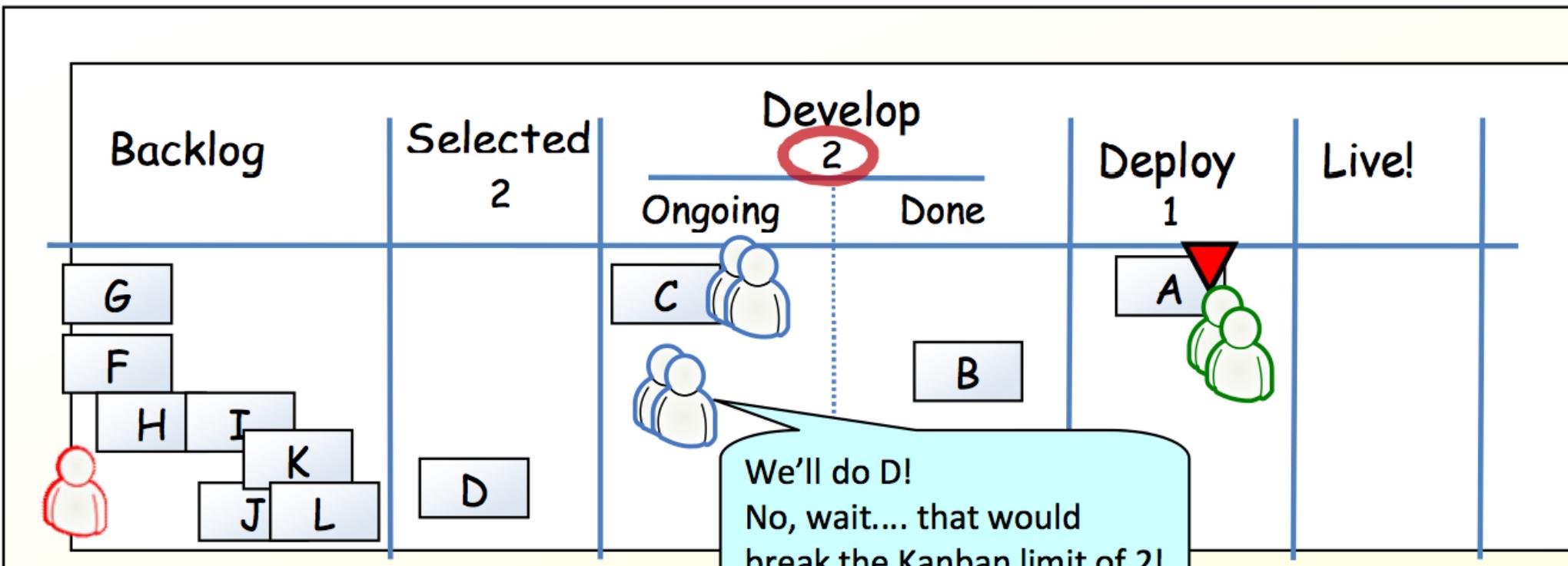
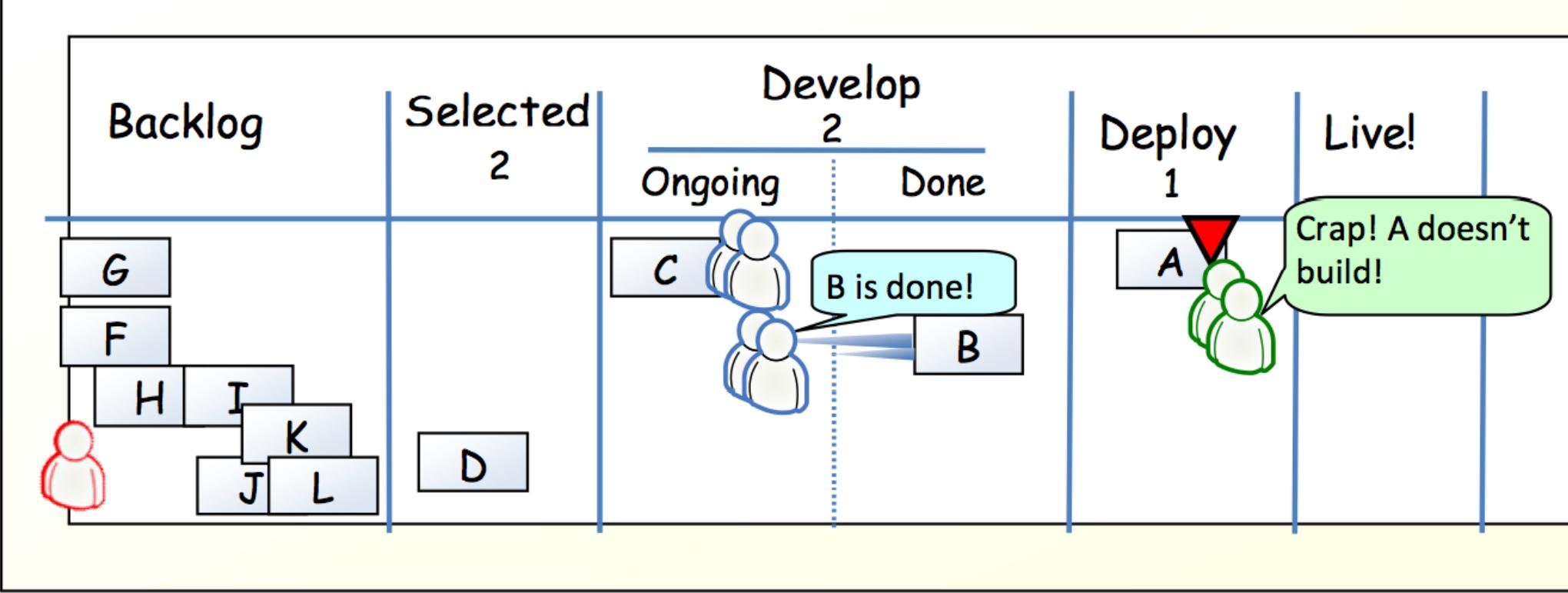
# Yhtäaikaa tehtävän työn rajoittaminen

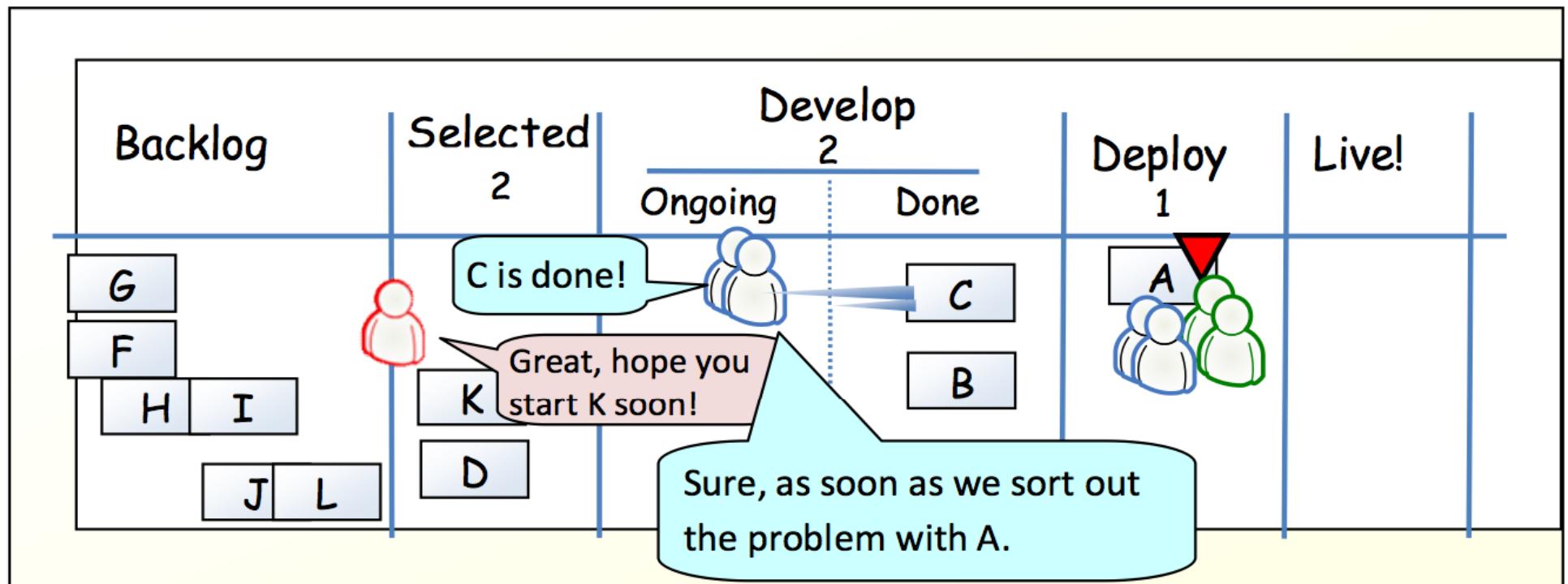
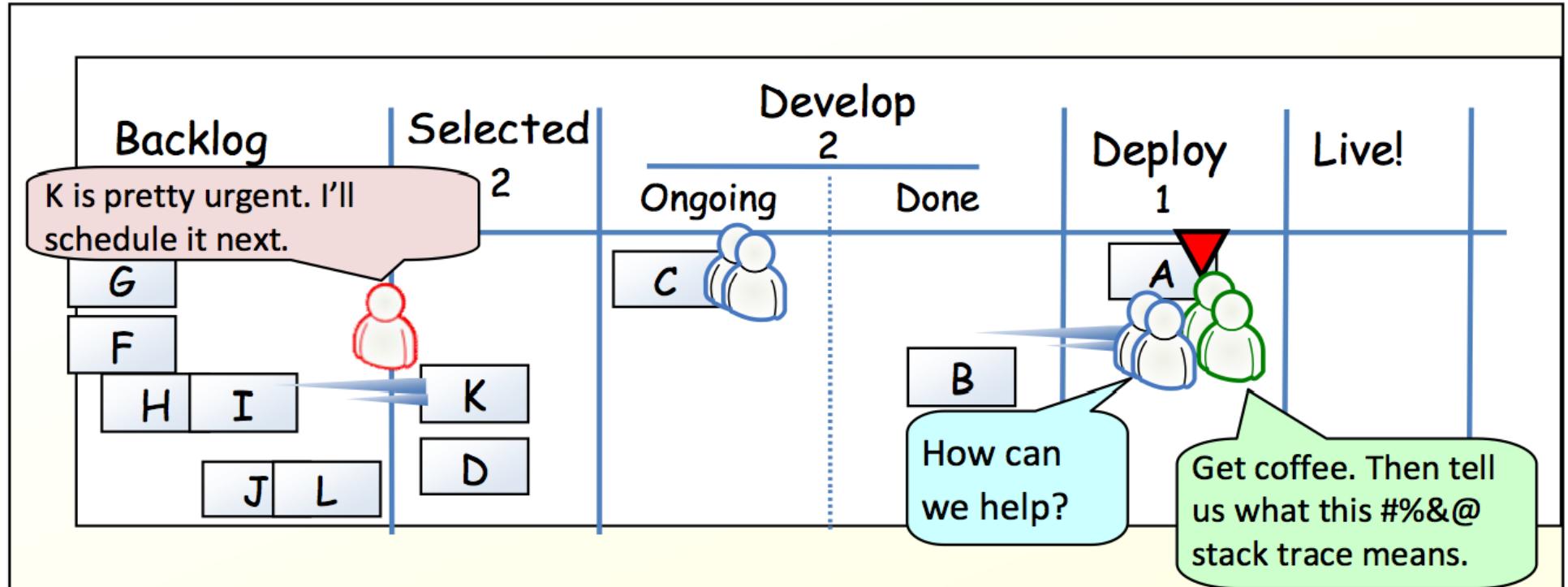
- Voikin olla mielekästä rajoittaa yhtä aikaa tekemisen alla olevien töiden määrää asettamalla **work in progress (eli WIP) -rajoituksia**
  - WIP-rajoitukset on lainattu **Kanban**-menetelmästä
  - Scrumin ja Kanbanin yhdistelmää kutsutaan usein nimellä *Scrumban*
  - Scrumbanissa on tosin muitakin Kanbanista lainattuja elementtejä kuin WIP-rajoitukset
- Esimerkki kirjasta <http://www.infoq.com/minibooks/kanban-scrum-minibook>
- Kunkin vaiheen WIP-rajoitus on merkitty numerona, eli vaiheessa saa olla yhtä aikaa vain rajoituksen verran taskeja

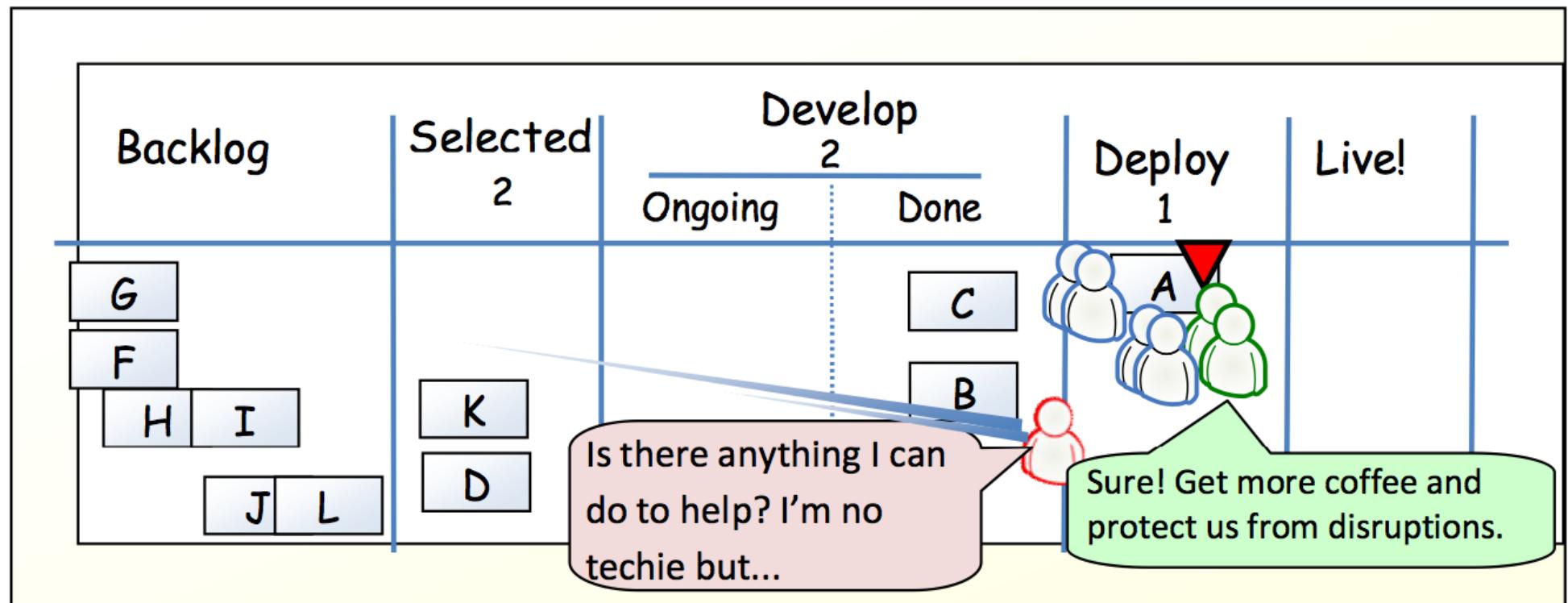
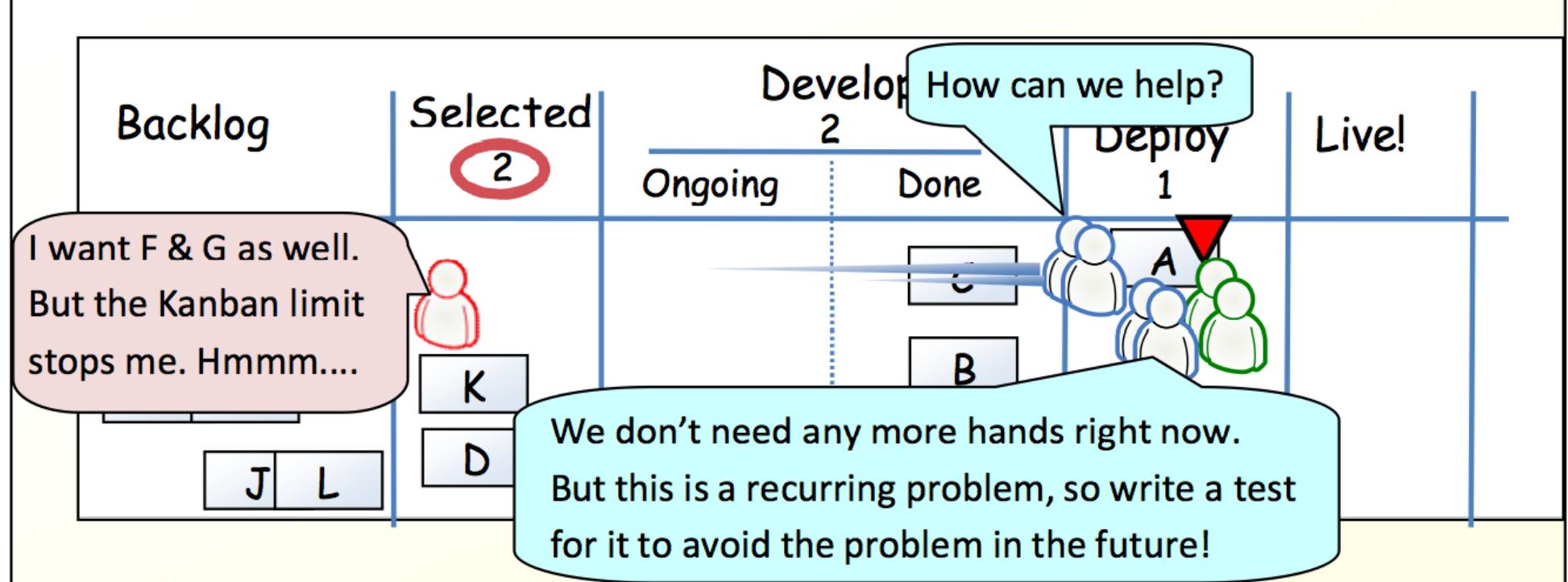












# Sprintin seuranta taulukkomuodossa

- Taskboardin sijaan sprintin seuranta hoidetaan usein taulukkolaskennan avulla, erityisesti jos tiimillä ei ole käytössä omaa "seinää"
- Tällöin sprintin jokaiselle päivälle on oma sarake, johon merkitään kunkin päivän alussa estimaatti taskien jäljellä olevasta työmäärästä (tunteina)

User Story	Tasks	Day 1	Day 2	Day 3	Day 4	Day 5	...
As a member, I can read profiles of other members so that I can find someone to date.	Code the ...	8	4	8	0		
	Design the ...	16	12	10	4		
	Meet with Mary about ...	8	16	16	11		
	Design the UI	12	6	0	0		
	Automate tests ...	4	4	1	0		
	Code the other ...	8	8	8	8		
As a member, I can update my billing information.	Update security tests	6	6	4	0		
	Design a solution to ...	12	6	0	0		
	Write test plan	8	8	4	0		
	Automate tests ...	12	12	10	6		
	Code the ...	8	8	8	4		

- Erään ohtuprojektiin product- ja sprintbacklogit:
  - <https://docs.google.com/spreadsheets/d/13RzIZI2NFFuV0zdRjrrfoC-CrottK8AZNuHS571WIxo/edit?usp=sharing>

# Sprintin etenemisen seuranta

- Taskboard ja burndown-käyrä tuovat selkeästi esille sprintin etenemisen asteen ja onkin suositeltavaa, että ne ovat kaikkien tiimiläisten ja projektin sidosryhmäläisten nähtävillä
- Ketterät menetelmät korostavat läpinäkyvyyttä (transparency) ja tiedon maksimaalista kommunikoitumista, näin mahdolliset ongelmatkaan eivät tule yllätyksenä ja niihin on helpompi puuttua ajoissa
- Lisää aiheesta:
  - <http://xprogramming.com/articles/bigvisiblecharts/>
  - <http://blog.mountaingoatsoftware.com/the-ideal-agile-workspace>
- Usein toki käytetään myös elektronisia vastineita taskboardista, erityisesti jos kyseessä on hajautettu tiimi, esim:
  - Edellisen sivun tyyliin google docs tai excel
  - Asana, Trello, Github project, Pivotal Tracker, JIRA, trac, bugzilla, ...
- Yleinen konsensus on kuitenkin, että ainakin Sprintin hallintaan manuaalinen postit-lappuja hyödyntävä taskboard on käytettävyydeltään ylivoimainen

# Sprint review ja retrospektiivi

- Kuten luennoilla 2 mainittiin pidetään sprintin lopussa sprint review eli katselointi ja sprintin retrospektiivi
- Katselmoinnissa arvioidaan kehitystiimin tekemää työtä
  - Kesken jäneet tai epäkelvostit toteutetut User storyt siirretään takaisin backlogiin
- Retrospektiivissä taas tiimi itse tarkastelee omaa toimintatapaansa ja identifioi mahdollisia kehityskohteita seuraavaan sprinttiin
- Sprintin aikana on product backlogiin tullut ehkä uusia User storyja tai jo olemassa olevia storyjä on muutettu ja uudelleenpriorisoitu
- On suositeltavaa että kehitystiimi käyttää pienen määrän aikaa sprintin aikana product backlogin vaatimiin toimiin eli (backlog groomingiin), esim. uusien User storyjen estimointiin
- Jos product backlog on hyvässä kunnossa (DEEP) sprintin loppuessa, on jälleen helppo lähteä sprintin suunnitteluun ja uuteen sprinttiin

Ohjelmistotuotanto

Luento 5

27.3.

# Verifointi ja Validointi

# Verifointi ja Validointi

- Kehitettävän ohjelmiston elinkaareen oleellisena osana kuuluu
  - Verifointi "*are we building the product right*" ja
  - Validointi "*are we building the right product*"
- Verifioinnissa siis pyritään varmistamaan, että ohjelmisto toteuttaa vaatimusmäärittelyn aikana sille asetetut vaatimukset
  - Yleensä tämä tapahtuu testaamalla, että ohjelma toteuttaa sille asetetut (ja vaatimusmäärittelyyn kirjatut) toiminnalliset ja ei-toiminnalliset vaatimukset
- Validointi pyrkii varmistamaan, että ohjelmisto täyttää käyttäjän odotukset
  - Vaatimusmäärittelyn aikana kirjatut ohjelmiston vaatimukset eivät ole aina se mitä käyttäjä ohjelmanlaista todella haluaa!
- Verifioinnin ja validoinnin tavoitteena on varmistaa että ohjelma on "riittävän hyvä" siihen käyttötarkoitukseen, mihin ohjelma on tarkoitettu
  - Hyvyys on suhteellista ja riippuu ohjelman käyttötarkoituksesta
  - Ohjelman ei esim. tarvitse yleensä olla virheetön ollakseen kuitenkin riittävän hyvä käytettäväksi

# Verifioinnin ja Validoinnin tekniikat

- Perinteisesti verifioinnissa on käytetty kahta tekniikkaa
  - Katselmointeja ja tarkastuksia
  - Testausta
- **Katselmoinneissa** (review) käydään läpi erilaisia ohjelmiston tuotantoprosessin aikana tuotettuja dokumentteja ja ohjelmakoodia, ja etsitään näistä erilaisia ongelmia
- **Tarkastukset** (inspection) ovat katselmointien muodollisempi versio
  - Järjestetään formaali kokous, jolla tarkkaan määritelty agenda ja kokouksen osallistujilla ennalta määritellyt roolit
- Katselointi on *staattinen tekniikka*, suorituskelpoista ohjelmakoodia ei tarvita ja jos katselmoinnin kohteena on ohjelmakoodi, ei ohjelmaa katselmuksissa suoriteta
- **Testaus** on *dynaaminen tekniikka*, joka edellyttää aina ohjelmakoodin suorittamista
  - Testauksessa tarkkaillaan miten ohjelma reagoi annettuihin testisyötteisiin

# Vaatimusten validointi

- Ohjelmistolle määritellyt vaatimukset on validoitava, eli varmistettava, että *määrittelydokumentti määrittelee sellaisen ohjelmiston, jonka asiakas haluaa*
- Vesiputousmallissa määrittelydokumentin kirjattujen vaatimusten validointi suoritetaan nimenomaan katselmoimalla
  - Vaatimusmäärittely päättyy siihen, että asiakas tarkastaa määrittelydokumenttiin kirjattujen vaatimuksien vastaavan asiakkaan kuvaan tilattavasta järjestelmästä
  - Katselmoinnin jälkeen määrittelydokumentti *jäädytetään* ja sen muuttaminen vaatii yleensä monimutkaista prosessia
- Ketterässä prosessissa vaatimusten validointi tapahtuu iteraation päättävien demonstraatioiden (Scrumissa sprint review) yhteydessä
  - Asiakkaalle näytetään ohjelman toimivaa versiota
  - Asiakas voi itse verrata vastaako lopputulos sitä mitä asiakas haluaa
    - Asiakkaan haluama toiminnallisuushan voi poiketa määritellystä toiminnallisuudesta!
  - Jos ei, on seuraavassa iteraatiossa mahdollista ottaa korjausliike
- On ilmeistä, että ketterän mallin käyttämä vaatimusten validointitapa toimii paremmin tuotekehitystyyppisissä tilanteissa, joissa ollaan tekemässä tuotetta, joka on vaikea määritellä tarkkaan etukäteen

# Koodin katselointi

- Koodin katselointi eli koodin lukeminen jonkun muun kuin ohjelmoijan toimesta on havaittu erittäin tehokkaaksi keinoksi koodin laadun parantamisessa
- Katselmoinnin avulla voidaan havaita koodista ongelmia, joita testauksella ei vältämättä havaita, esim.
  - noudattaako koodi sovittua tyylilä
  - onko koodi ylläpidettävä
- Koodin katselmoinnissa on perinteisesti käyty läpi onko koodissa tiettyjä "checklisteissä" listattuja riskialttiita piirteitä, ks. esim.
  - [http://www.oualline.com/talks/ins/inspection/c\\_check.html](http://www.oualline.com/talks/ins/inspection/c_check.html)
  - Joissakin kielissä, esim. Javassa käänäjän tuki tekee osan näistä tarkistuksista turhaksi
- Nykyään on tarjolla paljon katselointia automatisoivia *staattista analyysiä* tekeviä työkaluja esim. Javalla PMD ja Checkstyle
  - <http://pmd.sourceforge.net/>
  - <http://checkstyle.sourceforge.net/>
- Tutustumme checkstyleen laskareissa

# Koodin katselointi: GitHub ja pull requestit

- Yhä enenevä määrä ohjelmistotuotantoprojekteja tallettaa lähdekoodinsa GitHubiin
- GitHubin *pull requestit* tarjoavat hyvän työkalun koodikatselointien tekoon
- Pull requesteja käytettäessä työn kulku on seuraava
  - Sovelluskehittäjä *forkkaa* repositorin itselleen, tekee muutokset omaan repositorioon ja tekee *pull requestin* projektia hallinnoivalle taholle
  - Hallinnoija, esim. tiimin ”senior developer” tai opensource-projektin vastaava tekee katselmoinnin pull requestille
  - Jos koodi ei ole vielä siinä kunnossa että tehdyt muutokset voidaan *mergetä* repositorioon, kirjoittaa hallinnoija pull requestin tekijälle joukon parannusehdotuksia
  - Muutosten ollessa hyväksyttävässä kunnossa, pull request *mergetään* päärepositorioon
- Seuraavalla sivulla esimerkki TMC-projektiin tehdystä pull requestistä ja siihen liittyvistä kommenteista

# Koodin katselointi: GitHub ja pull requestit

 [testmycode / tmc-server](#)  Unwatch ▾ 8

---

## Course participants #201

 **Open** **kennyhei** wants to merge 9 commits into `testmycode:master` from `rage:course-participants`

 Conversation 24  Commits 9  Files changed 13

---



**kennyhei** commented on Oct 27, 2014 

Implementing [#185](#)



**kennyhei** added some commits on Oct 21, 2014

-  Course JSON with participants 9287e10
-  Course knows its students through submissions and vice versa e3e7c03
-  Prettier JSON b1b5dd7

```
@@ -31,6 +32,17 @@ def course_data(course)
 31   32     })
 32   33   end
 33   34
 35 + # Course JSON with participants
 36 + def course_participants_data(course)
 37 +   participants = course.users
 38 +
 39 +   data = {
 40 +     :id => course.id,
 41 +     :name => course.name,
 42 +     :participants => participants.map { |participant| participant_data(participant)}
```



mpartel added a note on Oct 29, 2014

Owner



On my desktop, with the mooc production DB dump, this takes around 30 seconds for the k2014-mooc course. I'd really like to avoid adding more really slow queries to TMC.

Would the following make sense?

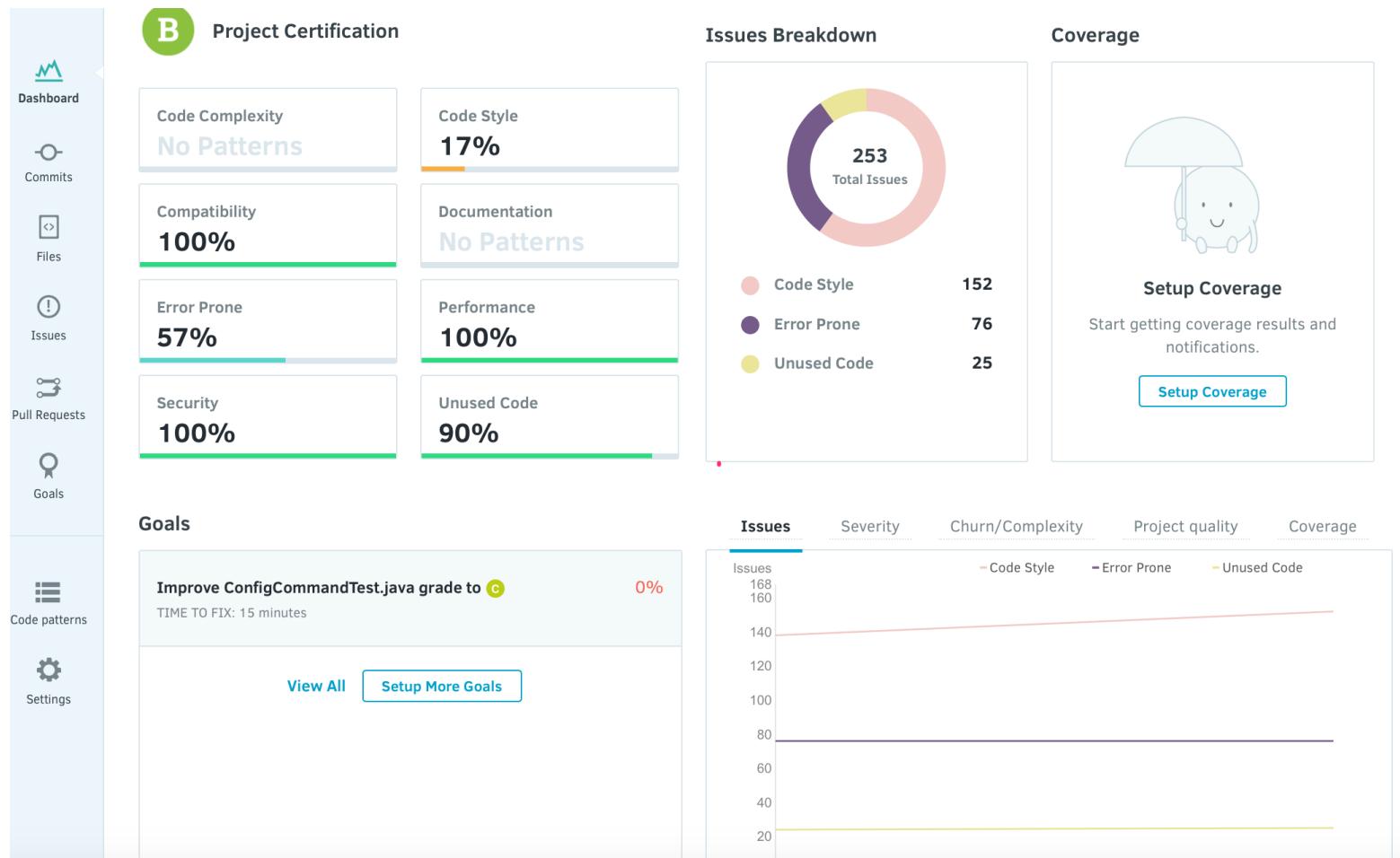
- Let this only return a list of participants and their newest submission IDs.
- Load a user's exercise statuses on demand, and cache them either on your side or maybe in TMC until the submission ID changes.
- Consider having the per-user URL support [ETags](#).

# Automatisoitu staattinen analyysi pilvessä

- Pilvipalvelut ovat helpottaneet sovelluskehittäjien työtä monissa asioissa, esim. GitHubin ansiosta omaa versionhallintapalvelinta ei ole enää tarvetta ylläpitää
- Pilveen on viime aikoina ilmestynyt myös koodille staattista analyysiä tekeviä palveluita, esim. <https://codeclimate.com/>
- CodeClimate analysoi koodista mm. seuraavia asioita:
  - Liian kompleksiset metodit ja luokat
  - Copy paste -koodi
  - Testaamaton koodi
- CodeClimate myös huomauttaa koodin laadun muutoksista, esim. jos koodin kompleksisuus kasvaa muutosten yhteydessä ja antaa parannusehdotuksia liian monimutkaisiin metodeihin
- Minkä tahansa GitHubissa olevan Ruby, Javascript tai PHP-projektiin saa konfiguroitua Codeclimatena tarkastettavaksi nappia painamalla
- Codeclimate suorittaa tarkastukset koodille aina kun uutta koodia pushataan GitHubiin
- Labtoolin eli laitoksen harjoitustöiden kirjanpito-ohjelmiston CodeClimate-raportti löytyy osoitteesta <https://codeclimate.com/github/mluukkai/labtool>

# Automatisoitu staattinen analyysi pilvessä

- Hiljattain ilmestynyt pilvipalvelu <https://www.codacy.com> osaa tehdä staattista analyysiä myös Javalla tehdyille ohjelmille
- Codacy analysoi osin samoja asioita kuin Codelimate, mutta erojakin löytyy. Codacy osaa identifioida koodista mm. tietoturvaan liittyviä ongelmia
- Ote TMC-komentorivclientin raporista



# Koodin katselointi ketterissä menetelmissä

- Toisin kuin Scrum, **eXtreme Programming** eli **XP** määrittelee useita käytänteitä, joita pyritään noudattamaan ohjelmistoa tehtäessä
  - Suuri osa XP:n käytänteistä on hyvin tunnettuja "best practiseja", mutta kuitenkin usein vietyvä äärimmäiseen (extreme) muotoon
  - Osa käytänteistä tähtää ohjelmiston laadun maksimoimiseen ja kolmen voidaan ajatella olevan katselmoinnin äärimmilleen vietyjä muotoja
  - **Pariohjelmoinnissa** (pair programming) kaksi ohjelmoijaa työskentelee yhdessä yhdellä koneella
    - Koodia kirjoittava osapuoli toimii *ohjaajana* (driver) ja toinen *navigoijana* (navigator), roolia vaihdetaan sopivin väliajoin
    - Navigoija tekee koodiin **jatkuvaa katselointia**
    - Etuja:
      - Parantaa ohjelmoijien kuria ja työhön keskittymistä
      - Hyvä oppimisen väline: ohjelmoijat oppivat toisiltaan erityisesti noviisit kokeneimmilta, järjestelmän tietyn osan tuntee aina useampi ohjelmoija
    - Tutkimuksissa todettu vähentävän bugien määrää 15-50%, kokonaisresurssin kulutus kuitenkin nousee hieman

# Koodin katselointi ketterissä menetelmissä

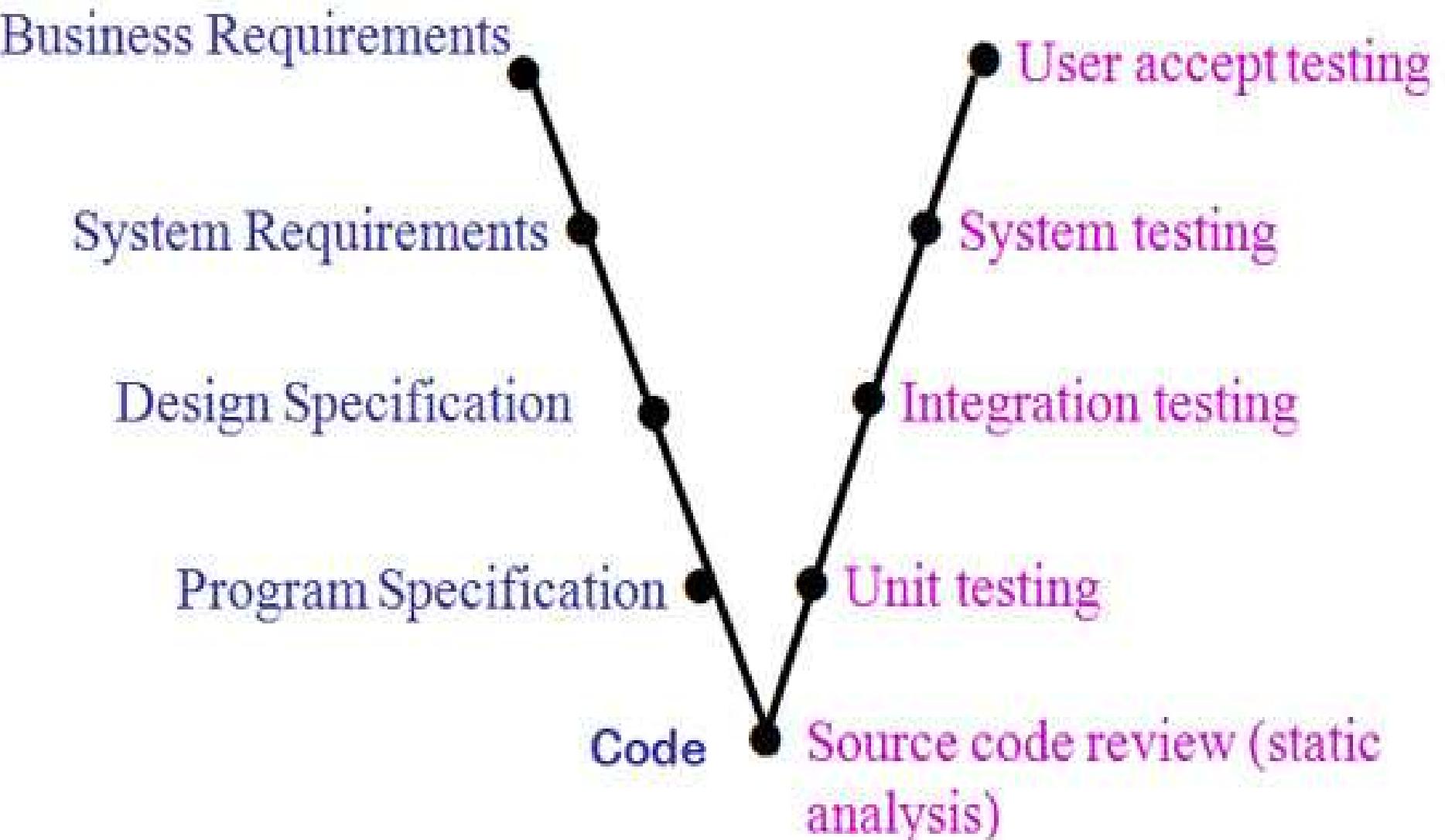
- Lisää pariohjelmoinnista esim. seuraavassa:
  - [http://jamesshore.com/Agile-Book/pair\\_programming.html](http://jamesshore.com/Agile-Book/pair_programming.html)
- Pariohjelmoinnin ohella ”katselointimaisia” tavoitteita koodin laadun nostajana voidaan ajatella olevan XP:n yhteisomistajuuden ja ohjelointistandardien periaatteilla
- Koodin **yhteisomistajuus** (collective code ownership) tarkoittaa periaatetta, jossa kukaan yksittäinen ohjelmoija ei hallitse yksin mitään kohtaa koodista
  - Kaikilla on lupa tehdä muutoksia mihin tahansa kohtaan koodia
  - Pariohjelointi tukee yhteisomistajuutta
  - Yhteisomistajuudessa on omat riskinsä: joku koodia kunnolla tuntematon voi saada pahaa jälkeä aikaan. XP eliminoi tämän testauksiin liittyvillä käytänteillä
  - [http://jamesshore.com/Agile-Book/collective\\_code\\_ownership.html](http://jamesshore.com/Agile-Book/collective_code_ownership.html)
- Ohjelointistandardi **coding standards** tarkoittaa, että tiimi määrittelee koodityylin, johon kaikki ohjelmoijat sitoutuvat
  - Tyylillä tarkoitetaan nimeämiskäytäntöä, koodin muotoilua ja myös tiettyjä ohjelman rakenteeseen liittyviä seikkoja
  - [http://jamesshore.com/Agile-Book/coding\\_standards.html](http://jamesshore.com/Agile-Book/coding_standards.html)

# Testaus

- Ohjelmien osoittaminen virheettömäksi on käytännössä mahdotonta
- Testauksen tarkoituksena onkin **vakuuttaa asiakas ja järjestelmän kehitystiimi siitä, että ohjelmisto on tarpeeksi hyvä käytettäväksi**
- Testauksella on kaksi hieman eriävää tavoitetta
  - osoittaa, että ohjelmisto täyttää sille asetetut vaatimukset
    - käytännössä tämä tarkoittaa vaatimusdokumenttiin/vaatimuksiin kirjattujen asioiden toteutumisen demonstroimista
  - löytää ohjelmistosta virheitä
    - yritetään rikkoa tai saattaa ohjelma jollain tavalla epäkonsistenttiin tilaan
- Molemmat näistä tavoitteista tähtäävät oikeastaan ohjelman **ulkosien laadun** (external quality) parantamiseen
  - **External Quality is the fitness for purpose of the software.** It's most obvious measure is the Functional Tests, and some measure of the bugs that are still loose when the product is released.
  - <http://c2.com/cgi/wiki?InternalAndExternalQuality>

# Testauksen tasot

- Testausta jakaantuu eri *tasoihin* sen mukaan mukaan, mikä testauksen ensisijaisena kohteena on
- Vesiputousmallia laajentava *testauksen V-malli* havainnollistaa testauksen eri tasoja



# Testauksen tasot

- **Yksikkötestaus (unit testing)**
  - Yksittäisten luokkien, metodien ja moduulien testaus erillään muusta kokonaisuudesta
  - Yleensä ohjelmoijat suorittavat
- **Integraatiotestaus (integration testing)**
  - Yksittäin testattujen komponenttien liittäminen yhteen eli integrointi ja kokonaisuuden testaus
  - Integroinnin tekevät sovelluskehittäjät suorittavat yleensä myös testauksen
- **Järjestelmätestaus (system testing)**
  - Varmistetaan että järjestelmä toimii vaatimuksiin kirjatulla tavalla
  - Testataan järjestelmää saman rajapinnan kautta, jonka kautta järjestelmää käytetään
  - Kehittäjäorganisaatio suorittaa
  - Jakautuu useisiin alalajeihin joista kohta lisää
- **Käyttäjän hyväksymistestaus (user acceptance testing)**
  - Loppukäyttäjän tuotteelle suorittama testaus

# Järjestelmätestaus

- Tarkoitus siis varmistaa, että järjestelmä toimii vaatimuksiin kirjatulla tavalla
- Testataan järjestelmää saman rajapinnan kautta, jonka kautta järjestelmää käytetään
- Testaus tapahtuu ilman tietoa järjestelmän sisäisestä rakenteesta eli kyseessä **black box -testaus**
- Yleensä järjestelmätestaus perustuu järjestelmän potentiaaliin käyttöskenaarioihin
  - jos vaatimukset on ilmaistu User storyina, on niistä melko helppo muotoilla testejä, joiden avulla voidaan varmistaa että järjestelmällä on Storyjen kuvaamat vaatimukset sekä tyypilliset virheskenaariot
- "perusmuotonsa" eli vaatimuksiin kirjattujen toiminnallisten vaatimuksien testaamisen lisäksi järjestelmätestaukseen kuuluu mm:
  - Käytettävyystestaus
  - Suorituskykytestaus tai stressitestaus
  - Tietoturvan testaus
  - lisää [http://en.wikipedia.org/wiki/System\\_testing](http://en.wikipedia.org/wiki/System_testing)

# Testitapausten valinta

- Kattava testaaminen on mahdotonta ja testaus joka tapauksessa työlästä
- Onkin tärkeää löytää kohtuullisen kokoinen testitapausten joukko, jonka avulla on kuitenkin mahdollista löytää mahdollisimman suuri määrä virheitä
- Testitapaus testaa järjestelmän toiminnallisuutta yleensä joillakin **syötteillä**
- Useat syötteet ovat järjestelmän toiminnan kannalta samanlaisia
- Testeissä kannattaakin pyrkiä jakamaan syötteet **ekvivalenssiluokkiin** ja tehdä yksi testitapaus kutakin ekvivalenssiluokkaa tai syötteiden ekvivalenssiluokkien kombinaatiota kohti
  - Testin kannalta samalla tavalla käytätytvät syötteet siis muodostavat oman ekvivalenssiluokkansa
  - Ekvivalenssiluokkien edustajien lisäksi kannattaa tehdä myös testitapaukset ekvivalenssiluokkien **raja-arvoille**

# Testisyötteiden valinta

- Mitä testitapauksia kannattaisi valita seuraavalla sivulla olevalle **tekstitv:n sivun valintaikkunaan?**
  - Tekstitv:n *sivu* vastaa lukua väliltä 100-899
  - Osaa välin luvuista vastaavaa sivua ei ole olemassa
- Testisyötteen ekvivalenssiluokkia olisivat ainakin seuraavat
  - Olemassaolevaa sivua vastaavat luvut
  - Validit luvut jotka eivät vastaa mitään sivua
  - Liian pienet ja liian suuret luvut
  - Syötteet jotka sisältävät kiellettyjä merkkejä
  - Tyhjä syöte
- Jokaisesta ekvivalenssiluokasta olisi siis hyvä valita ainakin yksi testattava syötearvo
- Olemassaolevaa sivua vastaavan ekvivalenssin rajatapaukset, eli luvut 100 ja 899 kannattaisi ehkä valita testisyötteiksi
- Samoin luvut 99 ja 900 jotka ovat oman ekvivalenssiluokkansa rajatapauksia

# Tekstityv:n testitapaukset

[Edellinen sivu](#) | [Edellinen alasivu](#) | [Seuraava alasivu](#) | [Seuraava sivu](#)

Teksti-TV

[yle.fi/tekstitv](http://yle.fi/tekstitv) 199 PÄÄHAKEMISTO

[106](#) Taksisääntelyn purkaminen etenee

[107](#) Tieto irtisanoo lähes [180](#)

[162](#) Reuters: Kreikan kasvu yllätti

[651](#) MM-karsinta: Turkki nujersi Suomen

[221](#) Janne Keränen jälleen KalPa-sankari

[101](#) UUTISET [160](#) TALOUS [190](#) ENGLISH

[201](#) URHEILU [350](#) RADIOT [470](#) VEIKKAUS

[300](#) OHJELMAT [400](#) SÄÄ [575](#) TEKSTI-TV

[799](#) SVENSKA [500](#) ALUEET [890](#) KALENTERI

Sää paikkakunnittain [406–408](#)

Nopea piiras [811](#)

[Edellinen sivu](#) | [Edellinen alasivu](#) | [Seuraava alasivu](#) | [Seuraava sivu](#)

[Kotimaa](#) | [Ulkomaat](#) | [Talous](#) | [Urheilu](#) | [Svenska sidor](#) | [Teksti-TV](#) [Yle.fi](#) [YLE Uutiset](#)

Sivun valinta:

# Testisyötteiden valinta

- Seuraavalla sivulla on tuttu kaavake, minkälaisia testitapauksia kannattaisi valita, jos oletetaan, että järjestelmä olisi määritelty seuraavasti
  - Oikein täytetty kaavake hyväksytään ja kaavakkeen tiedot talletetaan järjestelmään
  - Väärän syötearvon omaava kaavake hylätään
    - Käyttäjä palautetaan lomakenäkymään ja väärin syötetyille kentille annetaan virheilmoitus
- Kenttien syötemuodot on määritelty seuraavasti
  - Opiskelijanumero koostuu yhdeksästä numerosta ja alkaa numerolla 0
  - Etu- ja sukunimi ovat epätyhjiä kirjaimista koostuvia merkkijonoja
  - Email-osoite on epätyhjä merkkijono, joka on standardin määrittelemän syntaksin mukainen
  - GitHub-tunnus on epätyhjä alfanumeerisista merkeistä koostuva merkkijono, joka on muotoa  
<https://github.com/kayttajatunnus/repositorio>
  - Käytetyt tunnit on kokonaisluku

# Viikkopalautelomakkeen testitapaukset

A screenshot of a web browser window titled "Ohtustats". The URL in the address bar is "ohtustats.herokuapp.com/submissions/new". The page content is a form titled "create a submission for week 2". The form fields are:

- Student number:** An input field.
- First name:** An input field.
- Last name:** An input field.
- Email:** An input field.
- Github repository:** An input field.
- Completed exercises:** A list of numbered checkboxes from 1 to 11, followed by a "mark all" button.
  - 1
  - 2
  - 3
  - 4
  - 5
  - 6
  - 7
  - 8
  - 9
  - 10
  - 11
  - mark all**
- Hours:** An input field.
- Comments:** A large input field.

# Viikkopalautelomakkeen testitapaukset

- Esim. kentän opiskelijanumero ekvivalenssiluokat voisivat olla
  - Kelvollinen syöte
  - 9 numeroa sisältävä, muulla kuin 0:llä alkava numerosarja
  - Vähemmän kuin 9 numeroa sisältävä numerosarja
  - Yli 9 numeroa sisältävä numerosarja
  - Yli 9 numeroa pitkä, sallitun opiskelijanumeron sisältävä numerosarja
  - Jonkin muun merkin kuin numeron sisältävä syöte
- Kaikkien syötekentien ekvivalenssiluokkien kombinaatio olisi jo tässä yksinkertaisessa tapauksessa todella suuri
- Järkevä testistrategia lienee valita testitapaukset seuraavasti
  - Testataan jokaista kenttää siten, että kaikissa muissa kentissä on joku validi syöte ja kokeillaan yksittäiselle kentälle kaikkia sen syötteen ekvivalenssiluokkia
  - Tämä testistrategia olettaa, että erillisten kenttien validointi on toisistaan riippumatonta, ja että esim. kahden kentän yhtäaikainen virheellinen syöte ei saa aikaan mitään kummallista (esim. kaada järjestelmää)

# Yksikkötestaus

- Kohteena siis yksittäiset metodit ja luokat
- Ohjelmoijat suorittavat yksikkötestauksen
- Testattavan koodin rakenne otetaan huomioon testejä laatiessa, eli kyseessä **lasilaatikkotestaus** (white box testing)
- Yksikkötestauksella ei testata suoranaiseksi sitä täyttääkö ohjelmisto vaatimuksensa, pikemminkin tavoitteena on ohjelman **sisäisen laadun** (internal quality) kontrollointi
  - Internal quality is about the design of the software
  - This is purely the interest of development
  - If Internal quality starts falling the system will be less amenable to change in the future
  - Hence the need for refactoring, clear coding, relentless testing, and the like
  - You need to be very careful about letting internal quality slip
  - <http://c2.com/cgi/wiki?InternalAndExternalQuality>

# Yksikkötestaus

- Ohjelman sisäinen laatu siis vaikuttaa erityisesti siihen, miten ohjelmaa voidaan laajentaa ja jatkokehittää
- Ketterissä menetelmissä sisäisellä laadulla onkin todella suuri merkitys, tähän palataan tarkemmin huomenna
- Pelkän sisäisen laadun kontrollimekanismi yksikkötestaus ei toki ole
- Kattavilla yksikkötesteillä saadaan parannettua myös ohjelman ulkoista, eli asiakkaan näkemää laatua
  - Yksikkötestit voivat eliminoida joitain asiakkaalle näkyviä virheitä, joita järjestelmätestauksen testitapaukset eivät löydä
- Bugit on taloudellisesti edullista paikallistaa mahdollisimman aikaisessa vaiheessa, eli yksikkötestauksessa löydetty virhe on halvempi ja nopeampi korjata kuin järjestelmä- tai integraatiotestauksessa löytyvä virhe
- Koska yksikkötestejä joudutaan ajamaan moneen kertaan, tulee niiden suorittaminen ja testien tulosten raportointi automatisoida, ja nykyinen hyvä työkalutuki tekeekin automatisoinnin helpoksi
  - xUnit eniten käytetty, uudempia tulokkaita mm. TestNG ja RSpec

# Mitä ja miten paljon tulee testata?

- Mitä tulisi testata yksikkötestein? JUnitin kehittäjän Kent Beckin vastaus:  
*"Do I have to write a test for everything?"*  
"No, just test everything that could reasonably break"  
[<http://junit.sourceforge.net/doc/faq/faq.htm>]
- Vastaus ei siis ole helppo. Ainakin tulisi olla testitapaukset
  - kaikkien metodien (ja loogisten metodikombinaatioiden) toiminta parametrien hyväksyttävillä arvoilla
  - ja virheellisillä parametrien arvoilla
- Parametrien mahdolliset arvot kannattaa jakaa ekvivalenssiluokkiin (ks. kalvo 17) ja jokaisesta luokasta valita yksi arvo testiä varten, myös ekvivalenssiluokkien raja-arvot kannattaa valita mukaan
- Koska yksikkötestejä tehtäessä ohjelmakoodi on nähtävillä, on testattavien arvojen parametrien ekvivalenssiluokat ja raja-arvot päättelyissä koodista
- Esim. Varaston metodi *otaVarastosta*, mitä testitapauksia tulisi generoida jotta kaikki edelläolevat ohjeet täytyvät?
  - [http://www.cs.helsinki.fi/u/wikla/ohjelmointi/materiaali/02\\_oliot](http://www.cs.helsinki.fi/u/wikla/ohjelmointi/materiaali/02_oliot)

```

public class Varasto {
    private double tilavuus;
    private double saldo;

    public double otaVarastosta(double maara) {
        if (maara < 0) return 0.0;

        if(maara > saldo) {
            double kaikkiMitaVoidaan = saldo;
            saldo = 0.0;
            return kaikkiMitaVoidaan;
        }

        saldo = saldo - maara;
        return maara;
    }
}

```

- Metodia *otaVarastosta* testatessa testitapauksessa on huomioitava parametrin maara lisäksi varaston tilanne
- Varastotilanteita on kolmea "ekvivalenssiluokkaa"
  - Tyhjä (esim. saldo 0, tilavuus 10)
  - Ei tyhjä eikä täysi (saldo 5, tilavuus 10)
  - Täysi (saldo 10, tilavuus 10)
- Näitä kutakin kohti on metodin parametrilla *maara* omat ekvivalenssiluokkansa
  - Esim. täysi varasto: maara = -1, 0, 5, 10, 11
- *Tarvitseeko esim. nollan tai negatiivisen määrän ottamista tarkastaa kaikkien varastotilanteiden yhteydessä?*

# Testauskattavuus

- Yksikkötestien (ja toki myös muunkinlaisten testien) hyvyyttä voidaan mitata **testauskattavuuden** (test coverage) käsitteellä
- Testauskattavuutta on muutamaa eri tyyppiä
- **Rivikattavuudella** (line coverage) tarkoitetaan kuinka montaa prosenttia testattavan metodin/luokan koodirivejä testimetodit suorittavat
  - Vaikka rivikattavuus olisi 100% ei tämä tietenkään tarkoita, että kaikki oleellinen toiminnallisuus olisi tutkittu
- **Haarautumakattavuudella** (branch coverage) tarkoitetaan kuinka montaa prosenttia testattavan metodin/luokan sisältävistä ehtolauseiden haaroista testit ovat suorittaneet
- Monet työkalut, esim. käyttämämme JaCoCo mittaavat testien suorituksen yhteydessä testauskattavuuden
- Muitakin kattavuuden tyyppejä on olemassa, mm. *ehtokattavuus* ja *polkukattavuus*, useat työkalut eivät niitä kuitenkaan testaa
- **Hyvätkö testit siis saavuttavat mahdollisimman suuren kattavuuden ja ottavat huomioon edellisen sivun ohjeistuksen**

# Testauskattavuus JaCoCossa

- JaCoCo ilmoittaa sekä rivi- (instruction) että haaraumakattavuuden (branches)

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
Varasto(double, double)		0%		0%	4	4	10	10	1	1
toString()		0%		n/a	1	1	1	1	1	1
otaVarastosta(double)		62%		50%	2	3	4	8	0	1
lisaaVarastoon(double)		77%		50%	2	3	2	6	0	1
Varasto(double)		82%		50%	1	2	1	6	0	1
paljonkoMahtuu()		100%		n/a	0	1	0	1	0	1
getSaldo()		100%		n/a	0	1	0	1	0	1
getTilavuus()		100%		n/a	0	1	0	1	0	1
Total	66 of 126	47%	11 of 16	31%	10	16	18	34	2	8

- Epäkattavasti testattu haarautumiskohda esim. if ilmaistaan keltaisella

```
51.     public void lisaaVarastoon(double maara) {
52.         if (maara < 0) // virhetilanteessa voidaan tehdä
53.         {
54.             return; // tällainen pikapoistuminenkin!
55.         }
56.         if (maara <= paljonkoMahtuu()) // omia aksessoreita voi kutsua
57.         {
58.             saldo = 1 of 2 branches missed. +; // ihan suoraan sellaisinaan
59.         } else {
60.             saldo = tilavuus; // täyteen ja ylimäärä hukkaan!
61.         }
62.     }
```

# Mutaatiotestaus

- Pelkkä testikattavuus ei vielä kerro paljoakaan testien hyvyydestä, Hyvien testien tulisi olla sellaisia, että jos ohjelmaan tulee bugi, huomaavat testit virheen
- **Mutaatiotestauksen** (engl. mutation testing) idea on nimenomaan testata testitapausten hyvyyttä generoimalla koodiin systemaattisesti *mutantteja* eli pieniä "bugeja" ja katsoa havaitsevatko testit koodiin tulleet bugit
- Erilaisia mutanttityyppejä, joita mutaatiotestauksessa koodiin generoidaan on paljon erilaisia, mm.
  - Manipuloidaan ehtolausesta: if (  $x < 0$  ) → if (  $x \leq 0$  ) tai if ( true )
  - Vaihdetaan operaattoria:  $x += 1 \rightarrow x -= 1$
  - Kovakoodaan paluuarvo: return x; → return true;
  - Korvataan konstruktorikutsu: olio = new Olio() → olio = null;
- Mutaatiotestauksen ongelmana on mutaatioiden suuri määrä ja ns. *ekvivalentit mutantit*, joiden takia mutaatiotestauksen tulos vaatii aina ihmisen tulkintaa
  - Ekvivalentti mutanti tarkoittaa muutosta koodissa, joka ei kuitenkaan muuta ohjelman toiminnallisuutta. Eli mutantin lisäämistä koodiin ei voi mikään testi havaita. Mutantin toteaminen ekvivalentiksi algoritmisesti on mahdotonta
- Lisätietoa
  - [http://en.wikipedia.org/wiki/Mutation\\_testing](http://en.wikipedia.org/wiki/Mutation_testing) ja <http://pitest.org/>

# Integraatiotestaus

- Järjestelmän yksittäiset, erillään yksikkötestatut luokat tulee **integroida** toimivaksi kokonaisuudeksi
- Integroinnin yhteydessä tai sen jälkeen suoritetaan **integrointitestaus**
- Integraatiotestauksen painopiste on osien välisten rajapintojen toimivuuden tutkimisessa sekä komponenttien yhdessä tuottaman toiminnallisuuden oikeellisuuden varmistamisessa
- Järjestelmän integrointi voi edetä joko järjestelmän rakenteeseen perustuen tai järjestelmän toteuttamien omaisuksien mukaan
  - **Rakenteeseen perustuvassa** integraatiossa keskitytään kerrallaan sovelluksen yksittäisten rakenteellisten komponenttien integrointiin
    - Esim. olutkaupassa integroitaisiin sovelluslogiikan luokat, käyttöliittymän toteutus ja tietokantarajapinta omina kokonaisuuksinaan
  - **Omaisuksiin perustuvassa** integroinnissa, taas liitetään yhteen alikomponentit, jotka toteuttavat järjestelmän loogisen toimintakonkreettuuden
    - Olutkaupassa voitaisiin esim. integroida kerrallaan kaikki toiminnallisuuteen "lisää tuote ostoskorisiin" liittyvät luokat

# Integraatiotestaus

- Sekä rakenteeseen, että ominaisuuksiin perustuva integrointi voi tapahtua joko ylhäältä alas tai alhaalta ylös:
  - **Bottom up:** lähdetään liikkeelle yksittäisistä komponenteista, liitetään niitä yhteen ja suoritetaan testejä kunnes kaikki integroitavat komponentit on yhdistetty
  - **Top-down:** ensin kehitetään järjestelmän korkean tason rakenteet siten, että yksittäisten komponenttien paikalla on *tynkäkomponentteja* (stub). Tyngät korvataan sitten yksi kerrallaan todellisilla komponenteilla koko ajan kokonaisuutta testaten
- Oldschool-ohjelmistotuotannossa toimintatapa oli se, että kaikki ohjelman yksittäiset komponentit ohjelmoitiin ja yksikkötestattiin erikseen ja tämän jälkeen ne integroitiin (yleensä rakenteeseen perustuen) kerralla yhteen
  - *Tämän tyylinen big bang -integraatio* on osoittautunut todella riskialttiaksi (seurauksena usein ns. integraatiohelvetti) ja sitä ei enää kukaan täysijärkinen suosittele käytettäväksi
- Moderni ohjelmistotuotanto suosii ns. jatkuvaa integraatiota, joka on hyvin tiheässä tahdissa tapahtuva ominaisuuksiin perustuvaa integrointia
  - Palaamme aiheeseen huomenna

# Regressiotestaus

- Iteratiivisessa ja ketterässä ohjelmistotuotannossa, jossa jokainen iteraatio tuottaa ohjelmistoon uusia ominaisuuksia, on oltava tarkkana, että lisäykset eivät hajota ohjelman jo toimivia osia
- Testit siis on ajettava uudelleen aina kun ohjelmistoon tehdään muutoksia
- Tätä käytäntöä sanotaan **regressiotestaukseksi**
- Regressiotesteinä ei välttämättä tarvitse käyttää kaikkia ohjelmiston testejä, sopiva osajoukko voi taata riittävän luottamuksen
  - Regressiotestijoukko koostuu siis yksikkö-, integraatio- ja järjestelmätesteistä
- Testaus on erittäin työlästä ja regressiotestauksen tarve tekee siitä entistä työlämpää
- Tämän takia on erittäin tärkeää pyrkiä automatisoimaan testit mahdollisimman suurissa määrin
- Käsittelemme muutamia järjestelmätason testauksen automatisoinnin menetelmiä huomenna

Ohjelmistotuotanto

Luento 6

28.3.

Testaus ketterissä menetelmissä

Testauksen automatisointi

# Ketterien menetelmien testauskäytänteet

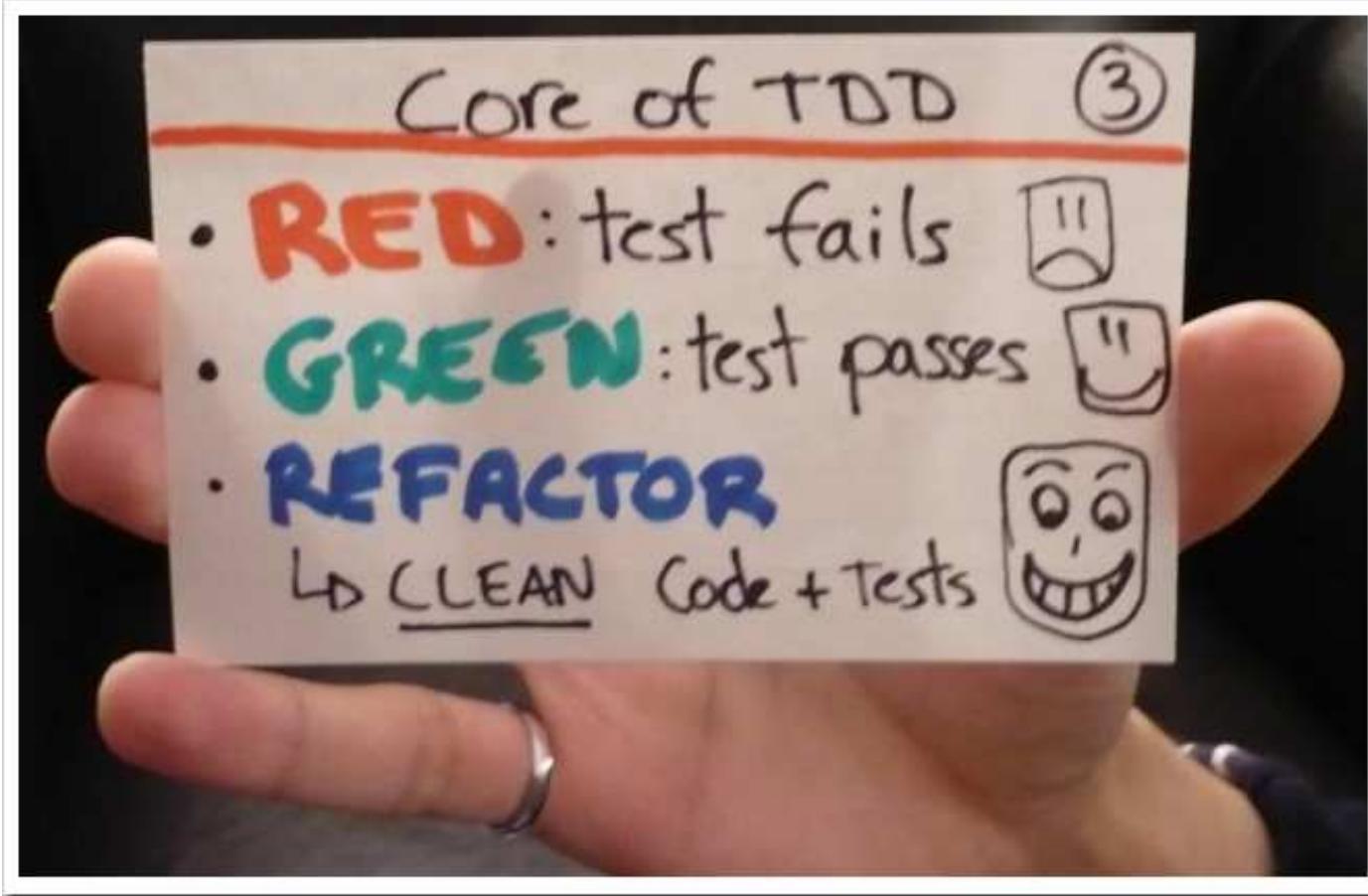
- Testauksen rooli ketterissä menetelmissä poikkeaa huomattavasti vesiputousmallisesta ohjelmistotuotannosta
- Iteraation/sprintin aikana toteutettavat ominaisuudet integroidaan muuhun koodiin ja testataan yksikkö-, integraatio- ja järjestelmätasolla
  - Sykli ominaisuuden määrittelystä siihen että se on valmis ja testattu on erittäin lyhyt, viikosta kuukauteen
- Testausta tehdään sprintin "ensimmäisestä päivästä" lähtien, testaus "integroitu" suunnittelun ja toteutukseen
- Testauksen automatisointi erittäin tärkeässä roolissa, sillä testejä ajetaan usein
  - Regressiotestaus tärkeää
- Ideaalilanteessa testaajia sijoitettu kehittäjätiimiin, ja myös ohjelmoijat kirjoittavat testejä
  - Testaajan rooli muuttuu virheiden etsijästä virheiden estäjään: testaaja auttaa tiimiä kirjoittamaan automatisoituja testejä, jotka pyrkivät estämään bugien pääsyn koodiin

# Ketterien menetelmien testauskäytänteitä

- Puhumme tänään neljästä ketterien menetelmien suosimasta testauskäytänteestä
- **Test driven development (TDD)**
  - Nimestään huolimatta kyseessä enemmänkin suunnittelu- ja toteutustason tekniikka
  - "sivutuotteena" syntyy kattava joukko automaattisesti ajettavia yksikkö- ja integraatiotestejä
- **Acceptance Test Driven Development / Behavior Driven Development**
  - Käyttäjätason vaatimusten tasolla tapahtuva "TDD"
- **Continuous Integration (CI)** suomeksi jatkuva integraatio
  - Perinteisen integraatio- ja integraatiotestausvaiheen korvaava työskentelytapa
  - Kaikista edellisistä käytänteistä seurausena suuri joukko eritasoisia (eli yksikkö-, integraatio-, järjestelmä-) automatisoituja testejä
- **Exploratory testing**, suomeksi tutkiva testaus
  - Järjestelmätestauksen tekniikka, jossa testaaminen tapahtuu ilman formaalia testisuunnitelmaa, testaaja luo lennossa uusia testejä edellisten testien antaman palautteen perusteella

# Test driven development

- TDD on yksi XP:n käytänteistä, Kent Beckin lanseeraama
- Joskus TDD:ksi kutsutaan tapaa, jossa testit kirjoitetaan ennen koodin kirjoittamista
  - Tätä tekniikkaa parempi kuitenkin kutsua nimellä *test first programming*
- "määritelmän mukainen" TDD etenee seuraavasti
  - 1) Kirjoitetaan sen verran testiä että testi ei mene läpi
    - Ei siis luoda heti kaikkia luokan testejä, edetään tekemällä ainoastaan yksi testi kerrallaan
  - 2) Kirjoitetaan koodia sen verran, että testi saadaan menemään läpi
    - Ei heti yritetäkään kirjoittaa "lopullista" koodia
  - 3) Jos huomataan koodin rakenteen menneen huonoksi (copypastea koodissa, liian pitkiä metodeja, ...) *refaktoroidaan* koodin rakenne paremmaksi
    - Refaktoriinnilla tarkoitetaan koodin sisäisen rakenteen muuttamista sen rajapinnan ja toiminnallisuuden säilyessä muuttumattomana
  - 4) Jatketaan askeleesta 1



- TDD:llä ohjelmoitaessa toteutettavaa komponenttia ei yleensä ole tapana suunnitella tyhjentävästi etukäteen
- Testit kirjoitetaan ensisijaisesti ajatellen komponentin käyttäjää
  - huomio on komponentin rajapinnassa ja rajapinnan helppokäyttöisyydessä, ei niinkään komponentin sisäisessä toteutuksessa
- Komponentin sisäinen rakenne muotoutuu refaktorointien kautta

# TDD

- TDD:ssä perinteisen suunnittelu-toteutus-testaus -syklin voi ajatella kääntyneen täysin päinvastaiseen järjestykseen, tarkka oliosuunnittelu tapahtuu vasta refaktorointivaiheiden kautta
- TDD:tä tehtäessä korostetaan yleensä lopputuloksen yksinkertaisuutta, toteutetaan toiminnallisuutta vain sen verran, mitä testien läpimeno edellyttää
  - Ei siis toteuteta "varalta" ekstratoiminnallisuutta, sillä "You ain't gonna need it" (YAGNI)
- Koodista on vaikea tehdä testattavaa jos se ei ole modulaarista ja löyhästi kytketyistä selkeärajapintaisista komponenteista koostuvaan
  - Tämän takia TDD:llä tehty koodi on yleensä laadukasta ylläpidettävyyden ja laajennettavuuden kannalta
- Muita TDD:n hyviä puolia:
  - Rohkaisee ottamaan pieniä askelia kerrallaan ja näin toimimaan fokusoidusti
  - Tehdyt virheet havaitaan nopeasti suuren testijoukon takia
  - Hyvin kirjoitetut testit toimivat toteutetun komponentin rajapinnan dokumentaationa

# TDD

- TDD:llä on myös ikävät puolensa
  - Testikoodia tulee paljon, usein suunnilleen saman verran kuin varsinaista koodia
    - Toisaalta TDD:llä tehty tuotantokoodi on usein hieman normaalista tehtyä koodia lyhempi
  - Jos ja kun koodi muuttuu, tulee testejä ylläpitää
  - TDD:n käyttö on haastavaa (mutta ei mahdotonta) mm. käyttöliittymä-, tietokanta- ja verkkoyhteyksistä huolehtivan koodin yhteydessä
    - testauksen kannalta hankalat komponentit kannattaakin eristää mahdollisimman hyvin muusta koodista, näin on järkevää tehdä, käytettiin TDD:tä tai ei
  - Jo olemassaolevan "legacy"-koodin laajentaminen TDD:llä voi olla haastavaa
- Lisää TDD:stä
  - [http://jamesshore.com/Agile-Book/test\\_driven\\_development.html](http://jamesshore.com/Agile-Book/test_driven_development.html)
  - <http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd>

# Riippuvuudet yksikkötesteissä

- TDD:tä ja muutenkin yksikkötestejä tehdessä on ratkaistava kysymys, miten testeissä suhtaudutaan testattavien luokkien riippuvuuksiin, eli luokkiin, joiden oliota testattava luokka käyttää
- Dependency Injection -suunnittelumalli parantaa luokkien testattavuutta sillä se mahdollistaa riippuvuuksien asettamisen luokille testistä käsin
  - [https://github.com/mluukkai/ohtu2017/blob/master/web/riippuvuuksien\\_injektioti.md](https://github.com/mluukkai/ohtu2017/blob/master/web/riippuvuuksien_injektioti.md)
- Yksi mahdollisuus on tehdä testejä varten riippuvuudet korvaavia tynkäkomponentteja eli stubeja, näin tehtiin mm. viikon 2 tehtävässä 3:
  - <https://github.com/mluukkai/ohtu2017/blob/master/laskarit/2.md>
- Stubeihin voidaan esim. kovakoodata metodikutsujen tulokset valmiiksi
  - Testi voi myös kysellä stubilta millä arvoilla testattava metodi sitä kutsui
- Stubeja on viimeaikoina ruvettu myös kutsumaan **mock-olioiksi**
- Martin Fowlerin artikkeli selventää asiaa ja terminologiaa
  - <http://martinfowler.com/articles/mocksArentStubs.html>
- On olemassa useita kirjastoja mock-olioiden luomisen helpottamiseksi, tutustumme laskareissa Javalle tarkoitettuun *Mockito*-kirjastoon

# Riippuvuudet yksikkötesteissä: Mockito

- Esimerkki viikon 2 laskareista
  - <https://github.com/mluukkai/ohtu2017/blob/master/laskarit/2.md>
- Ostotapahtuman yhteydessä kaupan tulisi veloittaa asiakkaan tililtä ostosten hinta *kutsumalla luokan pankki metodia maksa*:

```
Pankki myNetBank = new Pankki();
```

```
Viitegeneraattori viitteet = new Viitegeneraattori();
```

```
Kauppa kauppa = new Kauppa(myNetBank, viitteet);
```

```
kauppa.aloitaOstokset();
```

```
kauppa.lisaaOstos(5);
```

```
kauppa.lisaaOstos(7);
```

```
kauppa.maksa("1111");
```

- Miten varmistamme, että maksun suorittavaa metodia on kutsuttu?
- Käytetään *mockito*-kirjastoa

# Riippuvuudet yksikkötesteissä: Mockito

- Luodaan testissä kaupan riippuvuuksista mock-oliot:

```
@Test
```

```
public void kutsutaanPankkiaOikeallaTilinumerollaJaSummalla() {
```

```
    Pankki mockPankki = mock(Pankki.class);
```

```
    Viitegeneraattori mockViite = mock(Viitegeneraattori.class);
```

```
    kauppa = new Kauppa(mockPankki, mockViite);
```

```
    kauppa.aloitaOstokset();
```

```
    kauppa.lisaaOstos(5);
```

```
    kauppa.lisaaOstos(5);
```

```
    kauppa.maksa("1111");
```

```
    verify(mockPankki).maksa(eq("1111"), eq(10), anyInt());
```

```
}
```

- Pankkia edustavalle mock-oliolle on asetettu *ekspekaatio*, eli vaatimus joka varmistaa, että metodia *maksa* on kutsuttu testin aikana sopivilla parametreilla

# Riippuvuudet yksikkötesteissä

- Kalvolla 14 esimerkki Rubyn Rspec-kirjastolla tehdystä testistä, jossa testin riippuvuus mockataan
- Testin kohteena on luokka **Action** ja sen metodi **run**
  - Luokan instanssi luodaan rivillä 6, riveillä 3 ja 4 luodun testisyötteen perusteella
  - Testattavan metoden *run* tarkoitus on saada aikaan järjestelmässä erilaisia asioita Action-olion saamista parametreista riippuen
  - Tällä kertaa testin on tarkoitus saada aikaan se, että järjestelmä lähettää sähköpostin (rivin 3 luoman) testisyötteen määrittelemiin osoitteisiin
  - Sähköpostin lähtettämisestä huolehtii järjestelmän komponentti **Engine::Email**, ja sen metodi **perform**, joka saa parametriksi lähetettävän emailin tiedot
  - **Testauksen kohteena olevan luokan Action metodin run tulee siis toimiakseen saada aikaa metodin perform kutsu oikeilla parametreilla**

# Riippuvuudet yksikkötesteissä

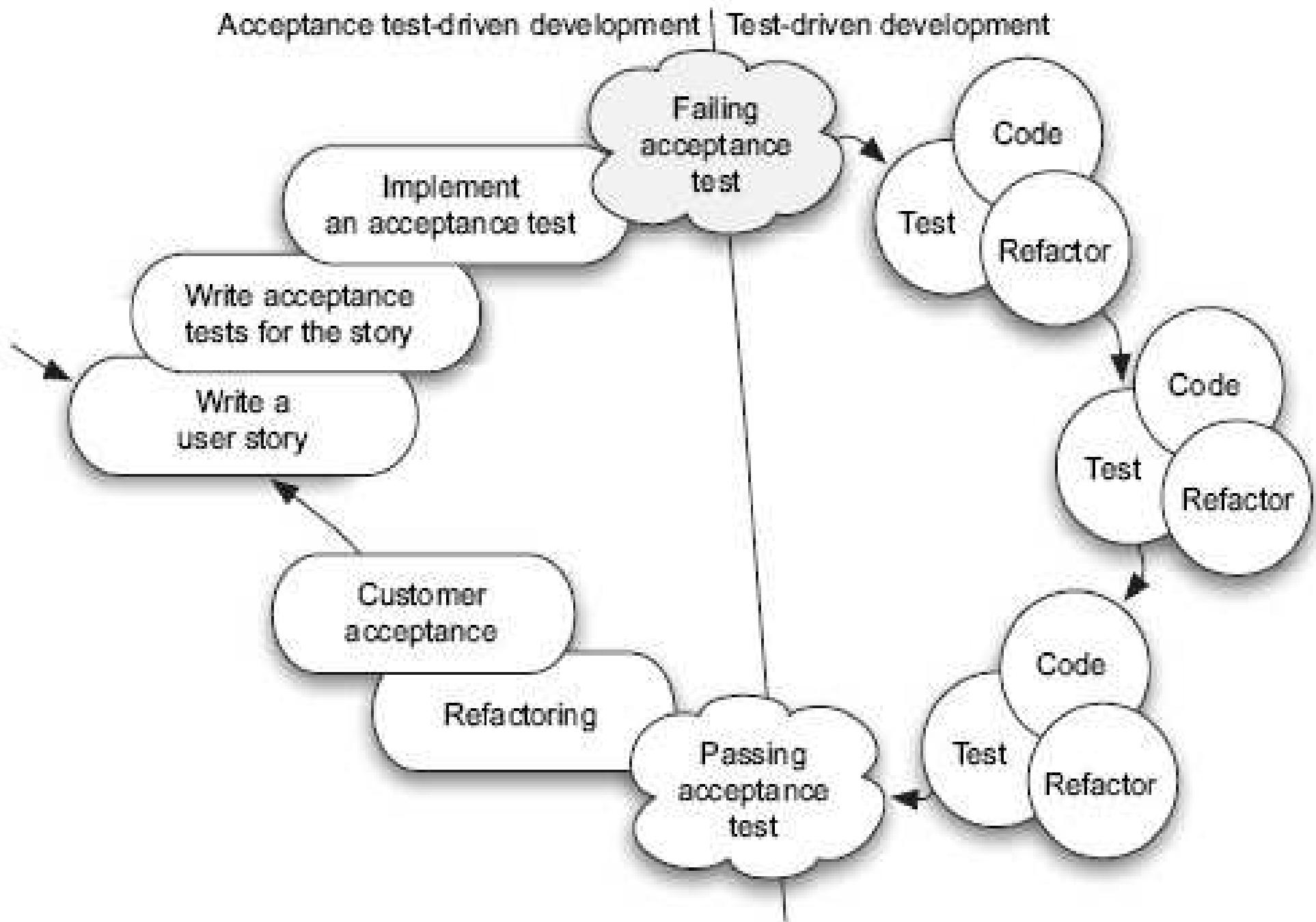
- Riveillä 8-13 asetetaan mock-kirjaston avulla *ekspekaatio eli vaatimus*, jonka mukaan testin suorituksen jälkeen metodia *perform* tulee olla kutsuttu oikeilla parametreilla
  - Määritellään myös, että mockattu metodikutsu palauttaa kutsujalle totuusarvon true
- Testin viimeisellä rivillä kutsutaan testattavaa metodia *run*
- Testi menee läpi vain jos metodikutsu saa aikaan ekspekaation mukaisen metodikutsun
- Kyseinen esimerkki on eräästä todellisuudessa tuotantokäytössä olevasta järjestelmästä
- Riippuvuuden korvaaminen mockaamalla oli esimerkin tapauksessa erittäin hyödyllistä
  - Testi on nopea sillä se ei riipu emailin lähetyksen kaltaisista hitaista, verkkoyleyden olemassaolosta ja nopeudesta riippuvasta toimenpiteestä

```
1 it "task creation triggers email sending to recipients" do
2
3   event = build_event "...simulate task creation..."
4   rule = Rule.find_by name:"task_creation_triggers_email_sending"
5
6   action = Action.create(rule, event)
7
8   Engine::Email.should_receive(:perform).with(
9     to: [ "avihavai@cs.helsinki.fi", "josalmi@cs.helsinki.fi" ],
10    from: "mluukkai@iki.fi",
11    subject: "testi",
12    body: "testiviesti"
13  ).and_return( true )
14
15   action.run()
```

# User Storyjen testaaminen

- Luennon 2 kalvolla 16 mainittiin, että tärkeä osa User Storyn käsitettä ovat Storyn hyväksymätestit, eli Mike Cohnin sanoin:
  - *Tests that convey and document details and that will be used to determine that the story is complete*
- User Storyt kuvaavat loppukäyttäjän kannalta arvoa tuottavia toiminnallisuksia, esim:
  - *Asiakas voi lisätä oluen ostoskoriin*
  - Myös hyväksymätestit on tarkoituksenmukaista ilmaista käyttäjän kielellä
    - Usein pidetään hyvänä asiana, että asiakas on mukana laatimassa hyväksymätestejä
  - Edellisen User storyn hyväksymätestejä voisivat olla
    - Ollessaan tuotelistauksessa ja valitessaan tuotteen jota on varastossa, menee tuote ostoskoriin ja ostoskorin hinta sekä korissa olevien tuotteiden määrä päivittyy oikein
    - Ollessaan tuotelistauksessa ja valitessaan tuotteen jota ei ole varastossa, pysyy ostoskorin tilanne muuttumattomana
  - Storyn hyväksymätestit on tarkoituksenmukaista kirjoittaa heti Storyn toteuttavan sprintin alussa

- Jos näin tehdään voidaan sprintissä tapahtuva ohjelmistokehitys ajatella hyväksymätestien tasolla tapahtuvana TDD:nä



# User Storyjen testaaminen

- Tälläisestä käytännöstä käytetään nimitystä *Acceptance Test Driven Development, ATDD*
  - ATDD:stä käytetään myös nimiä StoryTest Driven Development ja Customer Test Driven Development
  - <http://testobsessed.com/wp-content/uploads/2011/04/atddexample.pdf>
  - <http://www.methodsandtools.com/archive/archive.php?id=23>
  - <http://www.methodsandtools.com/archive/archive.php?id=72>
  - [www.industriallogic.com/papers/storytest.pdf](http://www.industriallogic.com/papers/storytest.pdf)
- Osittain sama idea kulkee nimellä *Behavior Driven Development, BDD*
  - <http://dannorth.net/introducing-bdd/>
- ATDD:ssä sovelluskehityksen lähtökohta on User story eli asiakkaan tasolla mielekäs toiminnallisuus
  - Asiakkaan terminologialla yhdessä asiakkaan kanssa kirjoitetut hyväksymätestit määrittelevät toiminnallisuuden ja näin ollen korvaavat perinteisen vaatimusdokumentin
  - Testien kirjoittamisprosessi lisää asiakkaan ja tiimin välistä kommunikaatiota

# Hyväksymätestauksen työkalut

- Yleensä hyväksymätesteistä pyritään tekemään automaattisesti suoritettavia, käytössä olevia työkaluja mm:
  - Fitnesse, FIT, Robot (ATDD)
  - Cucumber ja JBehave (BDD)
- ATDD:ssä ja BDD:ssä on kyse lähes samasta asiasta pienin painotuseroin
  - BDD kiinnittää testeissä käytettävän terminologian tarkemmin, BDD ei mm. puhu ollenkaan testeistä vaan spesifikaatioista
  - BDD:llä voidaan tehdä myös muita kuin hyväksymätason testejä
  - kurssilla käytämme pääosin BDD:n nimentäkäytäntöjä
- Tutustumme johtavaan BDD-työkaluun Cucumberiin
  - <https://cucumber.io>
- Kuten kaikissa ATDD/BDD-työkaluissa, testit kirjoitetaan asiakkaan kielellä
- Ohjelmoija kirjoittaa testeistä mäppäyksen koodiin, näin testeistä tulee automaattisesti suoritettavia

# Cucumber

- Tarkastellaan esimerkkinä käyttäjätunnuksen luomisen ja sisäänsijautumisen tarjoamaa palvelua
- Palvelun vaatimuksen määrittelevät User Storyt
  - A new user account can be created if a proper unused username and a proper password are given
  - User can log in with a valid username/password-combination
- Cucumberissa jokaisesta User Storystä kirjoitetaan oma .feature-päätteinen tiedosto, joka sisältää
  - nimen ja
  - joukon storyyn liittyvä hyväksymätestejä joita Cucumber kutsuu *skenarioiksi*
- Storyn hyväksymätestit eli *skenaariot* kirjoitetaan *Gherkin*-kielellä, muodossa
  - *Given [initial context], when [event occurs], then [ensure some outcomes]*
- Esimerkki seuraavalla sivulla

**Feature:** User can log in with valid username/password-combination

**Scenario:** user can login with correct password

**Given** command login is selected

**When** username "pekka" and password "akkep" are entered

**Then** system will respond with "logged in"

**Scenario:** user can not login with incorrect password

**Given** command login is selected

**When** username "pekka" and password "wrong" are entered

**Then** system will respond with "incorrect username or password"

**Scenario:** nonexistent user can not login to

**Given** command login is selected

**When** username "nonexisting" and password "wrong" are entered

**Then** system will respond with "incorrect username or password"

# Cucumber: skenaarioiden mäppäys koodiksi

- Ideana on että asiakas tai product owner kirjoittaa tiimissä olevien testaajien tai tiimiläisten kanssa yhteistyössä Storyn liittyvät testit
  - Samalla Storyn haluttu toiminnallisuus tulee dokumentoitua sillä tarkkuudella, että ohjelmoijat toivon mukaan ymmärtävät mistä on kyse
- Skenaariot muutetaan automaattisesti suoritettaviksi testeiksi kirjoittamalla niistä *mäppäys* ohjelmakoodiin
  - Ohjelmoijat tekevät mäppäyksen siinä vaiheessa, kun tuotantokoodia on tarpeellinen määärä valmiina
- Esimerkki seuraavalla sivulla
- Käytännössä jokaista testin given, when ja then -askelta vastaa oma metodinsa
  - Metodit kutsuvat ohjelman luokkia simuloiden käyttäjän syötettä
  - varmistaen että ohjelma reagoi käyttäjän toimiin halutulla tavalla
- Palaamme cucumberiin laskareissa

```
public class Stepdefs {
    App app;
    StubIO io;
    AuthenticationService auth = new AuthenticationService(new InMemoryUserDao());
    List<String> inputLines = new ArrayList<>();

    @Given("^command login is selected$")
    public void command_login_selected() throws Throwable {
        inputLines.add("login");
    }

    @When("^username \"([^\"]*)\" and password \"([^\"]*)\" are entered$")
    public void a_username_and_password_are_entered(String username, String password) {
        inputLines.add(username);
        inputLines.add(password);

        io = new StubIO(inputLines);
        app = new App(io, auth);
        app.run();
    }

    @Then("^system will respond with \"([^\"]*)\"$")
    public void system_will_respond_with(String expectedOutput) {
        assertTrue(io.getPrints().contains(expectedOutput));
    }
}
```

# Websovellusten testien automatisointi

- Olemme jo nähtneet, miten dependency injectionin avulla on helppo tehdä komentoriviltä toimivista ohjelmista testattavia
- Myös Java Swing- ja muilla käyttöliittymäkirjastoilla sekä web-selaimella käytettävien sovellusten automatisoitu testaaminen on mahdollista
- Tutustumme laskareissa Web-sovellusten testauksen automatisointiin käytettävään Selenium 2.0 WebDriver -kirjastoon
  - [http://seleniumhq.org/docs/03\\_webdriver.html](http://seleniumhq.org/docs/03_webdriver.html)
- Selenium tarjoaa rajapinnan, jonka avulla on mahdollisuus simuloida ohjelmakoodista tai testikoodista käsin selaimen toimintaa, esim. linkkien klikkauksia ja tiedon syöttämistä lomakkeeseen
  - Selenium Webdriver -rajapinta on käytettävissä lähes kaikilla ohjelmointikielillä
- Seleniumia käyttävät testit voi tehdä normaalina testikoodin tapaan joko JUnit- tai Cucumber-testeinä
- Katsotaan esimerkinä käyttäjätunnuksista ja sisäänsijautumisesta huolehtivan järjestelmän web-versiota
  - Asiaan palataan tarkemmin viikon 4 laskareissa

```
public class Stepdefs {  
    WebDriver driver = new ChromeDriver();  
    String baseUrl = "http://localhost:4567";  
  
    @Given("^login is selected$")  
    public void login_selected() throws Throwable {  
        driver.get(baseUrl);  
        WebElement element = driver.findElement(By.linkText("login"));  
        element.click();  
    }  
  
    @When("^username \"([^\"]*)\" and password \"([^\"]*)\" are given$")  
    public void username_and_password_are_given(String username, String password) {  
        WebElement element = driver.findElement(By.name("username"));  
        element.sendKeys(username);  
        element = driver.findElement(By.name("password"));  
        element.sendKeys(password);  
        element = driver.findElement(By.name("login"));  
        element.submit();  
    }  
  
    @Then("^system will respond \"([^\"]*)\"$")  
    public void system_will_respond(String pageContent) {  
        assertTrue(driver.getPageSource().contains(pageContent));  
    }  
}
```

# Pois integraatiohelvetistä

- Vesiputousmallissa eli lineaarisesti etenevässä ohjelmistotuotannossa ohjelmiston toteutusvaiheen päätää integrointivaihe
  - yksittään testatut komponentit integroidaan yhdessä toimivaksi kokonaisuudeksi
  - suoritetaan integraatiotestaus, joka varmistaa yhteistoiminnallisuuden
- Perinteisesti juuri integrointivaihe on tuonut esiin suuren joukon ongelmia
  - tarkasta suunnittelusta huolimatta erillisten tiimien toteuttamat komponentit rajapinnoiltaan tai toiminnallisuudeltaan epäsopivia
  - "integrointihelvetti" <http://c2.com/cgi/wiki?IntegrationHell>
- Suurten projektien integrointivaihe on kestänyt ennakoimattoman kauan
  - integrointivaiheen ongelmat ovat aiheuttaneet ohjelmaan suunnittelutason muutoksia
- 90-luvulla alettiin huomaamaan, että riskien minimoimiseksi integraatio kannattaa tehdä useammin kuin vain projektiin lopussa
- best practiceksi muodostui päivittääin tehtävä koko projektin käänäminen *daily/nightly build* ja samassa yhteydessä *ns. smoke test*:in suorittaminen
  - <http://www.stevemcconnell.com/ieeesoftware/bp04.htm>

# Päivittäisestä jatkuvaan integraatioon

- Smoke test:
  - The smoke test should exercise the entire system from end to end. It does not have to be exhaustive, but it should be capable of exposing major problems
- Daily buildia ja smoke testiä käytettäessä järjestelmän integraatio tehdään (ainakin jollain tarkkuustasolla) joka päivä
  - Komponenttien yhtensopivuusongelmat huomataan nopeasti ja niiden korjaaminen helpottuu
  - Tiimin moraali paranee, kun ohjelmistosta on olemassa päivittäin kasvava toimiva versio
- Mahdollisimman usein tapahtuva integraatiovaihe todettiin hyväksi käytännöksi. Tästä syntyi idea toistaa integraatiota vielä päivittäistä sykliäkin useammin: **jatkuva integraatio eli continuous integration**
  - <http://martinfowler.com/articles/continuousIntegration.html>
  - [http://jamesshore.com/Agile-Book/continuous\\_integration.html](http://jamesshore.com/Agile-Book/continuous_integration.html)
- Integraatiovaiheen yllätysten minimoinnin lisäksi jatkuvassa integraatiossa on tarkoitus eliminoida "but it works on my machine"-ilmiö

# Jatkuva integraatio – Continuous Integration

- Integraatiosta tarkoitus tehdä todella vaivaton operaatio, "nonevent", ohjelmistosta koko ajan olemassa integroitu ja testattu tuore versio
- Ohjelmisto ja kaikki konfiguraatiot pidetään keskitetyssä repositorioissa
- Koodi sisältää kattavat automatisoidut testit
  - Yksikkö-, integraatio- ja hyväksymätason testejä
- Yksittäinen palvelin, jonka konfiguraatio vastaa tuotantopalvelimen konfiguraatiota, varattu CI-palvelimeksi
- CI-palvelin tarkkailee repositoriota ja jos huomaa siinä muutoksia, hakee koodin, käänää sen ja ajaa testit
  - Jos koodi ei käänny tai testit eivät mene läpi, seurauksena poikkeustilanne joka korjattava välittömästi: ***do not break the build***
- Sovelluskehittäjän työprosessi etenee seuraavasti
  - Haetaan repositorista koodin uusi versio
  - Toteutetaan työn alla oleva toiminnallisuus ja sille automatisoidut testit
  - Integroidaan kirjoitettu koodi suoraan muun koodin yhteyteen
  - Kun työ valmiina, haetaan repositorioon tulleet muutokset ja ajetaan testit

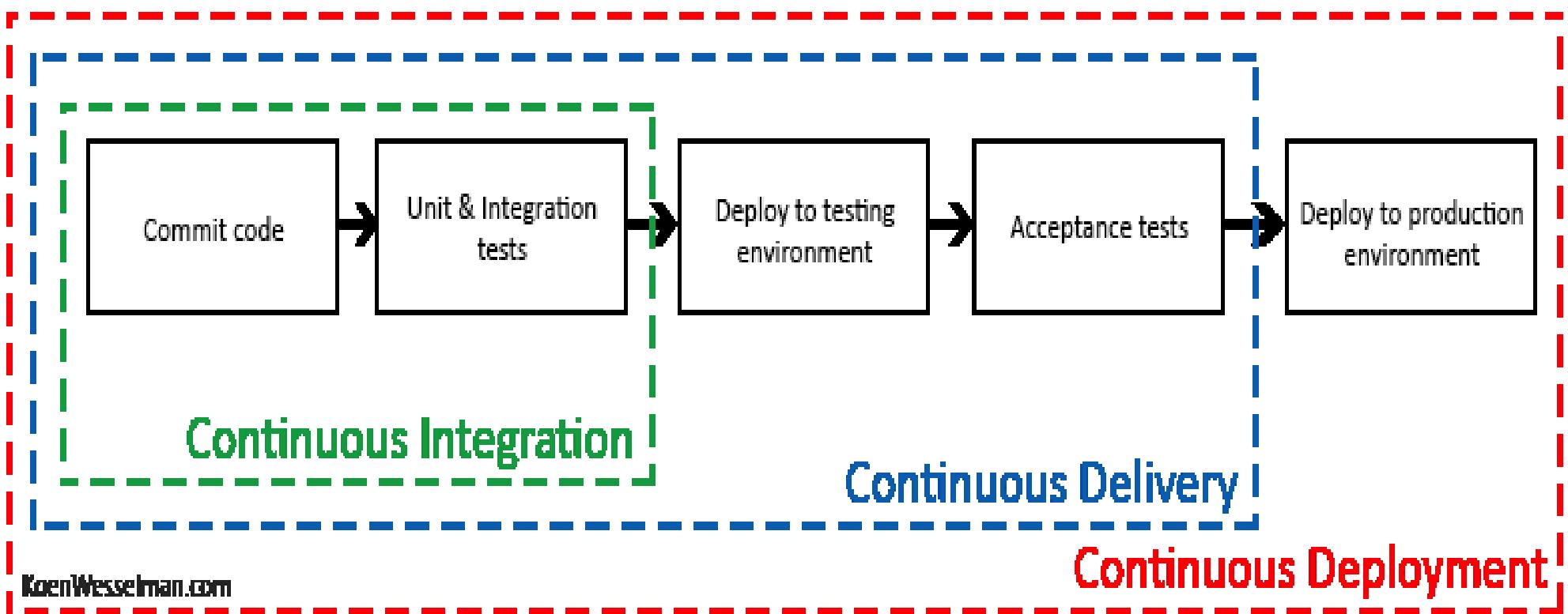
- Kun kehittäjän omalla koneella kaikki testit menevät läpi ja koodi on integroitu muuhun ohjelmakoodiin, pushaa kehittäjä koodin repositorioon
- CI-palvelin huomaa tehdyt muutokset, hakee koodit ja suorittaa testit
- Näin minimoituu mahdollisuus sille, että lisätty koodi toimii esim. konfiguraatioerojen takia ainoastaan kehittäjän paikallisella työasemalla
- Tarkoituksena on, että jokainen kehittäjä integroi tekemänsä työn muuhun koodiin mahdollisimman usein, *vähintään* kerran päivässä
  - CI siis rohkaisee jakamaan työn pieniin osiin, sellaisiin jotka saadaan testeineen "valmiaksi" yhden työpäivän aikana
  - CI-työprosessin noudattaminen vaatii kurinalaisuutta
- Jotta CI-prosessi toimisi joustavasti, tulee testien ajamisen tapahtua suhteellisen nopeasti, maagisena rajana pidetään usein kymmentä minuuttia
- Jos osa testeistä on hitaita, voidaan testit konfiguroida ajettavaksi kahdessa vaiheessa
  - *commit build*:in läpimeno antaa kehittäjälle oikeuden pushata koodi repositorioon
  - CI-palvelimella suoritetaan myös hitaamat testit sisältävä *secondary build*
- Ensimmäisellä viikolla käyttämämme <https://travis-ci.org> on tämän hetken ehkä eniten huomiota saanut CI-palvelinohjelmisto. Eräs travisin suurista eduista on se, että ohjelmisto toimii pilvessä ja tarvetta oman CI-palvelimen asentamiselle ei ole

# Jatkuva toimitusvalmius ja käyttöönotto

- Viime aikoina nousseen trendin mukaan CI:tä viedään vielä askel pidemmälle ja integraatioprosessiin lisätään myös automaattinen "deployaus"
  - käännetty ja testattu koodi siirretään suoritettavaksi ns. *staging*- eli testipalvelimelle
- **Staging-palvelin**, on ympäristö, joka on konfiguraatioidensa ja myös sovelluksen käsittelemän datan osalta mahdollisimman lähellä varsinaista tuotantoymäristöä
- Kun ohjelmiston uusi versio on deployattu staging-palvelimelle, suoritetaan sille hyväksymätestit
- Hyväksymätestien suorittamisen jälkeen uusi versio voidaan siirtää tuotantopalvelimelle
- Parhaassa tapauksessa myös hyväksymätestien suoritus on automatisoitu, ja ohjelmisto kulkee koko **deployment pipelinen** läpi, eli sovelluskehittäjän koneelta CI-palvelimelle, sieltä stagingiin ja lopulta tuotantoon, automaatisesti

# Jatkuva toimitusvalmius ja käyttöönotto

- Käytännöstä, jossa jokainen CI:n läpäisevä ohjelmiston uusi versio viedään staging-palvelimelle ja siellä tapahtuvan hyväksymätestauksen jälkeen tuotantoon, käytetään nimitystä **jatkuva toimitusvalmius** engl. **continuous delivery**
- Jos staging-palvelimella ajettavat testit ja siirto tuotantopalvelimelle tapahtuvat automatisesti, puhutaan **jatkuvasta käyttöönnotosta** engl. **continuous deployment**
- Viime aikoina on ruvettu suosimaan tyyliä, jossa web-palveluna toteutettu ohjelmisto julkaistaan tuotantoon jopa useita kertoja päivästä



# Tutkiva testaaminen

- Jotta järjestelmä saadaan niin virheettömäksi, että se voidaan laittaa tuotantoon, on testauksen oltava erittäin perusteellinen
- Perinteinen tapa järjestelmätestauksen suorittamiseen on ollut laatia ennen testausta hyvin perinpohjainen testaussuunnitelma
  - Jokaisesta testistä on kirjattu testisyötteet ja odotettu tulos
  - Testauksen tuloksen tarkastaminen on suoritettu vertaamalla järjestelmän toimintaa testitapaukseen kirjattuun odotettuun tulokseen
- Automatisoitujen hyväksymätestien luonne on täsmälleen samanlainen, syöte on tarkkaan kiinnitetty samoin kuin odotettu tuloskin
- Jos testaus tapahtuu pelkästään etukäteen mietittyjen testien avulla, ovat ne kuinka tarkkaan tahansa harkittuja, ei kaikkia yllättäviä tilanteita osata välttämättä ennakoida
- Hyvät testaajat ovat kautta aikojen tehneet "virallisen" dokumentoidun testauksen lisäksi epävirallista "ad hoc"-testausta
- Pikkuhiljaa "ad hoc"-testaus on saanut virallisen aseman ja sen strukturoitua muotoa on ruvettu nimittämään **tutkivaksi testaamiseksi** (exploratory testing)

# Tutkiva testaaminen

- *Exploratory testing is simultaneous learning, test design and test execution*
  - [www.satisfice.com/articles/et-article.pdf](http://www.satisfice.com/articles/et-article.pdf)
  - [http://www.satisfice.com/articles/what\\_is\\_et.shtml](http://www.satisfice.com/articles/what_is_et.shtml)
- Ideana on, että testaaja ohjaa toimintaansa suorittamiensa testien tuloksen perusteella
- Testitapauksia ei suunnitella kattavasti etukäteen, vaan testaaja pyrkii kokemuksensa ja suoritettujen testien perusteella löytämään järjestelmäästä virheitä
- Tutkiva testaus ei kuitenkaan etene täysin sattumanvaraisesti
- Testaussessiolle asetetaan jonkinlainen tavoite
  - Mitä tutkitaan ja minkälaisia virheitä etsitään
- Ketterässä ohjelmistotuotannossa tavoite voi hyvin jäsentyä yhden tai useamman User storyn määrittelemän toiminnallisuuden ympärille
  - Esim. testataan ostosten lisäystä ja poistoa ostoskorista

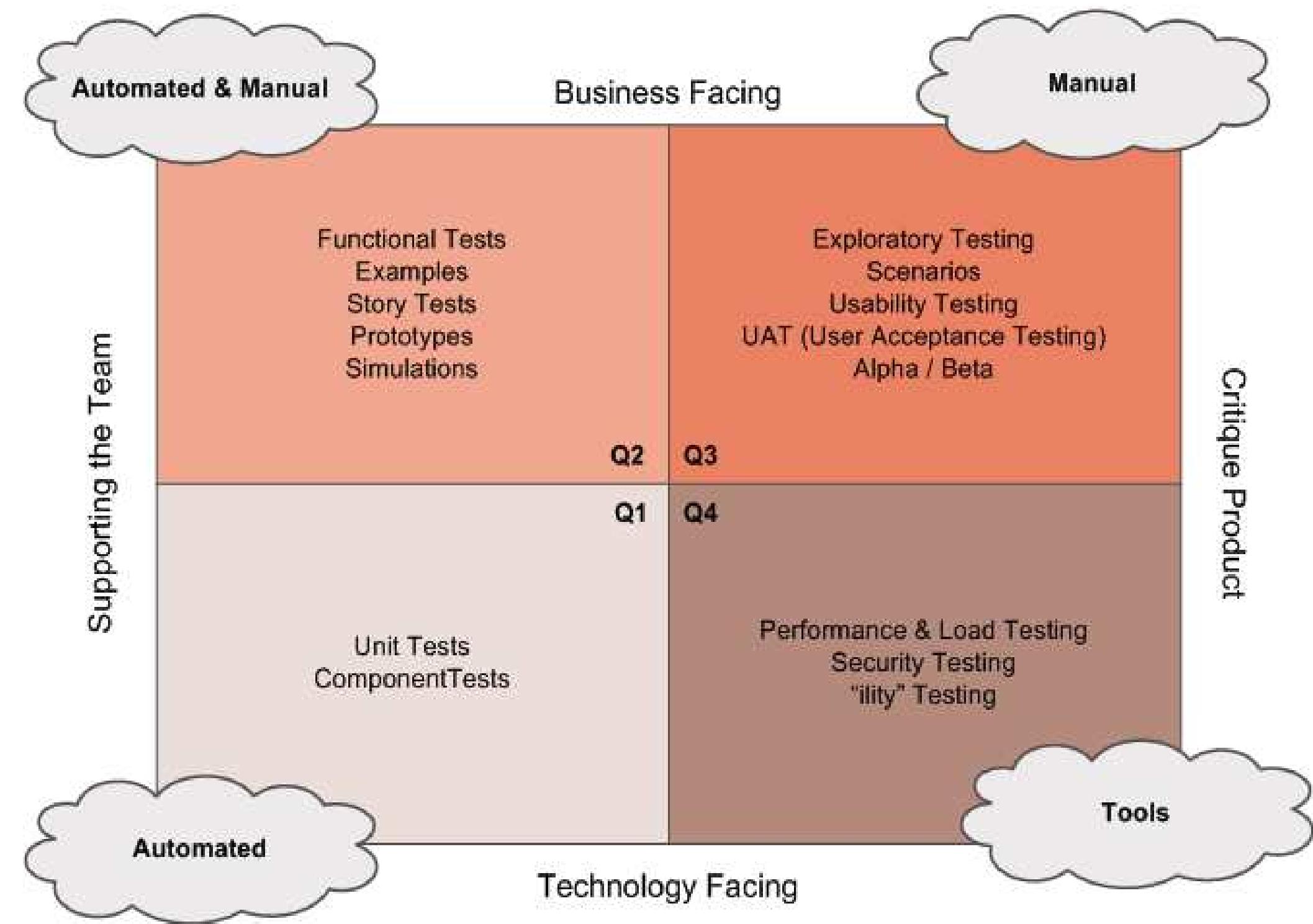
# Tutkiva testaaminen

- Tutkivassa testauksessa keskeistä on kaiken järjestelmän tekemien asioiden havainnointi
  - Normaaleissa etukäteen määritellyissä testeissähän havainnoidaan ainoastaan reagoiko järjestelmä odotetulla tavalla
  - Tutkivassa testaamisessa kiinnitetään huomio myös varsinaisen testattavan asian ulkopuoleisiin asioihin
- Esim. jos huomattaisiin selaimen osoiterivillä URL  
<http://www.kumpulabiershop.com/ostoskorri?id=10>voitaisiin yrittää muuttaa käsin ostoskorin id:tä ja yrittää saada järjestelmä epästabiiliin tilaan
- Tutkivan testaamisen avulla löydettyjen virheiden toistuminen jatkossa kannattaa eliminoida lisäämällä ohjelmalle sopivat automaattiset regressiotestit
  - Tutkivaa testaamista ei siis kannata käyttää regressiotestaamisen menetelmänä vaan sen avulla kannattaa ensisijaisesti testata sprintin yhteydessä toteutettuja uusia ominaisuuksia
- Tutkiva testaaminen siis ei ole vaihtoehto normaaleille tarkkaan etukäteen määritellyille testeille vaan niitä täydentävä testauksen muoto

# Loppupäätelmiä testauksesta

- Seuraavalla sivulla alunperin Brian Maricin ketterän testauksen kenttää jäsentävä kaavio *Agile Testing Quadrants*
  - <http://lisacrispin.com/2011/11/08/using-the-agile-testing-quadrants/>
  - <http://www.exampler.com/old-blog/2003/08/22/#agile-testing-project-2>
  - Kaavio on jo hieman vanha, alunperin vuodelta 2003
- Ketterän testauksen menetelmät voidaan siis jakaa neljään luokkaan (Q1...Q4) seuraavien dimensioiden suhteen
  - Business facing ... technology facing
  - Supporting team ... critique to the product
- Testit ovat suurelta osin automatisoitavissa, mutta esim. tutkiva testaaminen ja käyttäjän hyväksymätestaus ovat luonteeltaan manuaalista työtä edellyttäviä
- Kaikilla ”neljänneksillä” on oma roolinsa ja paikkansa ketterissä projekteissa, ja on pitkälti kontekstisidonnaista missä suhteessa testaukseen ja laadunhallintaan käytettävissä olevat resurssit kannattaa kohdentaa

# Agile Testing Quadrants



# Loppupäätelmiä testauksesta

- Ketterissä menetelmissä kantavana teemana on arvon tuottaminen asiakkaalle
- Tätä kannattaa käyttää ohjenuorana myös arvioitaessa mitä ja miten paljon projektissa tulisi testata
- Testauksella ei ole itseisarvoista merkitystä, mutta testaamattomuus alkaa pian heikentää tuotteen laatua liikaa
- Joka tapauksessa testausta ja laadunhallintaa on tehtävä paljon ja toistuvasti, tämän takia testauksen automatisointi on yleensä pidemmällä tähtäimellä kannattavaa
- Testauksen automatisointi ei ole halpaa eikä helppoa ja väärin, väärään aikaan tai väärälle ”tasolle” tehdyt automatisoidut testit voivat tuottaa enemmän harmia ja kustannuksia kuin hyötyä
- Jos ohjelmistossa on komponentteja, jotka tullaan ehkä poistamaan tai korvaamaan pian, saattaa olla järkevää olla automatisoimatta niiden testejä
  - Vrt luennolla 3 esitelty Minimal Viable Product
  - Ongelmallista kuitenkin usein on, että tästä ei tiedetä yleensä ennalta ja pian poistettavaksi tarkoitettu komponentti voi jäädä järjestelmään pitkäksiin aikaa

- Kattavien yksikkötestien tekeminen ei välttämättä ole mielekästä ohjelman kaikille luokille, parempi vaihtoehto voi olla tehdä integraatiotason testejä ohjelman isompien komponenttien rajapintoja vasten
  - Testit pysyvät todennäköisemmin valideina komponenttien sisäisen rakenteen muuttuessa
- Yksikkötestaus lienee hyödyllisimmillään kompleksia logiikkaa sisältäviä luokkia testattaessa
- Oppikirjamääritelmän mukaista TDD:tä sovelletaan melko harvoin
- Välillä kuitenkin TDD on hyödyllinen väline, esim. testattaessa rajapintoja, joita käyttäviä komponentteja ei ole vielä olemassa. Testit tekee samalla vaivalla kuin koodia käyttävän "pääohjelman"
- Testitapauksista kannattaa aina tehdä mahdollisimman paljon testattavan komponentin oikeita käytöskenaarioita vastaavia, pelkkiä testauskattavuutta kasvattavia testejä on turha tehdä
- Automaattisia testejä kannattaa kirjoittaa mahdollisimman paljon etenkin niiden järjestelmän komponenttien rajapointoihin, joita muokataan usein
- Liian aikaisessa vaiheessa projektia tehtävät käyttöliittymän läpi suoritettavat testit saattavat aiheuttaa kohtuuttoman paljon ylläpitovaivaa, sillä testit hajoavat helposti pienistäkin käyttöliittymään tehtävistä muutoksista

Ohjelmistotuotanto

Luento 7

3.4.

Testaus ketterissä menetelmissä, nopea kertaus..

# Ketterien menetelmien testauskäytänteitä

- Testauksen rooli ketterissä menetelmissä poikkeaa huomattavasti vesiputousmallisesta ohjelmistotuotannosta
  - Testaus on integroitu kehitysprosessiin ja testaajat työskentelevät osana kehittäjätiimejä
  - Testausta tapahtuu projektin "ensimmäisestä päivästä" lähtien
  - Toteutuksen iteratiivisuus tekee regressiotestauksen automatisoinnista erityisen tärkeää
- Viimeksi puhuimme neljästä ketterästä testaamisen menetelmästä:
  - **Test driven development (TDD)**
  - **Acceptance Test Driven Development / Behavior Driven Development**
    - Etenkin TDD:ssä on on kyse enemmän ohjelman suunnittelusta kuin testaamisesta. sivutuotteena syntyy toki kattava joukko testejä
  - **Continuous Integration (CI)** suomeksi jatkuva integraatio
    - Moderni kehitys on kulkenut kohti **Continuous deploymentiä** eli automaattisesti tapahtuvaa jatkuvaa tuotantoonvientiä
  - **Exploratory testing**, suomeksi tutkiva testaus

# Ohjelmiston suunnittelu

# Ohjelmiston suunnittelu ja toteutus

- Riippumatta tyylistä ja tavasta jolla ohjelmisto tehdään, ohjelmistojen tekeminen sisältää
  - vaatimusten analysoinnin ja määrittelyn
  - suunnittelun
  - toteuttamisen
  - testauksen ja
  - ohjelmiston ylläpidon
- Olemme käsitelleet vaatimusmäärittelyä ja testaamista erityisesti ketterien ohjelmistotuotantomenetelmien näkökulmasta
- Siirrymme kurssin "viimeisessä osassa" käsittelemään ohjelmiston *suunnittelua ja toteuttamista*
  - Osa suunnittelusta tapahtuu vasta toteutusvaiheessa, joten suunnittelun ja toteuttamisen käsitellyä ei ole järkevä eriyttää
- **Ohjelmiston suunnittelun tavoitteena on määritellä miten saadaan toteutettua vaatimusmäärittelyn mukaisella tavalla toimiva ohjelma**

# Ohjelmiston suunnittelu

- Suunnittelun ajatellaan yleensä jakautuvan kahteen vaiheeseen:
  - **Arkkitehtuurisuunnittelu**
    - Ohjelman rakenne karkealla tasolla
    - Mistä suuremmista rakennekomponenteista ohjelma koostuu?
    - Miten komponentit yhdistetään, eli komponenttien väliset rajapinnat
  - **Oliosuunnittelu**
    - yksittäisten komponenttien suunnittelu
- Suunnittelun ajoittuminen riippuu käytettävästä tuotantoprosessista:
  - Vesiputousmallissa suunnittelu tapahtuu vaatimusmäärittelyn jälkeen ja ohjelointi aloitetaan vasta kun suunnittelu valmiina ja dokumentoitu
  - Ketterissä menetelmissä suunnittelua tehdään tarvittava määrä jokaisessa iteraatiossa, tarkkaa suunnitteludokumenttia ei yleensä ole
  - Vesiputousmallin mukainen suunnitteluprosessi tuskin on enää juuri missään käytössä, ”jäykimmissäkin” prosesseissa ainakin vaatimusmäärittely ja arkkitehtuurisuunnittelu limittyytä
    - Tarkkaa ja raskasta ennen ohjelointia tapahtuva suunnittelu (BDUF eli Big Design Up Front) toki edelleen tapahtuu ja tietynlaisiin järjestelmiin (hyvin tunnettu sovellusalue, muuttumattomat vaatimukset) se osittain sopiaikin

# Arkkitehtuurisuunnittelu

# Ohjelmiston arkkitehtuuri

- Termiä ohjelmistoarkkitehtuuri (software architecture) on käytetty jo vuosikymmeniä
- Termi on vakiintunut yleiseen käyttöön 2000-luvun aikana ja on siirtynyt mm. "tärkeää työntekijää" tarkoittavaksi nimikkeeksi
  - Ohjelmistoarkkitehti engl. Software architech
- Useimmissa alan ihmisiä on jonkinlainen kuva siitä, mitä ohjelmiston arkkitehtuurilla tarkoitetaan
  - Kyseessä ohjelmiston rakenteen suuret linjat
- Termiä ei ole kuitenkaan yrityksistä huolimatta onnistuttu määrittelemään siten että asiantuntijat olisivat määritelmästä yksimielisiä
- IEEE:n standardi *Recommended practices for Architectural descriptions of Software intensive systems* määrittelee käsitteen seuraavasti
  - *Ohjelmiston arkkitehtuuri on järjestelmän perusorganisaatio, joka sisältää järjestelmän osat, osien keskinäiset suhteet, osien suhteet ympäristöön sekä periaatteet, jotka ohjaavat järjestelmän suunnittelua ja evoluutiota*

# Ohjelmiston arkkitehtuuri, muita määritelmiä

- Krutchen:
  - An architecture is the **set of significant decisions about the organization of a software system**, the selection of structural elements and their interfaces by which the system is composed, together with their behavior as specified in the collaborations among those elements, the composition of these elements into progressively larger subsystems, and the **architectural style** that guides this organization -- these elements and their interfaces, their collaborations, and their composition.
- McGovern:
  - The software architecture of a system or a collection of systems consists of all the important design decisions about the software structures and the interactions between those structures that comprise the systems. **The design decisions support a desired set of qualities that the system should support to be successful.** The design decisions provide a conceptual basis for system development, support, and maintenance.

# Arkkitehtuurin kuuluu

- Vaikka arkkitehtuurin määritelmät hieman vaihtelevat, löytyy määritelmistä joukko samoja teemoja
- Lähes jokaisen määritelmän mukaan arkkitehtuuri määrittelee ohjelmiston rakenteen, eli jakautumisen erillisiin osiin ja osien väliset rajapinnat
- Arkkitehtuuri ottaa kantaa rakenteen lisäksi myös käyttäytymiseen
  - Arkkitehtuuritason rakenneosien vastuut ja niiden keskinäisen kommunikoinnin muodot
- Arkkitehtuuri keskittyy järjestelmän tärkeisiin/keskeisiin osiin
  - Arkkitehtuuri ei siis kuvaaa järjestelmää kokonaisuudessaan vaan on isoihin linjoihin keskittyvä abstraktio
  - Tärkeät osat voivat myös muuttua ajan myötä, eli arkkitehtuuri ei ole muuttumaton
  - <http://www.ibm.com/developerworks/rational/library/feb06/eeles/>

# "Arkkitehtuuri on ne asiat joita on vaikea vaihtaa"

- Artikkelissa "Who needs architect" Martin Fowler toteaa seuraavasti
  - you might end up defining architecture as **things that people perceive as hard to change**
  - <https://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf>
- Melkein sama hieman toisin ilmaistuna oli Krutchenin määritelmässä
  - **set of significant decisions about the organization of a software system**
- Konkreettinen esimerkki täälläisestä arkkitehtoonisesta päätöksestä tuli esiin miniprojektiensä ensimmäisellä viikolla
- Monissa projekteissa oli seuraava User story
  - *Ohjelmaa tulee voida käyttää useilta eri koneilta*
- Story ei varmasti ole millään projektilla priorisoitu kovin korkealle, ja ei ole ainakaan ensimmäisen sprintin tavoitteissa

# "Arkkitehtuuri on ne asiat joita on vaikea vaihtaa"

- Kaikki ryhmät tekevät kuitenkin jo ensimmäisellä viikolla tärkeän ratkaisun (**significant decisions about ...**) sen suhteen miten ohjelma toteutetaan:
  - konsolisovelluksesta
  - Java Swingillä
  - Web-sovelluksena
- Tällä "arkitehtoonisella päätöksellä" on erittäin suuri vaikutus miten story *Ohjelmaa tulee voida käyttää useilta eri koneilta* voidaan toteuttaa siinä vaiheessa kun sen toteututuksen aika tule
- Tämä ensimmäisen viikon tärkeä arkitehtooninen päätös siis johtaa tilanteeseen, joka on myöhemmin **hard to change**
  - Esim. siirtyminen konsolisovelluksesta WebMVC-sovellukseen ei ole suinkaan mahdoton, mutta se edellyttää paljon töitä

# Arkkitehtuurin vaikuttavia tekijöitä

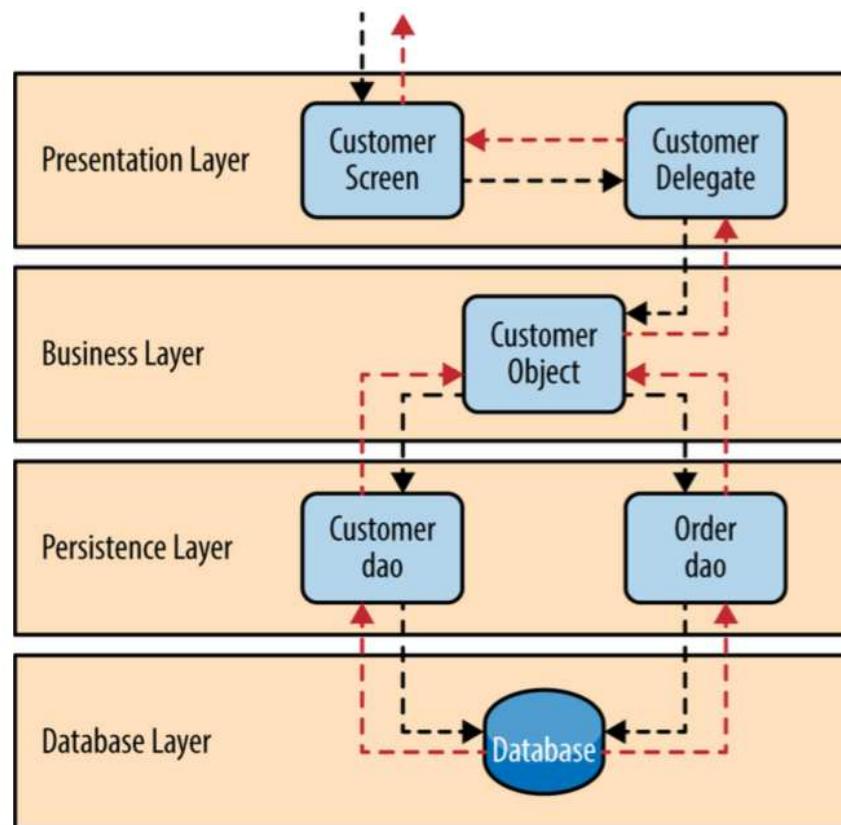
- Järjestelmälle asetetuilla *ei-toiminnallisilla laatuvaatimuksilla* (engl. -ilities) on suuri vaikutus arkkitehtuuriin
  - Käytettävyys, suorituskyky, skaalautuvuus, vikasietoisuus, tiedon ajantasaisuus, tietoturva, ylläpidettävyys, laajennettavuus, hinta, time-to-market, ...
- Laatuvaatimukset ovat usein ristiriitaisia, joten arkkitehdin tulee hakea kaikkia sidosryhmiä tyydyttävä kompromissi
  - Esim. time-to-market lienee ristiriidassa useimpien laatuvaatimusten kanssa
  - Tiedon ajantasaisuus, skaalautuvuus ja vikasietoisuus ovat myös piirteitä, joiden suhteen on pakko tehdä kompromisseja, kaikkia ei voida saavuttaa ks. [http://en.wikipedia.org/wiki/CAP\\_theorem](http://en.wikipedia.org/wiki/CAP_theorem)
- Myös järjestelmän toimintaympäristö ja valitut toteutusteknologiat muokkaavat arkkitehtuuria
  - Organisaation standardit
  - Integraatio olemassaoleviin järjestelmiin
  - Toteutuksessa käytettävät sovelluskehykset

# Arkkitehtuurimalli

- Järjestelmän arkkitehtuuri perustuu yleensä yhteen tai useampaan **arkkitehtuurimalliin** (architectural pattern), jolla tarkoitetaan hyväksi havaittua tapaa strukturoida tietyytyyppisiä sovelluksia
  - Samasta asiasta käytetään joskus nimitystä **arkkitehtuurityyli** (architectural style)
- Arkkitehtuurimalleja on suuri määrä, esim:
  - Kerrosarkkitehtuuri
  - MVC
  - Pipes-and-filters
  - Repository
  - Client-server
  - publish-subscribe
  - event driven
  - REST
  - Microservice
  - SOA
- Useimmiten sovelluksen rakenteesta löytyy monien arkkitehtuuristen mallien piirteitä

# Kerrosarkkitehtuuri

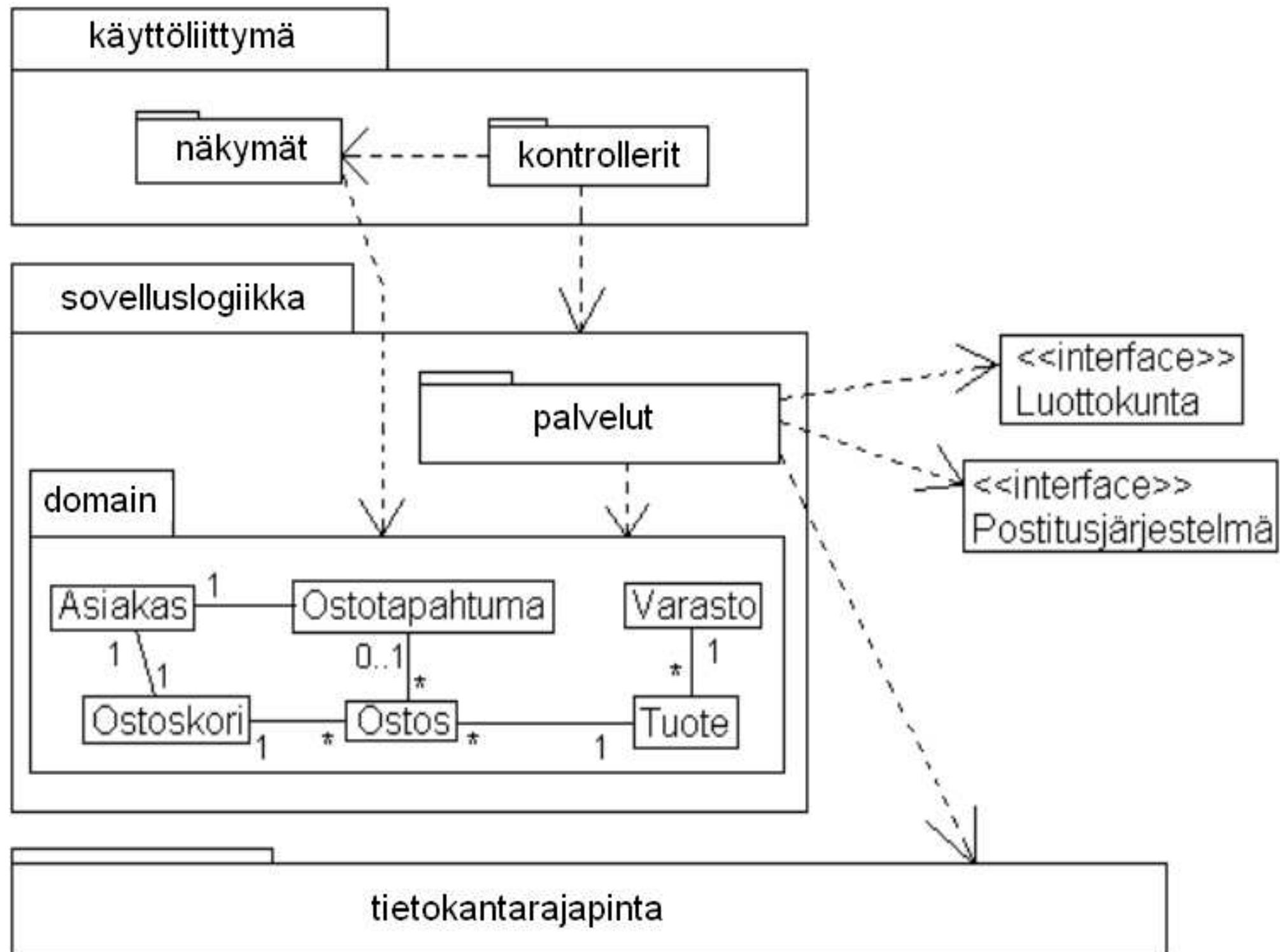
- Eräs tunnetuimmista on *kerrosarkkitehtuurimalli*
  - Kerros on kokoelma toisiinsa liittyviä olioita tai alikomponentteja, jotka muodostavat toiminnallisuuden suhteiden loogisen kokonaisuuden
  - Pyrkimyksenä järjestellä komponentit siten, että ylempänä oleva kerros käyttää ainoastaan alempaan olevien kerroksien tarjoamia palveluita



- Kerrosarkkitehtuuri on sovelluskehittäjän kannalta selkeä mutta saattaa johtaa massiivisiin monoliittisiin sovelluksiin, joita on lopulta vaikea laajentaa ja joiden skaalaaminen suurille käyttäjämääritteille voi muodostua ongelmaksi

# Kumpulabeershop

- Seuraavalla sivulla kuvaus Kumpulabiershopin arkkitehtuurista
- Arkkitehtuuri on mukaelma kerrosarkkitehtuuria (layered architecture) ja MVC-mallia
  - Kuvaus on UML-pakkauskaaviona, näytäen osin myös pakkausten sisäisiä luokkia
  - Luokkatasolle ei yleensä arkkitehtuurikuvaauksissa mennä
- Koodi osoitteessa <https://github.com/mluukkai/BeerShop>
- Koodin tasolla arkkitehtuuri ilmenee luokkien sijoittelusta pakkauksiin
- Ohjelman kaikki koodi on nyt yhdessä projektissa
- Laajempien ohjelmien tapauksessa voi olla tarkoituksenmukaista jakaa koodi useampaan eri projektiin, erityisesti jos sovelluksessa on komponentteja, joita hyödynnetään useimmissa ohjelmissa
- Jos sovellus koostuu eri projekteissa toteutetuista komponenteista, "pääprojekti" sisällyttää silloin aliprojekteissa toteutetut komponentit esim. gradle-riippuvuuksina

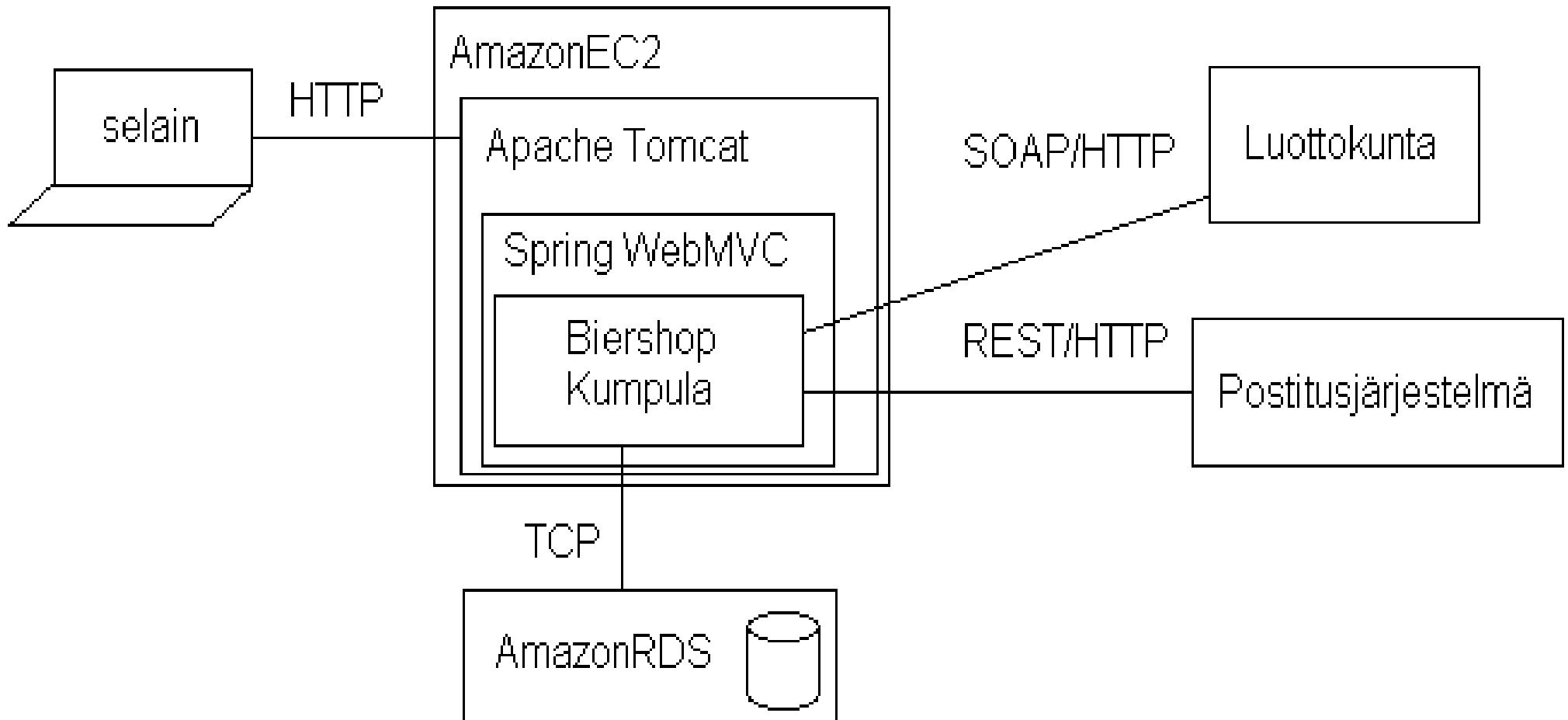


# Kumpula biershopin arkkitehtuuri

- Arkkitehtuurikuvaus näyttää järjestelmän jakaantumisen kolmeen kerroksittain järjestettyyn komponenttiin
  - Käyttöliittymä
  - Sovelluslogiikka
  - Tietokantarajapinta
- Sovelluslogiikkakerros on jaettu vielä kahteen alikomponenttiin, sovellusalueen käsitteistön sisältävään *domainiin* ja sen olioita käytäviin sekä tietokantarajapinnan kanssa keskusteleviin palveluihin
- Käyttöliittymäkerros on myös jakautunut kahteen osaan, näkymään ja kontrollereihin
  - Käytännössä näkymällä tarkoitetaan HTML-tiedostoja
  - Kontrollereilla taas tarkoitetaan main-metodin sisällä olevia selaimen tekemien pyyntöjen käsittelymetodeja
- Kuva tarjoaa **loogisen näkymän** arkkitehtuuriin mutta ei ota kantaa siihen mihin eri komponentit sijoitellaan, eli toimiiko esim. käyttöliittymä samassa koneessa kuin sovelluksen käyttämä tietokanta

# Kumpula biershopin arkkitehtuuri

- Alla **fyysisen tason kuvaus**, josta selviää että kyseessä on selaimella käytettävä, SpringWebMVC-sovelluskehysellä tehty sovellus, jota suoritetaan AmazonEC2-palvelimella ja tietokantana on AmazonRDS
  - Myös kommunikointitapa järjestelmän käyttämiin ulkoisiin järjestelmiin (Luottokunta ja Postitusjärjestelmä) selviää kuvasta

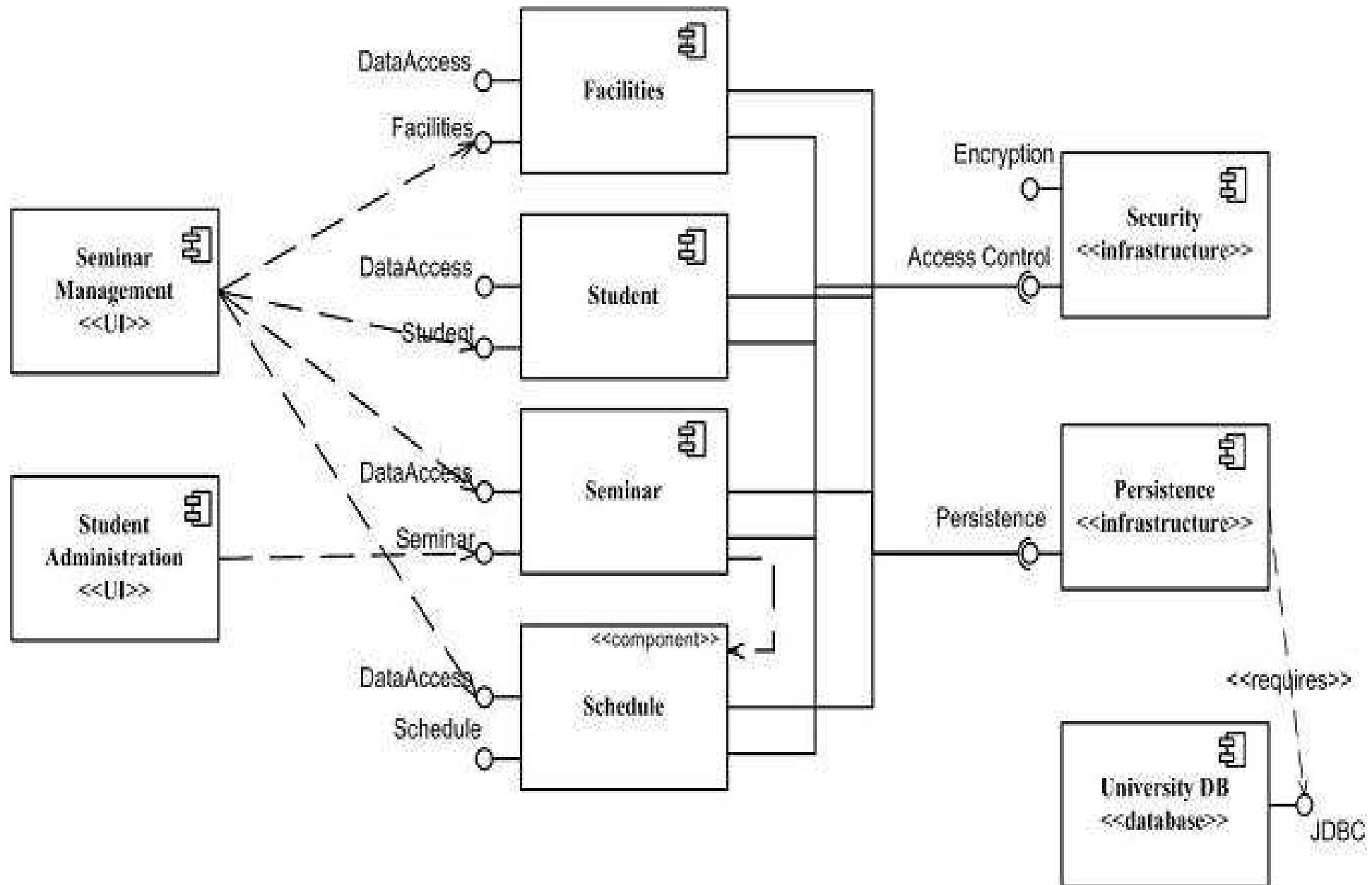


# Arkkitehtuurin kuvaamisesta

- UML:n lisäksi arkkitehtuurikuvausille ei ole vakiintunutta formaattia
  - Luokka ja pakkauskaavioiden lisäksi UML:n komponentti- ja sijoittelukaaviot voivat olla käyttökelpoisia (ks. seuraavat kalvot)
  - Usein käytetään epäformaaleja laatikko/nuoli-kaavioita
- Arkkitehtuurikuvaus kannattaa tehdä useasta eri *näkökulmasta*, sillä eri näkökulmat palvelevat erilaisia tarpeita
  - Korkean tason kuvauksen avulla voidaan strukturoida keskusteluja eri sidosryhmien kanssa, esim.:
    - Vaatimusmäärittelyprosessin jäsentäminen
    - Keskustelut järjestelmäylläpitäjien kanssa
  - Tarkemmat kuvaukset toimivat ohjeena järjestelmän tarkemmassa suunnittelussa ja ylläpitovaiheen aikaisessa laajentamisessa
- Arkkitehtuurikuvaus ei suinkaan ole pelkkä kuva: mm. komponenttien vastuu tulee tarkentaa sekä niiden väliset rajapinnat määritellä
  - Jos näin ei tehdä, kasvaa riski sille että arkkitehtuuria ei noudateta
  - Hyödyllinen kuvaus myös perustelee tehtyjä arkkitehtuurisia valintoja

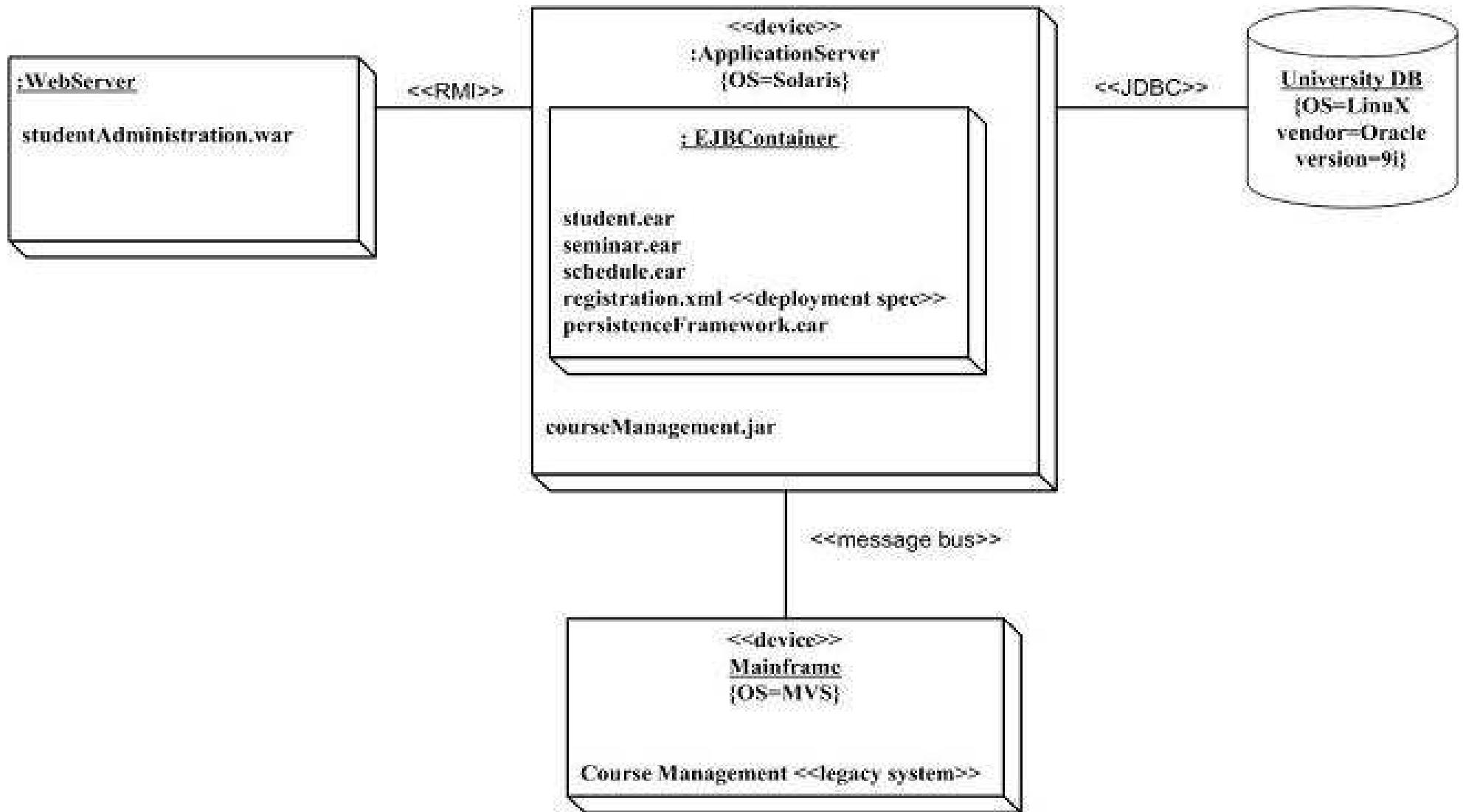
# UML komponenttiakaavio

- <http://www.agilemodeling.com/artifacts/componentDiagram.htm>



# UML:n sijoittelukaavio

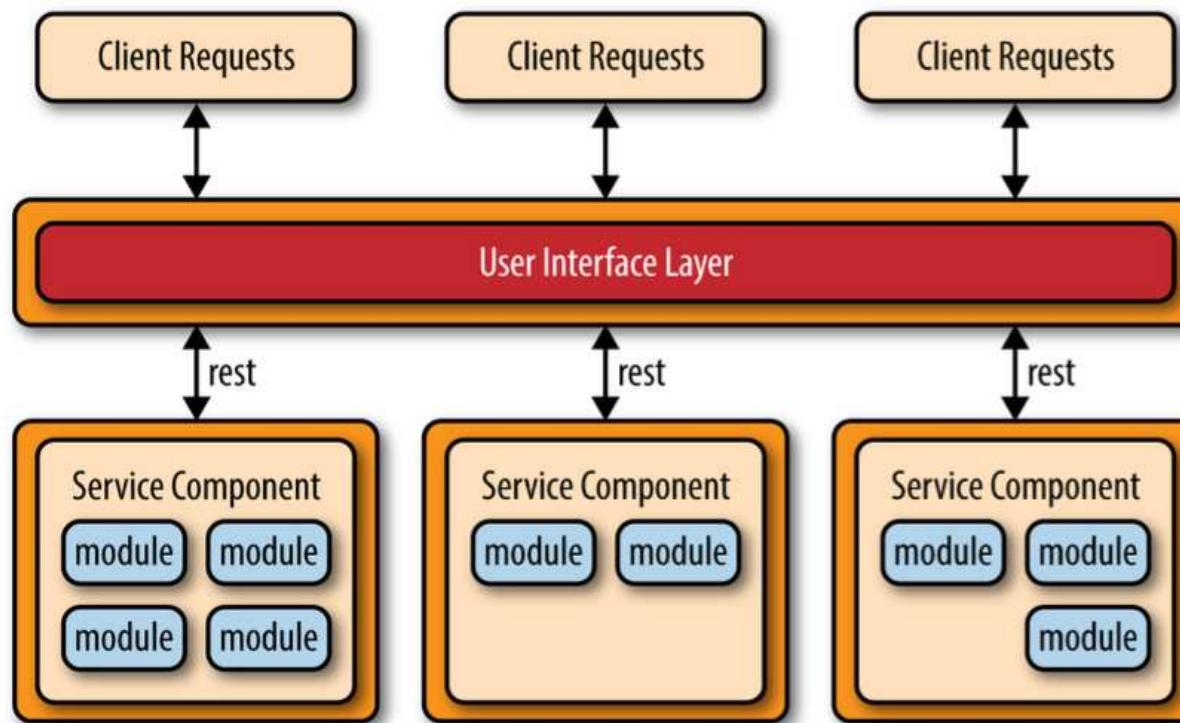
- <http://www.agilemodeling.com/artifacts/deploymentDiagram.htm>



- UML:n komponentti- ja sijoittelukaavio ovat jossain määrin käyttökelpoisia mutta melko harvoin käytännössä käytettyjä

# Hieman lisää arkkitehtuurimalleista

- Tarkastellaan vielä hieman paria arkkitehtuurimallia
- Muutama kalvo sitten todettiin että kerrosarkkitehtuuri saattaa johtaa massiivisiin monoliittisiin sovelluksiin, joita on lopulta vaikea laajentaa ja joiden skaalaaminen suurille käyttäjämääritteille voi muodostua ongelmaksi
- Viime aikoina nopeasti yleistynyt **mikropalvelumalli** (microservices) pyrkii vastaamaan näihin haasteisiin koostamalla sovelluksen useista (jopa sadoista) pienistä verkossa toimivista autonomisista palveluista jotka keskenään verkon yli kommunikoiden toteuttavat järjestelmän toiminnallisuuden



# Mikropalvelut

- Mikropalveluihin perustuvassa sovelluksessa yksittäisistä palveluista pyritään tekemään mahdollisimman riippumattomia
  - Palvelut eivät esim. käytä yhteistä tietokantaa
  - Palvelut on toteutettu omissa koodiprojekteissaan ja ne eivät jaa koodia
  - Palvelut eivät kutsu toistensa metodeja vaan ne keskustelevat keskenään verkon välityksellä
- Mikropalveluiden on tarkoitus olla pieniä ja huolehtia vain "yhdestä asiasta"
  - Kun järjestelmään lisätään toiminnallisuutta, se yleensä tarkoittaa uusien palveluiden toteuttamista tai ainoastaan joidenkin palveluiden laajentamista
  - Sovelluksen laajentaminen voi olla helpompaa kuin kerrosarkkitehtuurissa
- Mikropalveluja hyödyntävä sovellusta voi olla helpompi skaalata
  - suorituskyvyn pullonkaulan aiheuttavia mikropalveluja voidaan suorittaa useita rinnakkain

# Mikropalvelut

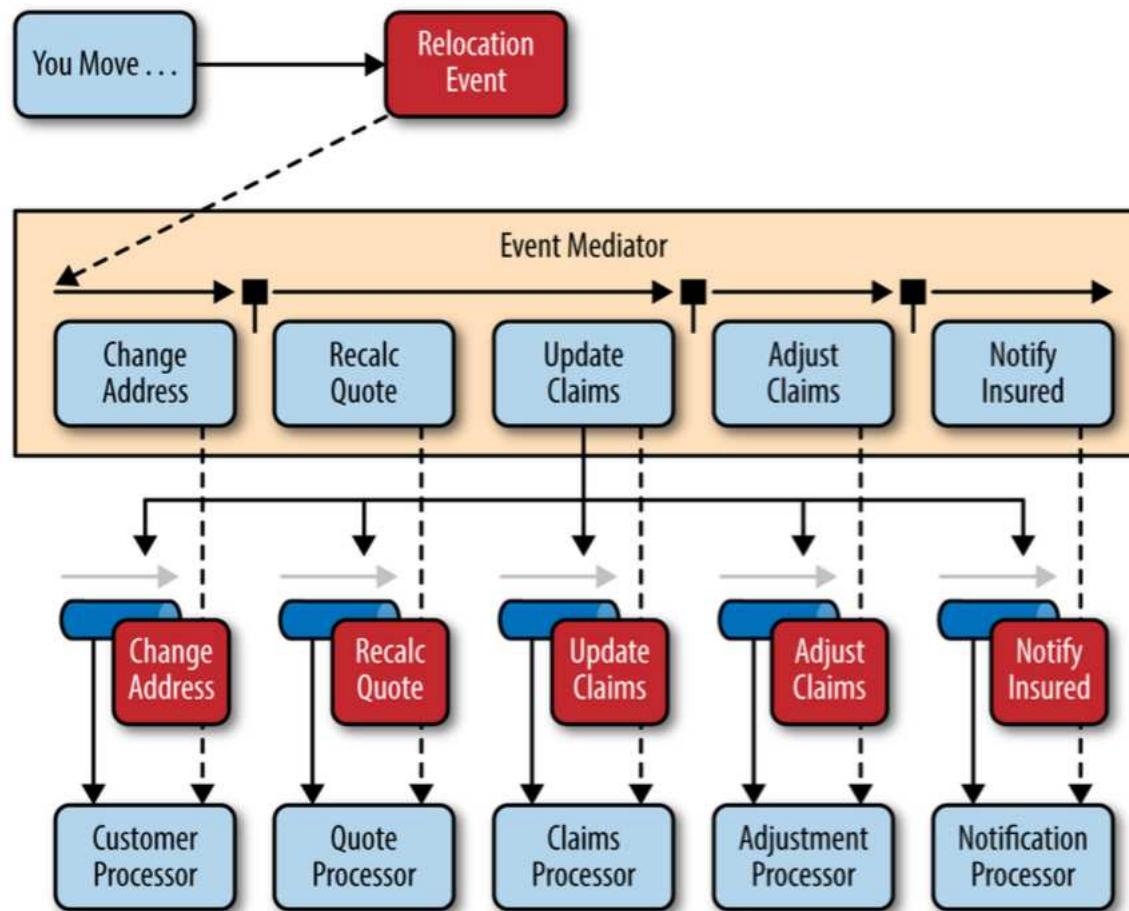
- Mikropalveluiden käyttö mahdollistaa sen, että sovellus voidaan helposti koodata ”monella kielellä”, toisin kuin monoliittisissa projekteissa, mikään ei edellytä, että kaikki mikropalvelut olisi toteutettu samalla kielellä
- Sovelluksen jakaminen järkeviin mikropalveluihin on haastavaa
- Kymmenistä tai jopa sadoista mikropalveluista koostuvan ohjelmiston operoiminen eli ”käynnistäminen” tuotantopalvelimilla on haastavaa ja vaatii pitkälle menevää automatisointia
  - Sama koskee sovelluskehitysympäristöä ja jatkuvaa integraatiota
- Mikropalveluiden yhteydessä käytetäänkin paljon ns *kontainereja* eli käytännössä dockeria
  - Kontainerit ovat hieman yksinkertaistaen sanottuna kevyitä virtuaalikoneita, joita voi suorittaa yhdellä palvelimella suuren määrän rinnakkain
  - Jos mikropalvelu on omassa kontainerissaan, vastaa se käytännössä tilannetta, jossa mikropalvelua suoritetaisiin omalla koneellaan
  - Aihe on tärkeä, mutta emme valitettavasti voi mennä siihen tämän kurssin puitteissa ollenkaan...

# Mikropalveluiden kommunikointi

- Mikropalvelut kommunikoivat keskenään verkon välityksellä
- Kommunikointimekanismeja on useita. Yksinkertainen vaihtoehto on käyttää kommunikointiin HTTP-protokollaa, eli samaa mekanismia, jonka avulla web-selaimet keskustelevat palvelimien kanssa
  - Tällöin puhutaan usein että mikropalvelut tarjoavat kommunikointia varten REST-rajapinnan
  - Viikon 4 laskareissa haettiin suorituksiin liittyvää dataa palautusjärjestelmän tarjoamasta REST-rajapinnasta
- Vaihtoehtoinen, huomattavasti joustavampi kommunikeino on ns. viestikanavien käyttö, tällöin voi ajatella, että mikropalveluita on höystetty *event-driven*-arkkitehtuurilla
- Tällöin verkkoon käynnistetään eräänlainen viestinvälityspalvelu, johon muut palvelut voivat lähettilleä tai **Julkaista** (publish) viestejä
  - Viesteillä on tyypillisesti joku aihe (topic) ja sen lisäksi datasisältö
  - Esim: *topic: new\_user, data: ( username: Arto Hellas, age: 21 )*
- Palvelut voivat **Tilata** (subscribe) viestipalvelulta viestit joista ne ovat kiinnostuneita
  - Esim. käyttäjähallinnasta vastaava palvelu tilaa viestit joiden aihe on *new\_user*
- Viestinvälitysjärjestelmä välittää vastaanottamansa viestit kaikille, jotka ovat aiheen tilanneet

# Mikropalveluiden kommunikointi viestien välityksellä

- Kaikki viestien (tai joskus puhutaan myös tapahtumista, *event*) välitys tapahtuu viestinvälityspalvelun (kuvassa *event mediator*) kautta
- Näin mikropalveluista tulee erittäin löyhästi kytkettyjä, ja muutokset yhdessä palvelussa eivät vaikuta mihinkään muualle, niin kauan kuin viestit säilyvät entisen muotoisina



- Viestinvälitykseen perustuvat mikropalvelut eivät ole ilmainen lounas, erityisesti debuggaus voi olla väillä melko haastavaa

Arkkitehtuuri ketterissä menetelmissä

# Arkkitehtuuri ketterissä menetelmissä

- Ketterien menetelmien kantava teema on toimivan, asiakkaalle arvoa tuottavan ohjelmiston nopea toimittaminen (agile manifestin periaatteita):
  - Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
  - Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Ketterät menetelmät suosivat yksinkertaisuutta suunnitteluratkaisuissa
  - Simplicity--the art of maximizing the amount of work not done--is essential.
  - YAGNI eli "you are not going to need it"-periaate
- Arkkitehtuuriin suunnittelu ja dokumentointi on perinteisesti ollut melko pitkäkestoinen, ohjelmoinnin aloittamista edeltävä vaihe
  - BUFD eli Big Up Front Design
- Ketterät menetelmät ja "arkkitehtuurivetoinen" ohjelmistotuotanto ovat siis jossain määrin keskenään ristiriidassa

# Arkkitehtuuri ketterissä menetelmissä

- Ketterien menetelmien yhteydessä puhutaan *inkrementaalisesta tai evolutiivisesta suunnittelusta ja arkkitehtuurista*
- Arkkitehtuuri mietitään riittävällä tasolla projektin alussa
- Jotkut projektit alkavat ns. nollasprintillä ja alustava arkkitehtuuri määritellään tällöin
  - Scrumin varhaisissa artikkeleissa puhuttiin "pre game"-vaiheesta jolloin mm. alustava arkkitehtuuri luodaan
  - Sittemmin koko käsite on hävinnyt Scrumista ja Ken Schwaber (Scrumin kehittäjä) jopa eksplisiittisesti kielää ja tyrmää koko "nollasprintin" olemassaolon: <http://www.scrum.org/assessmentdiscussion/post/1317787>
- Ohjelmiston "lopullinen" arkkitehtuuri muodostuu iteraatio iteraatiolta samalla kun ohjelmaan toteutetaan uutta toiminnallisuutta
  - Esim. kerrosarkkitehtuurin mukaista sovellusta ei rakenneta "kerros kerrallaan"
  - Jokaisessa iteraatiossa tehdään pieni pala jokaista kerrostaa, sen verran kuin iteraation toiminnallisuksien toteuttaminen edellyttää
  - <http://msdn.microsoft.com/en-us/architecture/ff476940>

# Walking skeleton

- Yleinen lähestymistapa inkrementaaliseen arkkitehtuuriin on *walking skeletonin* käyttö
  - A Walking Skeleton is a **tiny implementation of the system that performs a small end-to-end function**. It need not use the final architecture, **but it should link together the main architectural components**. The architecture and the functionality can then evolve in parallel.
- What constitutes a walking skeleton varies with the system being designed
  - For a layered architecture system, it is a working connection between all the layers
- The walking skeleton is not complete or robust (it only walks, pardon the phrase), and it is missing the flesh of the application functionality. Incrementally, over time, the infrastructure will be completed and full functionality will be added
- A walking skeleton, is permanent code, built with production coding habits, regression tests, and is intended to grow with the system
- Eli tarkoitus on toteuttaa arkkitehtuurin rungon sisältävä Walking skeleton jo ensimmäisessä sprintissä ja kasvattaa se pikkuhiljaan projektin edetessä
- Katso lisää esim <http://alistair.cockburn.us/Walking+skeleton>

# Arkkitehtuuri ketterissä menetelmissä

- Perinteisesti arkkitehtuurista on vastannut ohjelmistoarkkitehti ja ohjelmoijat ovat olleet velvoitettuja noudattamaan arkkitehtuuria
- Ketterissä menetelmissä ei suosita erillistä arkkitehdin roolia, esim. Scrum käyttää kaikista ryhmän jäsenistä nimikettä *developer*
- Ketterien menetelmien ideaali on, että kehitystiimi luo arkkitehtuurin yhdessä, tämä on myös yksi agile manifestin periaatteista:
  - The best architectures, requirements, and designs emerge from self-organizing teams.
- Arkkitehtuuri on siis koodin tapaan tiimin *yhteisomistama*, tästä on muutamia etuja
  - Kehittäjät sitoutuvat paremmin arkkitehtuurin noudattamiseen kuin "norsunluutornissa" olevan tiimin ulkopuolisen arkkitehdin määrittelemään arkkitehtuuriin
  - Arkkitehtuurin dokumentointi voi olla kevyt ja informaali (esim. valkotaululle piirretty) sillä tiimi tuntee joka tapauksessa arkkitehtuurin hengen ja pystyy sitä noudattamaan

# Inkrementaalinen arkkitehtuuri

- Ketterissä menetelmissä oletuksena on, että parasta mahdollista arkkitehtuuria ei pystytä suunnittelemaan projektin alussa, kun vaatimuksia, toimintaympäristöä ja toteutusteknologiaita ei vielä tunneta
  - Jo tehtyjä arkkitehtonisia ratkaisuja muutetaan tarvittaessa
- Eli kuten vaatimusmäärittelyn suhteen, myös arkkitehtuurin suunnittelussa ketterät menetelmät pyrkii välttämään liian aikaisin tehtävää ja myöhemmin todennäköisesti turhaksi osoittautuvaa työtä
- Inkrementaalinen lähestymistapa arkkitehtuurin muodostamiseen edellyttää koodilta hyvää sisäistä laattua ja toteuttajilta kurinalaisuutta
- Martin Fowler <http://martinfowler.com/articles/designDead.html>:
  - Essentially evolutionary design means that the design of the system grows as the system is implemented. Design is part of the programming processes and as the program evolves the design changes.
  - **In its common usage, evolutionary design is a disaster.** The design ends up being the aggregation of a bunch of ad-hoc tactical decisions, each of which makes the code harder to alter
- Seuraavaksi siirrymme käsittelemään oliosuunnittelua

Ohjelmistotuotanto

Luento 8

4.4.

# Oliosuunnittelu

# Oliosuunnittelu

- Käytettäessä ohjelmiston toteutukseen olio-ohjelmostikieltä, on suunnitteluvaiheen tarkoituksesta löytää sellaiset oliot, jotka pystyvät yhteistoiminnallaan toteuttamaan järjestelmän vaatimuksen
- Oliosuunnittelua ohjaa ohjelmistolle suunniteltu arkkitehtuuri
- Ohjelman ylläpidettävyyden kannalta on suunnittelussa hyvä noudattaa "ikiaikaisia" hyvän suunnittelun käytänteitä
  - Ketterissä menetelmissä tämä on erityisen tärkeää, sillä jos ohjelman rakenne pääsee rapistumaan, on ohjelmaa vaikea laajentaa jokaisen sprintin aikana
- Ohjelmiston suunnittelun on olemassa useita erilaisia menetelmiä, mikään niistä ei kuitenkaan ole vakiintunut
- Ohjelmistosuunnittelu onkin "enemmän taidetta kuin tiedettä", kokemus ja hyvien käytänteiden opiskelu toki auttaa
- Erityisesti ketterissä menetelmissä tarkka oliosuunnittelu tapahtuu yleensä ohjelmoinnin yhteydessä

# Oliosuunnittelu

- Emme keskity tällä kurssilla mihinkään yksittäiseen oliosuunnittelumenetelmään
- Sen sijaan tutustumme muutamiin tärkeisiin menetelmäriippumattomiin teemoihin:
- Laajennettavuuden ja ylläpidettävyyden suhteen laadukkaan koodin/olioluunnittelun tunnusmerkkeihin ja *laatuatribuutteihin*
  - kapselointi, koheesio, riippuvuuksien vähäisyys, toisteettomuus, selkeys, testattavuus
- ja niitä tukeviin ”ikiaikaisiin” hyvän suunnittelun periaatteisiin
- *Koodinhajuihin* eli merkkeihin siitä että suunnittelussa ei kaikki ole kunnossa
- *Refaktoriointiin* eli koodin rajapinnan ennalleen jättäään rakenteen parantamiseen
- Erelaisissa tilanteissa toimiviksi todettuihin geneerisiä suunnitteluratkaisuja dokumentoiviin *suunnittelumalleihin*
  - Olemme jo nähneet muutamia suunnittelumalleja, ainakin seuraavat: dependency injection, singleton, data access object

# Helposti ylläpidettävän koodin tunnusmerkit

- Ylläpidettävyyden ja laajennettavuuden kannalta tärkeitä seikkoja
  - Koodin tulee olla luettavuudeltaan selkeää, eli koodin tulee kertoa esim. nimennällään mahdollisimman selkeästi mitä koodi tekee, eli tuoda esiin koodin alla oleva "design"
  - Yhtä paikkaa pitää pystyä muuttamaan siten, ettei muutoksesta aiheudu sivuvaikutuksia sellaisiin kohtiin koodia, jota muuttaja ei pysty ennakoimaan
  - Jos ohjelmaan tulee tehdä laajennus tai bugikorjaus, tulee olla helppo selvittää miin kohtaan koodia muutos tulee tehdä
  - Jos ohjelmasta muutetaan "yhtä asiaa", tulee kaikkien muutosten tapahtua vain yhteen kohtaan koodia (metodiin tai luokkaan)
  - Muutosten ja laajennusten jälkeen tulee olla helposti tarkastettavissa ettei muutos aiheuta sivuvaikutuksia muualle järjestelmään
- Näin määritelty koodin *sisäinen laatu* on erityisen tärkeää ketterissä menetelmissä, joissa koodia laajennetaan iteraatio iteraatiolta
- Jos koodin sisäiseen laatuun ei kiinnitetä huomiota, on väistämätöntä että pidemmässä projektissa kehitystiimin velositeetti alkaa tippua ja eteneminen alkaa vaikeutua iteraatio iteraatiolta
  - Koodin sisäinen laatu on siis usein myös asiakkaan etu

# Koodin laatuattribuutteja

- Edellä lueteltuihin hyvän koodin tunnusmerkkeihin päästään kiinnittämällä huomio seuraaviin *laatuattribuutteihin*
  - Kapselointi
  - Koheesio
  - Riippuvuuksien vähäisyys
  - Toisteettomuus
  - Testattavuus
  - Selkeys
- Tutkitaan nyt näitä laatuattribuutteja sekä periaatteita, joita noudattaen on mahdollista kirjoittaa koodia, joka on näiden mittarien mukaan laadukasta
- **HUOM** seuraaviin kalvoihin liittyvät koodiesimerkit löytyvät osoitteesta <https://github.com/mluukkai/ohtu2017/tree/master/web/luento8.md>
- Lue koodiesimerkkejä sitä mukaa kun kalvoissa viitataan niihin

# Kapselointi

- Ohjelmointikursseilla on määritelty kapselointi seuraavasti
  - "Tapaa ohjelmoida olion toteutuksen yksityiskohdat luokkamäärittelyn sisään – piiloon olion käyttäjältä – kutsutaan kapseloinniksi. Olion käyttäjän ei tarvitse tietää mitään olioiden sisäisestä toiminnasta. Eikä hän itse asiassa edes saa siitä mitään tietää vaikka kuinka haluaisi! "
- Aloitteleva ohjelmoija assosioi kapseloinnin yleensä seuraavaan periaatteeseen:
  - Oliomuuttujat tulee määritellä privaateiksi ja niille tulee tehdä tarvittaessa setterit ja getterit
- Tämä on kuitenkin aika kapea näkökulma kapselointiin
- Jatkossa näemme esimerkkejä monista kapseloinnin muista muodoista. Kapseloinnin kohde voi olla mm.
  - Käytettävän olion tyyppi, algoritmi, olioiden luomistapa, käytettävän komponentin rakenne
- Monissa suunnittelumalleissa on kyse juuri eritasoisten asioiden kapseloinnista

# Koheesio

- Koheesiolla tarkoitetaan sitä, kuinka pitkälle metodissa, luokassa tai komponentissa oleva ohjelmakoodi on keskittynyt tietyn toiminnallisuuden toteuttamiseen
- Hyväntä asiana pidetään mahdollisimman korkeaa koheesion astetta
- Koheesioon tulee siis pyrkiä kaikilla ohjelman tasoilla, metodeissa, luokissa, komponenteissa ja jopa muuttujissa
- **Metoditason koheesiolla** pyrkimyksenä että metodi tekee itse vain yhden asian
- Metoditason koheesiota ilmentävä Kent Beckin "Composed method"-suunnittelumalli ohjeistaa seuraavasti
- *The composed method pattern* defines three key statements:
  - Divide your programs into methods that perform one identifiable task.
  - Keep all the operations in a method at the same level of abstraction.
  - This will naturally result in programs with many small methods, each a few lines long.
- <http://www.ibm.com/developerworks/java/library/j-eaed4/index.html>

# Koheesio ja Single responsibility -periaate

- Metoditason koheesioon päästään jakamalla "koheesiiton" metodi useisiin metodeihin, joita alkuperäinen metodi kutsuu
  - Alkuperäinen metodi alkaa toimia korkeammalla abstraktiotasolla, koodinoiden kutsumiensa yhteen asiaan keskittyvien metodien suoritusta
  - esimerkki <https://github.com/mluukkai/ohtu2017/tree/master/web/luento8.md> kohdassa "koheesio metoditasolla"
- **Luokkataslon koheesiolla** pyrkimyksenä on, että luokan **vastuulla** on vain yksi asia
- Ohjelmistotekniikan menetelmistä tuttu **Single Responsibility** (SRP) -periaate tarkoittaa oikeastaan täysin samaa
  - [www.objectmentor.com/resources/articles/srp.pdf](http://www.objectmentor.com/resources/articles/srp.pdf) (linkki rikki)
  - Uncle Bob tarkentaa yhden vastuun määritelmää siten, että *luokalla on yksi vastuu jos sillä on vain yksi syy muuttua*
- Esimerkkejä  
<https://github.com/mluukkai/ohtu2017/tree/master/web/luento8.md> kohdassa "single responsibility -periaate eli koheesio luokatasolla"
- Vastakohta SRP:tä noudattavalle luokalle on *jumalaluokka/olio*
  - <http://blog.decayingcode.com/post/anti-pattern-god-object.aspx> (linkki rikki)

# Riippuvuuksien vähäisyys

- Single responsibility -periaatteen hengessä tehty ohjelma koostuu suuresta määrästä oliota, joilla on suuri määrä pieniä metodeja
- Olioiden on siis oltava vuorovaikutuksessa toistensa kanssa saadakseen toteutettua ohjelman toiminnallisuuden
- **Riippuvuuksien vähäisyyden** (engl. low coupling) periaate pyrkii eliminoimaan luokkien ja olioiden välisiä riippuvuuksia
- Koska olioita on paljon, tulee riippuvuuksia pakostakin, miten riippuvuudet sitten saadaan eliminoitua?
- Ideana on eliminoida *tarpeettomat* riippuvuudet ja välttää riippuvuuksia *konkreettisiin* asioihin
  - Riippuvuuden kannattaa kohdistua asiaan joka ei muutu herkästi, eli joko rajapintaan tai abstraktiin luokkaan
  - Sama idea kulkee parillakin eri nimellä
    - **Program to an interface, not to an Implementation**
      - <http://www.artima.com/lejava/articles/designprinciples.html>
    - **Depend on Abstractions, not on concrete implementation**

# Riippuvuuksien vähäisyys

- Konkreettisen riippuvuuden eliminointi voidaan tehdä rajapintojen avulla
  - Olemme tehneet näin kurssilla usein, mm. Verkkokaupan riippuvuus Varastoon, Pankkiin ja Viitegeneraattoriin korvattiin rajapinnoilla
  - Dependency Injection -suunnittelumalli toimi usein apuvälilineenä konkreettisen riippumisen eliminoinnissa
- Osa luokkien välisistä riippuvuuksista on tarpeettomia ja ne kannattaa eliminoida muuttamalla luokan vastuta
- Perintä muodostaa riippuvuuden perivän ja perittävän luokan välille, tämä voi jossain tapauksissa olla ongelmallista
- Yksi oliosuunnittelun kulmakivi on periaate **Favour composition over inheritance eli suosi yhteistoiminnassa toimivia oliota perinnän sijaan**
  - <http://www.artima.com/lejava/articles/designprinciples4.html>
  - Perinnällä on paikkansa, mutta sitä tulee käyttää harkiten
- ks. <https://github.com/mluukkai/ohtu2017/tree/master/web/luento8.md> kohta "Favour composition over inheritance eli milloin ei kannata periä"

# Suunnittelumalli: Static factory

- Tili-esimerkissä käytetty **Static factory method** on yksinkertaisin erilaisista tehdas-suunnittelumallin varianteista
- Luokkaan tehdään staattinen tehdasmetodi tai metodeja, jotka käyttävät konstruktoria ja luovat luokan ilmentymät
  - Konstruktorin suora käyttö usein estetään määrittelemällä konstruktori privateksi
- Tehdasmetodin avulla voidaan piilottaa olion luomiseen liittyviä yksityiskohtia
  - Esimerkissä Korko-rajapinnan toteuttavien olioiden luominen ja jopa olemassaolo oli tehdasmetodin avulla piilotettu tilin käyttäjältä
- Tehdasmetodin avulla voidaan myös piilottaa käyttäjältä luodun olion todellinen luokka
  - Esimerkissä näin tehtiin määräaikaistilin suhteen
- Staattinen tehdasmetodi ei ole testauksen kannalta erityisen hyvä ratkaisu, esimerkissämme olisi vaikea luoda tili, jolle annetaan Korko-rajapinnan toteuttama mock-olio
  - nyt se tosin onnistuu koska konstruktoria ei ole täysin piilotettu

# Suunnittelumalli: Strategy

- Esimerkissä tilien koron laskenta hoidettiin Strategy-suunnittelumallilla
  - [http://sourcemaking.com/design\\_patterns/strategy](http://sourcemaking.com/design_patterns/strategy)
  - <http://www.odesign.com/strategy-pattern.html>
  - <http://www.netobjectives.com/PatternRepository/index.php?title=TheStrategyPattern>
- Strategyn avulla voidaan hoitaa tilanne, jossa eri olioiden käyttäytyminen on muuten sama mutta tietyissä kohdissa on käytössä eri "algoritmi"
  - Esimerkissämme tämä "algoritmi" oli korkoprosentin määritys
- Sama tilanne voidaan hoitaa usein myös perinnän avulla käyttämättä erillisiä olioita, strategy kuitenkin mahdollistaa huomattavasti dynaamisemman ratkaisun, sillä strategia-olioa voi vaihtaa ajoikana
- Strategyn käyttö ilmentää hienosti "favour composition over inheritance"-periaatetta
- Usein strategiaa käytetään korvaamaan ohjelmassa oleva ikävä if- tai switch-hässäkkä
  - ks. <https://github.com/mluukkai/ohtu2017/tree/master/web/luento8.md> kohta "laskin ilman iffejä"

# Suunnittelumalli: command eli komento

- Laskin-esimerkissä päädyttiin eristämään jokaiseen erilliseen laskuoperaatioon liittyvä toiminta omaksi oliokseen
- Kaikki operaatiot toteuttavat yksinkertaisen rajapinnan, jolla on ainoastaan metodi **public void suorita()**
- Siirryttiin pelkän algoritmin kapseloivista strategiaolioista koko komennon suorituksen kapseloiviin olioihin eli **command-suunnittelumalliin**
  - <http://www.odesign.com/command-pattern.html>
  - [http://sourcemaking.com/design\\_patterns/command](http://sourcemaking.com/design_patterns/command)
- Strategian tapaan komento-oliolla saadaan eliminoitua koodista if-ketjuja
  - Esimerkissä komennot luotiin tehdasmetodin tarjoavan olion avulla, if:it piilotettiin tehtaan sisälle
- Komento-olioiden suorita-metodi suoritettiin esimerkissä välittömästi, näin ei välttämättä ole, komentoja voitaisiin laittaa esim. jonoon
- Joskus komento-olioilla on myös undo-operaatio
  - Esim. editorien undo- ja redo-toiminnallisuus toteutetaan säilyttämällä komento-olioita jonoissa

# Suunnittelumalli: template method

- Laskin-esimerkkiin oli pilotettu myös suunnittelumalli Template Method
  - <http://www.odesign.com/template-method-pattern.html>
  - <http://www.netobjectives.com/PatternRepository/index.php?title=TheTemplateMethodPattern>
- Summa- ja Tulo-komentojen suoritus on oleellisesti samanlainen:
  - Lue luku1 käyttäjältä
  - Lue luku2 käyttäjältä
  - *Laske operaation tulos*
  - Tulosta operaation tulos
- Ainoastaan kolmas vaihe eli *operaation tuloksen laskeminen* eroaa
- Asia hoidettiin tekemällä abstrakti yliluokka, joka sisältää metodin *suorita()* joka toteuttaa koko komennon suorituslogiikan
- Suorituslogiikan vaihtuva osa eli operaation laskun tulos on määritelty abstraktina metodina *laske()* jota *suorita()* kutsuu
- Konkreettiset toteutukset Summa ja Tulo ylikirjoittavat abstraktin metodin *laske()*

```
public abstract class KaksiparametrisenLaskuoperaatio implements Komento {
```

```
// ...
```

```
@Override
```

```
public void suorita() {
```

```
    io.print("luku 1: ");
```

```
    int luku1 = io.nextInt();
```

```
    io.print("luku 2: ");
```

```
    int luku2 = io.nextInt();
```

```
    int tulos = laske();
```

```
    io.print("vastaus: "+tulos);
```

```
}
```

```
protected abstract int laske();
```

```
}
```

# Template method

```
public class Summa extends KaksiparametrisenLaskuoperaatio {  
    @Override  
    protected int laske() {  
        return luku1+luku2;  
    }  
}
```

- Abstraktin luokan määrittelemä suorita() on **template-metodi**, joka määrittelee suorituksen siten, että osan suorituksen konkreettinen toteutus on abstraktissa metodissa, jonka aliluokat ylikirjoittavat
- Template-metodin avulla siis saadaan määriteltyä "geneerinen algoritmikunko", jota voidaan aliluokissa erikoistaa sopivalla tavalla
- Strategy-suunnittelumalli on osittain samaa sukua Template-metodin kanssa, siinä kokonainen algoritmi tai algoritmin osa korvataan erillisessä luokassa toteutetulla toteutuksella
- Strategioita voidaan vaihtaa ajonaikana, template-metodissa olio toimii samalla tavalla koko elinaikansa

# Komposiitti ja proxy

- <https://github.com/mluukkai/ohtu2017/tree/master/web/luento8.md> olevat esimerkit Komposiitti ja Proxy demonstroivat jälleen kahta suunnittelumallia
- Komposiitti on tapa järjestää puu/rekursiomaisesti rakentuvia samankaltaisesti ulospäin käytäytyviä olioita
  - [http://sourcemaking.com/design\\_patterns/composite](http://sourcemaking.com/design_patterns/composite)
- Komposiitti kapseloi yhden rajapinnan taakse joko yksittäisen olion (kuten esimerkissä erotinelementin) tai mielivaltaisen monimutkaisen elementeistä koostuvan puumaisen rakenteen
  - Käyttäjän eli esimerkissämme dokumentin kannalta yksittäisen elementin sisäisellä rakenteella ei ole merkitystä, elementti osaa tulostaa itsensä ja se riittää dokumentille
- Joskus käytettävä olio voi olla luonteeltaan sellainen, että olion itsensä käyttö on raskasta ja usein riittää että oliota edustaa joku muu siihen asti kunnes oliota itseään todellakin tarvitaan
- Tällaisissa tilanteissa proxy-suunnittelumalli tuo ratkaisun
  - [http://sourcemaking.com/design\\_patterns/proxy](http://sourcemaking.com/design_patterns/proxy)
  - Esimerkissämme web-elementti toteutettiin proxyn avulla

# Lisää koodin laatuattribuutteja: DRY

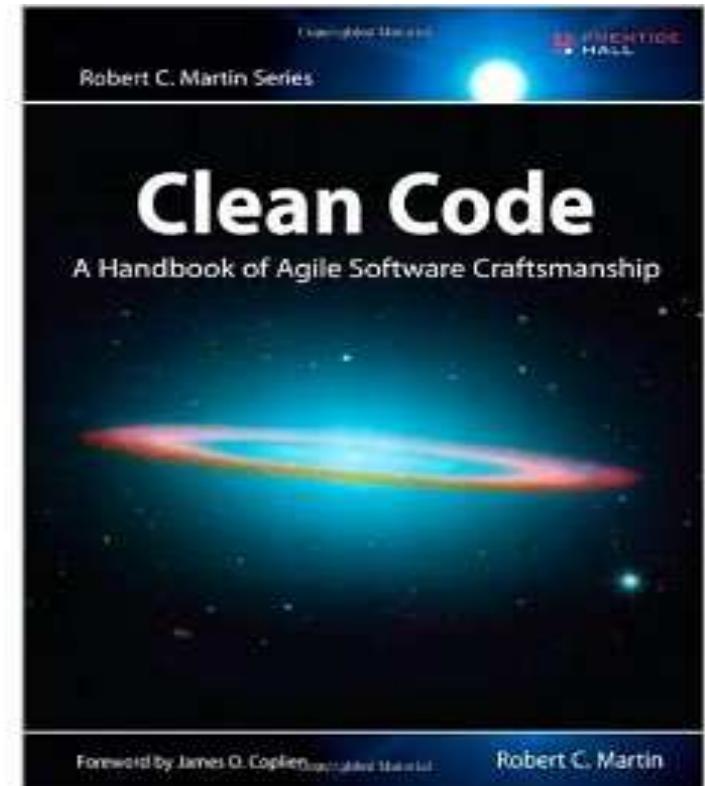
- Käsittelimme koodin laatuattribuuteista kapselointia, koheesiota ja riippuvuuksien vähäisyyttä, seuraavana vuorossa **redundanssi** eli **toisteisuus**
- Aloittelevaa ohjelmoijaa pelotellaan toisteisuuden vaaroista uran ensiaskelista alkaen: **älä copypastaa koodia!**
- Alan piireissä toisteisuudesta varoittava periaate kulkee nimellä **DRY, don't repeat yourself**
  - *"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."*
  - <http://c2.com/cgi/wiki?DontRepeatYourself>
  - DRY-periaate menee oikeastaan vielä paljon pelkkää koodissa olevaa toistoa eliminointia pidemmälle
- Ilmeisin toiston muoto koodissa on juuri copypaste ja se onkin helppo eliminoida esim. metodien avulla
- Kaikki toisteisuus ei ole yhtä ilmeistä ja monissa suunnittelumalleissa on kyse juuri hienovaraisempien toisteisuuden muotojen eliminoinnista
  - ks. <https://github.com/mluukkai/ohtu2017/tree/master/web/luento8.md> kohta "Koodissa olevan epätriviaalin copypasten poistaminen Strategy-patternin avulla, Java 8:a hyödyntävä versio"

# Lisää laatuatribuutteja

- Testattavuus
  - Hyvä koodi on helppo testata kattavasti yksikkötestein
  - Helppo testattavuus seuraa yleensä siitä, että koodi koostuu löyhästi kytketyistä, selkeän vastuun omaavista olioista ja ei sisällä toisteisuutta
  - Kääntäen, jos koodin kattava testaaminen on vaikeaa, on se usein seurausta siitä, että olioiden vastuut eivät ole selkeät, olioilla on liikaa riippuvuuksia ja toisteisuutta on paljon
  - Olemme pyrkineet jo ensimmäiseltä viikolta asti koodin hyvään testattavuuteen esim. purkamalla riippuvuuksia rajapintojen ja dependency injectionin avulla
- Koodin selkeys ja luettavuus
  - Suuri osa "ohjelointiin" kuluvasta ajasta kuluu olemassaolevan koodin (joko kehittäjän itsensä tai jonkun muun kirjoittaman) lukemiseen
    - ks. esim. <http://www.architexa.com/solutions/challenge>

# Koodin luettavuus

- Perinteisesti ohjelmakoodin on ajateltu olevan väkisinkin kryptistä ja vaikeasti luettavaa
  - Esim. c-kielessä on tapana ollut kirjoittaa todella tiivistä koodia, jossa yhdellä rivillä on ollut tarkoitus tehdä mahdollisimman monta asiaa
  - Metodikutsuja on vältetty tehokkuussyyistä
  - ...
- Ajat ovat muuttuneet ja nykytrendin mukaista on pyrkiä kirjoittamaan koodia, joka nimennällään ja muodollaan ilmaisee mahdollisimman hyvin sen mitä koodi tekee
- Selkeän nimennän lisäksi muita luettavan eli "puhtaan" koodin (clean code) tunnusmerkkejä ovat jo monet meille entuudestaan tutut asiat
  - [www.planetgeek.ch/wp-content/uploads/2011/02/Clean-Code-Cheat-Sheet-V1.3.pdf](http://www.planetgeek.ch/wp-content/uploads/2011/02/Clean-Code-Cheat-Sheet-V1.3.pdf)



Ohjelmistotuotanto

Luento 9

10.4.2017

# Ohjelmiston suunnittelu, testausta

- Suunnittelun ajatellaan yleensä jakautuvan kahteen vaiheeseen:
  - **Arkkitehtuurisuunnittelu**
    - Ohjelman rakenne karkealla tasolla
    - Mistä suuremmista rakennekomponenteista ohjelma koostuu?
    - Miten komponentit yhdistetään, eli komponenttien väliset rajapinnat
    - Useimmiten ohjelma noudattaa jotain hyvin tunnettua **arkkitehtuurista mallia**, kuten kerrosarkkitehtuuria, MVC:tä, mikropalveluarkkitehtuuria jne
  - **Oliosuunnittelu**
    - Yksittäisten komponenttien suunnittelu
    - Tapahtuu usein ohjelmoinnin yhteydessä
- Suunnittelun ajoittuminen riippuu käytettävästä tuotantoprosessista:
  - Ketterissä menetelmissä suunnittelua tehdään tarvittava määrä jokaisessa iteraatiossa (**inkrementaalinen design ja arkkitehtuuri**), tarkkaa suunnitteludokumenttia ei yleensä ole
  - Jos ei olla tekemässä ”kertakäyttökoodia” tai ottamassa tietoisesti teknistä velkaa, on oliosuunnittelussa tärkeää pitää mielessä ohjelman ylläpidettävyys ja laajennettavuus

# Helposti ylläpidettävän koodin tunnusmerkit

- Ylläpidettävyyden ja laajennettavuuden kannalta tärkeitä seikkoja
  - Koodin tulee olla luettavuudeltaan selkeää, eli koodin tulee kertoa esim. nimennällään mahdollisimman selkeästi mitä koodi tekee, eli tuoda esiin koodin alla oleva "design"
  - Yhtä paikkaa pitää pystyä muuttamaan siten, ettei muutoksesta aiheudu sivuvaikutuksia sellaisiin kohtiin koodia, jota muuttaja ei pysty ennakoimaan
  - Jos ohjelmaan tulee tehdä laajennus tai bugikorjaus, tulee olla helppo selvittää miin kohtaan koodia muutos tulee tehdä
  - Jos ohjelmasta muutetaan "yhtä asiaa", tulee kaikkien muutosten tapahtua vain yhteen kohtaan koodia (metodiin tai luokkaan)
  - Muutosten ja laajennusten jälkeen tulee olla helposti tarkastettavissa ettei muutos aiheuta sivuvaikutuksia muualle järjestelmään
- Näin määritelty koodin *sisäinen laatu* on erityisen tärkeää ketterissä menetelmissä, joissa koodia laajennetaan iteraatio iteraatiolta
- Jos koodin sisäiseen laatuun ei kiinnitetä huomiota, on väistämätöntä että pidemmässä projektissa kehitystiimin velositeetti alkaa tippua ja eteneminen alkaa vaikeutua iteraatio iteraatiolta
  - Koodin sisäinen laatu on siis usein myös asiakkaan etu

# Koodin laatuattribuutteja

- Edellä lueteltuihin hyvän koodin tunnusmerkkeihin päästään kiinnittämällä huomio seuraaviin *laatuattribuutteihin*
  - Kapselointi
  - Koheesio
  - Riippuvuuksien vähäisyys
  - Toisteettomuus
  - Testattavuus
  - Selkeys
- Jatketaan laatuattribuutteihin ja niitä tukeviin "ikiaikaisiin" hyvän suunnittelun periaatteisiin sekä erilaisissa tilanteissa toimiviksi todettuihin geneerisiä suunnitteluratkaisuja dokumentoiviin *suunnittelumalleihin* tutustumista
- Viime viikolla käsiteltyjä design patterneja
  - Strategy, Factory, Command, Template method, Composite, Proxy

Lisää suunnittelumalleja

# Olion rikastaminen dekoraattorilla

- Joskus eteen tulee tarve lisätä olioon joitain ekstraominaisuuksia, pitäen kuitenkin olio sellaisena, että sitä käyttäviin ohjelmanosiin ei tarvitse tehdä muutoksia
- **Dekoraattori** (decorator) -suunnittelumalli tuo avun
  - [http://sourcemaking.com/design\\_patterns/Decorator](http://sourcemaking.com/design_patterns/Decorator)
- Dekoraattorissa muodostetaan "rikastettu" olio, jolla on täysin sama rajapinta kuin oliolla, johon lisäominaisuksia halutaan
  - Dekoraattoriolio yleensä delegoi varsinaisen tehtävän, eli olion vanhan vastuun suorittamisen alkuperäiselle oliolle
- Katsotaan ensin hieman yksinkertaisempaa tapausta
- ks <https://github.com/mluukkai/ohtu2017/blob/master/web/luento9.md>  
Dekoroitu Random
- Esimerkissä tehdään dekoroitu Random-olio, jonka avulla on mahdollista testata satunnaislukuja käyttäväää ohjelmaa
  - Dekoroitu Random ottaa talteen kaikki arvotut luvut
  - Testissä käytetään dekoroitua versiota normaalinv Randomin sijaan
  - Testi pääsee kysymään dekoroidulta randomilta arvotut numerot

# Dekoroitu pino, pinotehdas ja rakentaja

- Tarkastellaan esimerkkejä Dekoroitu Pino ja Pinotehdas osoitteessa  
<https://github.com/mluukkai/ohtu2017/blob/master/web/luento9.md>
- Saamme dekoraattorin avulla hienosti tehtyä monen eri ominaisuuskombinaation omaavia pinoja
- Dekoroitujen pinojen luominen on monimutkaista, mutta Factoryn avulla saamme peitettyä monimutkaisuuden pinon käyttäjältä
- Factorystä muodostuu kuitenkin ongelma...
- **Rakentaja** (engl builder) -suunnittelumalli kuitenkin ratkaisee ongelman!
- Rakentajassa on kiinnitetty erityinen huomio metodien nimeämiseen:

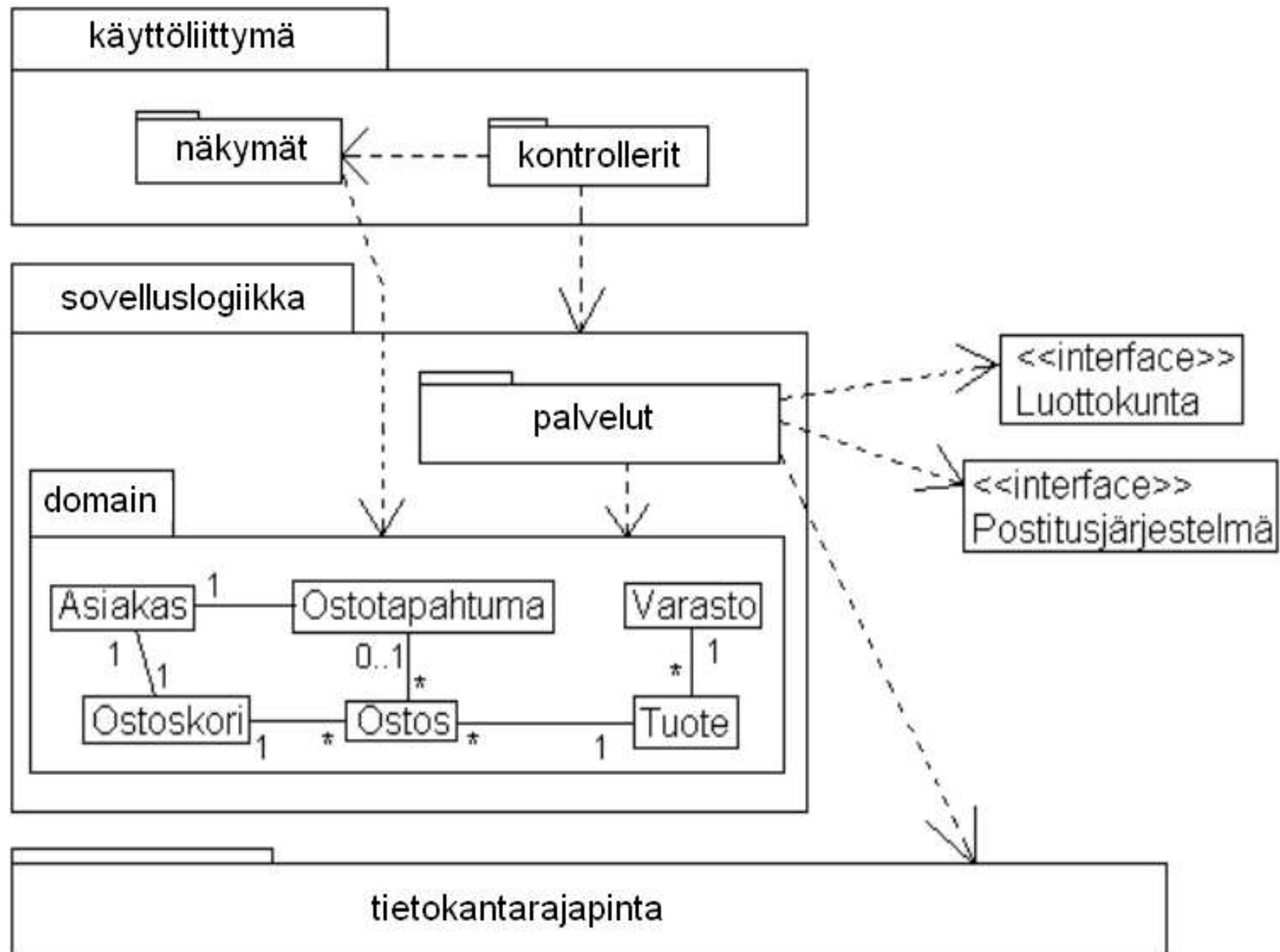
```
Pinorakentaja rakenna = new Pinorakentaja();  
Pino pino = rakenna.kryptattu().prepaid(10).pino();
```
- On haettu mahdollisimman luonnollista kieltyä muistuttavaa luettavuutta
- Muodostettiin **DSL (domain specific language)** pinojen luomiseen
  - <http://martinfowler.com/bliki/FluentInterface.html>
  - <http://www.infoq.com/articles/internal-dsls-java>

# Luokan rajapinnan muuttaminen adapterilla

- Äskentävien käsiteltyjen suunnittelumallien, *dekoraattorin*, *komposiitin* ja *proxyn* yhteinen puoli on, että saman ulkokuoren eli rajapinnan takana voi olla yhä monimutkaisempaa toiminnallisuutta, joka on kuitenkin täysin kapseloitu käyttäjältä
- Tarkastellaan nyt tilannetta, jossa käytettäväissä on luokka, joka oleellisesti ottaen tarjoaa halutun toiminnallisuuden, mutta sen rajapinta on hieman vääränlainen esim. metodien nimien tai parametrien osalta
  - Perintä ei siis sovi ratkaisumenetelmäksi
- Alkuperäistä luokkaa ei kuitenkaan haluta tai voida muuttaa sillä muutos rikkoisi luokan muut käyttäjät
- **Adapteri-suunnittelumalli** sopii tällaisiin tilanteisiin
  - [http://sourcemaking.com/design\\_patterns/adapter](http://sourcemaking.com/design_patterns/adapter)
- Tutkitaan esimerkkiä "adapteri" sivulta  
<https://github.com/mluukkai/ohtu2017/blob/master/web/luento9.md>
  - Pino adaptoidaan sopimaan rajapinnaltaan paremmin uuteen käyttötilanteeseen

# Paluu suuriin linjoihin

- Arkkitehtuurin yhteydessä mainitsimme kerrosarkkitehtuurin, josta esimerkkinä oli Kumpula biershopin arkkitehtuuri
- Kerroksittaisuudessa periaate on sama kuin useiden suunnittelumallien ja hyvän oliosuunnittelussa yleensäkin **kapseloidaan monimutkaisutta ja detaljeja rajapintojen taakse**
- Tarkoituksena ylläpidettävyyden parantaminen ja kompleksisuuden hallinnan helpottaminen
  - Kerroksen N käyttäjää on turha vaivata kerroksen sisäisellä rakenteella
  - Eikä sitä edes kannata paljastaa, koska näin muodostuisi eksplisiittinen riippuvuus käyttäjän ja N:n välille
- Pyrkimys siihen että *kerrokset ovat mahdollisimman korkean koheesioon omaavia*, eli "yhteen asiaan" keskittyvä
  - Käyttöliittymä
  - Tietokantayhteydet
  - Liiketoimintalogiikka
- Kerrokset taas ovat keskenään mahdollisimman *löyhästi kytkettyjä*



# Domain Driven Design

- Viimeaikaisena voimakkaasti nousevana trendinä on käyttää sovelluksen koodin tasolla nimentää, joka vastaa liiketoiminta-alueen eli "bisnesdomainin" terminologiaa
  - Yleisnimike tälle tyylille on Domain Driven Design, DDD
  - ks esim. <http://www.infoq.com/articles/ddd-evolving-architecture>
- Ohjelmiston arkkitehtuurissa on DDD:tä sovellettaessa (ja muutenkin kerrosarkkitehtuuria sovellettaessa) on kerros joka kuvaa *domainin*, eli sisältää *liiketoimintaoliot*
- Esim. Kumpula Biershopin domain-oliot:
  - Tuote
  - Varasto
  - Ostos
  - Ostoskori
  - Asiakas
  - Ostostapahtuma

# Domain Driven Design

- Domain-oliot tai osa niistä yleensä mäpätään tietokantaan
  - Mäppäyksessä käytetään usein DAO-suunnittelumallia, johon tutustuimme ohimennen laskareissa 3
  - DAO on oleellisesti sama asia jota kutsutaan data mapperiksi:
    - <http://martinfowler.com/eaaCatalog/dataMapper.html>
  - DAO:n lisäksi on muitakin mäppäystapoja, kuten Ruby on Railsin käyttämä Active Record  
<http://martinfowler.com/eaaCatalog/activeRecord.html>
- Domain-oliot tietokantaan mäppäävät komponentit muodostavat oman kerroksen kerrosarkkitehtuurissa
- Joissain suunnittelutyyleissä Domain-olioiden ja sovelluksen käyttöliittymän välissä on vielä erillinen palveluiden kerros
  - <http://martinfowler.com/eaaCatalog/serviceLayer.html>
- Palvelut koordinoivat domain-olioille suoritettavaa toiminnallisuutta, esim. *ostoksen laitto ostoskoriin* tai *ostosten maksaminen*
- Ideana on eristää palveluiden avulla kaikki sovelluslogiikka käyttöliittymältä

# Palvelukerros Kumpula Biershopissa

- Palvelukeroksessa on jokaisen käyttöliittymätason toiminnallisuuden toteutus omana **command**-suunnittelumallin mukaisena oliona
  - Seuraavilla havainnollistavana esimerkkinä LisäysKoriin-olian luonti ja kutsu
  - LisäysKoriin-olio suorittaa kaiken interaktion domain-olioiden kanssa
  - Käyttöliittymä käyttää domain-olioita ainoastaan web-sivulla näytettävän datan renderöintiin
- Komento-oliot muodostavat oikeastaan **fasaadi**-suunnittelumallin mukaisen eristävän kerroksen käyttöliittymän ja alempien kerrostosten välille
  - Tarjoaa hyvin rajatun rajapinnan jonka kautta kerrostosta käytetään, eristää kerroksen toiminnallisuuden täysin
  - [http://sourcemaking.com/design\\_patterns/facade](http://sourcemaking.com/design_patterns/facade)
- Sovelluslogiikan testaaminen ilman käyttöliittymää onnistuu helposti yksikkötesteillä testaamalla command-olioiden ja domain-olioiden interaktiota

käyttöliittymä



palvelut

LisaysKoriin

...

MaksunSuoritus

<<interface>>

Postitusjärjestelmä

<<interface>>

Luottokunta

malli

Asiakas

Ostotapahtuma

Varasto

Ostoskori

Ostos

Tuote

tietokantayhteydet (DAO)

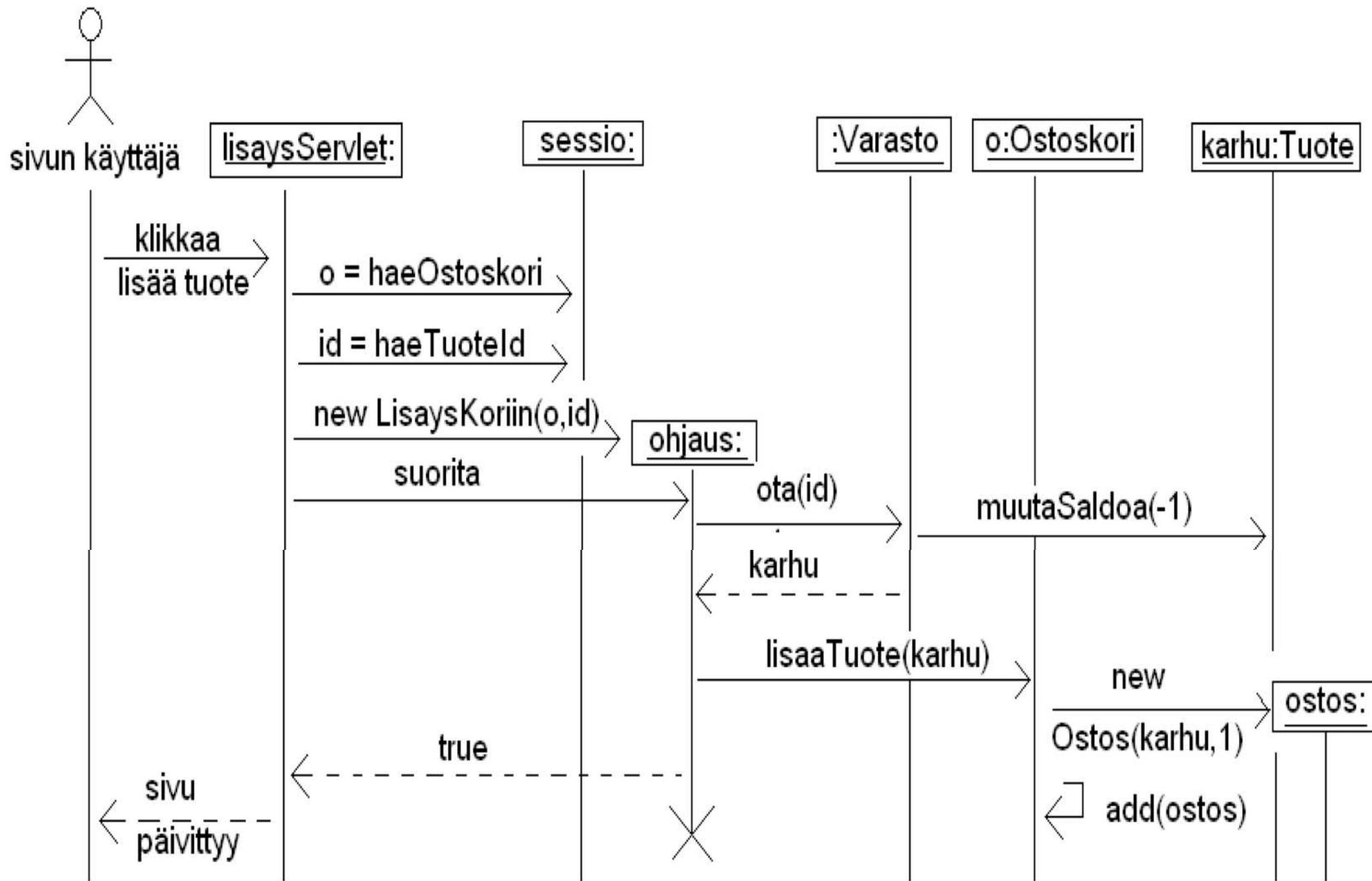
# Tuotteen lisäämisenestä huolehtiva kontrolleri

- Kun käyttäjä klikkaa nappia "lisää tuote ostoskoriin" suoritetaan koodi, joka luo tuotteen ostoskoriin lisäävän komento-olian
  - Komento-olio saa konstruktoriparametrina ostoskorin ja lisättävän tuotteen id:n

```
post("/tuotteet", (request, response) -> {
    OstoksenLisaysKoriin komento = new OstoksenLisaysKoriin(
        getOstoskoriFrom(request),
        new ObjectId(request.queryParams("id"))
    );
    komento.suorita();
    // palautetaan käyttäjä tuotelistanäkymään
    response.redirect("/tuotteet");
});
```

# Komento-olio OstoksenLisaysKoriin

```
public class OstoksenLisaysKoriin {  
    private Ostoskori ostoskori;  
    private ObjectId tuoteld;  
    private Varasto varasto;  
  
    public OstoksenLisaysKoriin(Ostoskori ostoskori, ObjectId tuoteld) {  
        this.ostoskori = ostoskori;  
        this.tuoteld = tuoteld;  
        this.varasto = Varasto.getInstance();  
    }  
  
    public void suorita() {  
        Tuote tuote = varasto.otaVarastosta(tuoteld);  
        if (tuote==null) { return; }  
        ostoskori.lisaaTuote(tuote);  
    }  
}
```

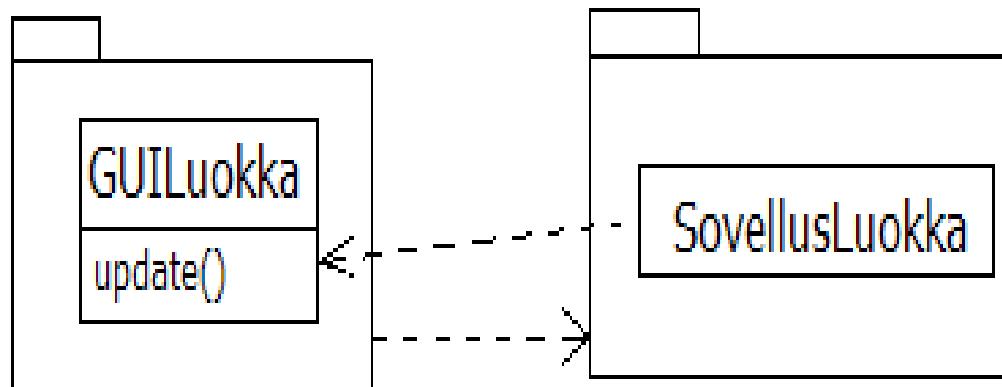


# Model View Controller el MVC -malli

- MVC-mallilla tarkoitetaan periaatetta, jonka avulla **malli** (model) eli liiketoimintalogiikan sisältävät olioit (esim. domain-oliot) eristetään käyttöliittymän **näytöt** (view) generoivasta koodista
  - Kumpula Biershopissa on oikeastaan sovellettu WebMVC:tä, eli MVC:n www-sovelluksiin sopivaa varianttia
- Ideana on laittaa näytön/näytöt generoivan koodin ja sovelluslogiikasta huolehtivien olioiden väliiin **kontrolleri** (controller)
- Kontrolleri huolehtii esim. nappien klikkaamisen tai web-sovelluksissa osoitteisiin navigoinnin tai lomakkeiden lähetämisen edellyttävän toiminnallisuuden suorittamisesta kutsumalla sopivia modelin olioita
- Näytöt generoivat käyttäjälle näytettävän käyttöliittymän käyttäen joko suoraan malleissa olevaa dataa tai saamalla datan kontrollerin välityksellä (kuten WebMVC:ssä tapahtuu)
  - ks. <https://github.com/mluukkai/ohtu2017/blob/master/web/luento9.md> kohta MVC
- Model ei tunne kontrollereja eikä näyttöjä ja samaan modelissa olevaan dataan voikin olla useita näyttöjä

# Riippuvuuksien eliminointi

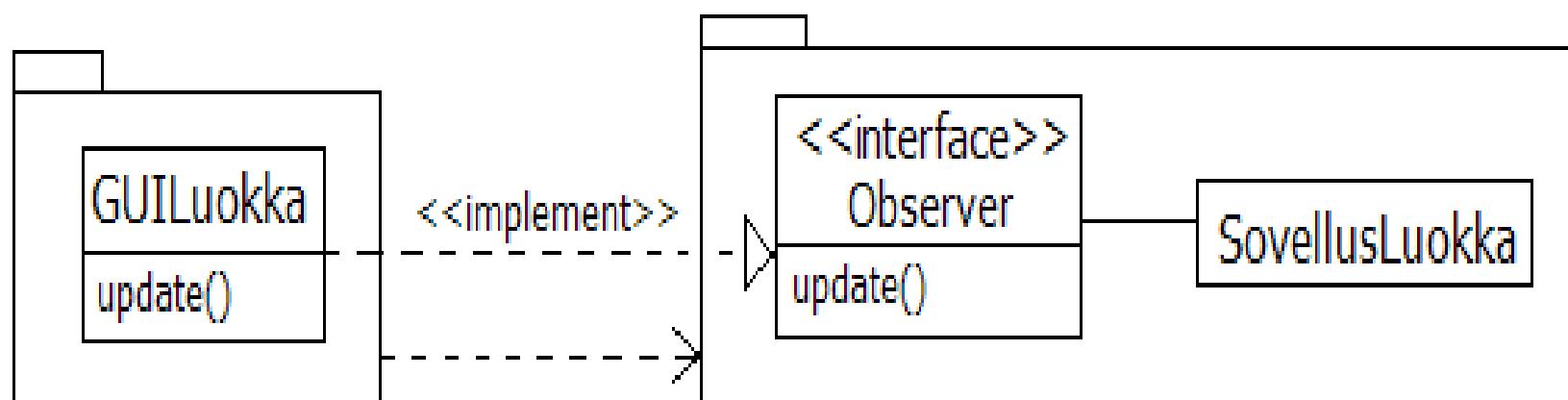
- Kerrosarkkitehtuurissa ja MVC-mallin mukaisissa sovelluksissa törmätään usein tilanteeseen, jossa sovelluslogiikan on kerrottava käyttöliittymälle jonkin sovellusolion tilan muutoksesta, jotta käyttöliittymä näyttäisi koko ajan ajantasaista tietoa
- Tästä muodostuu ikävä riippuvuus sovelluslogiikasta käyttöliittymään
- Kuvitellaan, että sovelluslogiikka ilmoittaa muuttuneesta tilasta kutsumalla jonkin käyttöliittymän luokan toteuttamaa metodia *update()*
  - Parametrina voidaan esim. kertoa muuttunut tieto



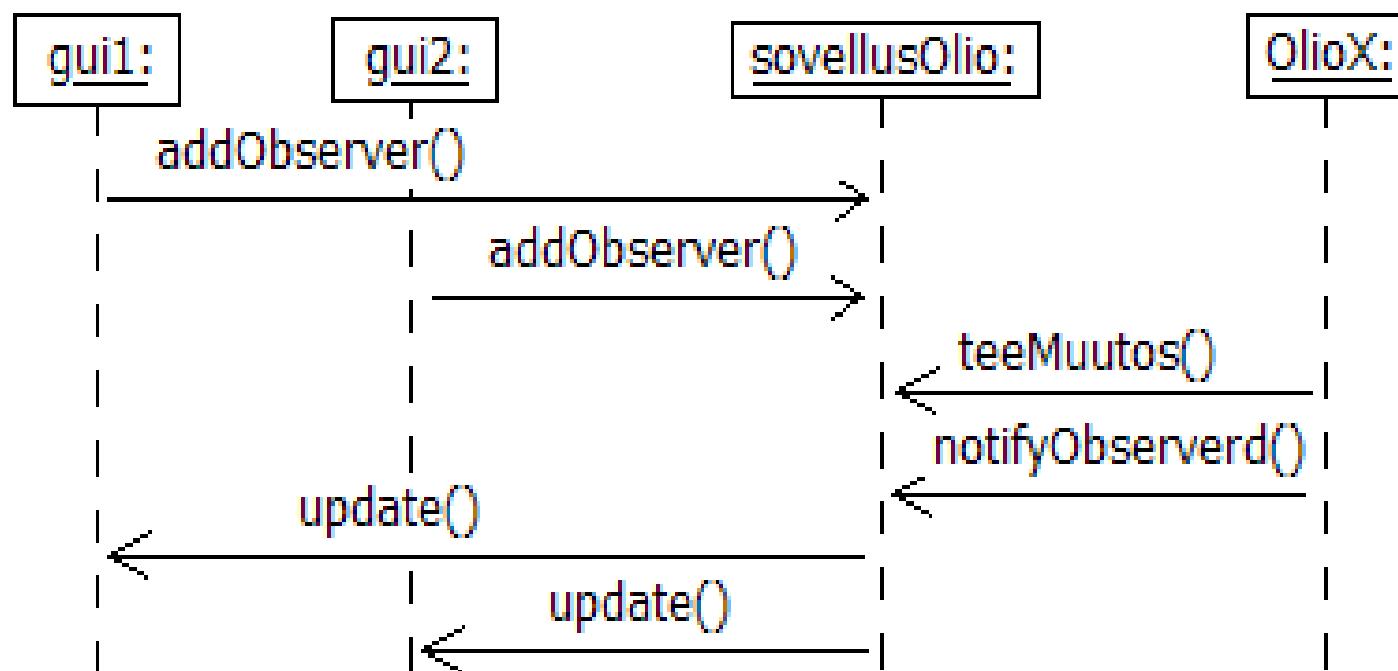
- Riippuvuus saadaan eliminoitua **observer**-suunnittelumallilla
  - Ks <https://github.com/mluukkai/ohtu2017/blob/master/web/luento9.md> kohta Observer

# Riippuvuuksien eliminointi observer-suunnittelumallilla

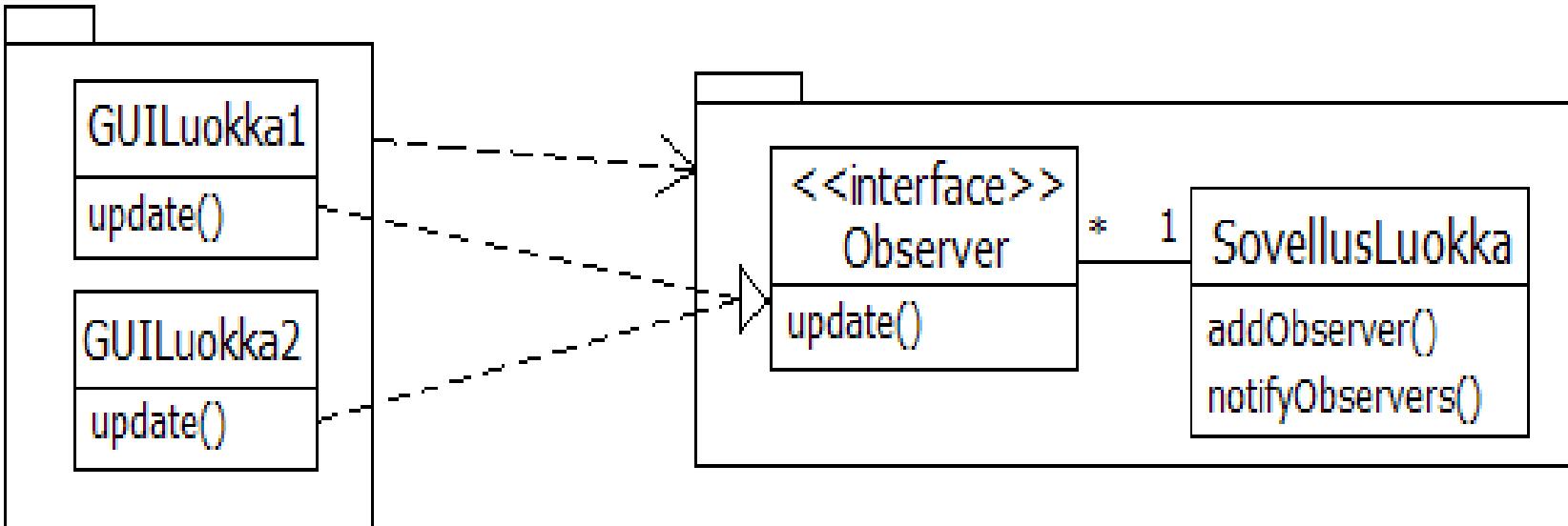
- Määritellään rajapinta, joka sisältää käyttöliittymäluokan päivitysmetodin update(), jota sovellusluokka kutsuu
  - Alla rajapinnalle on annettu nimeksi *Observer*
- Käyttöliittymäluokka toteuttaa rajapinnan, eli käytännössä toteuttaa update()-metodin haluamallaan tavalla
- Sovellusluokalle riittää nyt tuntea ainoastaan rajapinta, jonka metodia update() se tarvittaessa kutsuu
- Nyt kaikki menee siististi, sovelluslogiikasta ei enää ole riippuvuutta käyttöliittymään ja silti sovelluslogiikka voi kutsua käyttöliittymän metodia
  - Sovellusluokka tuntee siis vain rajapinnan, joka on määritelty sovelluslogiikkapakkauksessa



- Kyseessä on **observer**- eli tarkkailijasuunnittelumalli
  - [http://sourcemaking.com/design\\_patterns/observer](http://sourcemaking.com/design_patterns/observer)
- Jos käyttöliittymäolio haluaa tarkkailla jonkin sovellusolion tilaa, se toteuttaa Observer-rajapinnan ja rekisteröi rajapintansa tarkkailtavalle sovellusoliolle
  - Sovellusoliolla metodi addObserver()
  - Näin sovellusolio tuntee kaikki sitä tarkkailevat rajapinnat
- Kun joku muuttaa sovellusolion tilaa, kutsuu se sovellusolion metodia notifyObservers(), joka taas kutsuu kaikkien tarkkailijoiden update()- metodeja, joiden parametrina voidaan tarvittaessa välittää muutostieto



# Observer-suunnittelumalli



```
public class Sovellusluokka{
```

```
    ArrayList<Observer> tarkkailijat;
```

```
    void addObserver(Observer o){
```

```
        tarkkailijat.add(o);
```

```
}
```

```
    void notifyObservers(){
```

```
        for ( Observer o : tarkkailijat) o.update();
```

```
}
```

```
/* muu koodi */
```

```
Interface Observer{
```

```
    void update();
```

```
}
```

```
GUILuokka implements Observe {
```

```
    void update(){
```

```
        /* päivitetään näytöä */
```

```
}
```

```
/* muu koodi*/
```

```
}
```

# Tekninen velka

- Edellisten luentojen aikana tutustuimme moniin ohjelman sisäistä laatua kuvaaviin attribuutteihin:
  - kapselointi, koheesio, riippuvuuksien vähäisyys, testattavuus, luettavuus
- Tutustuimme myös yleisiin periaatteisiin, joiden noudattaminen auttaa päätymään laadukkaaseen koodiin
  - single responsibility principle, program to interfaces, favor composition over inheritance, don't repeat yourself
- Sekä suunnittelumalleihin (design patterns), jotka tarjoavat tiettyihin sovellustilanteisiin sopivia yleisiä ratkaisumalleja
- Koodi ja oliosuunnittelu ei ole aina hyvää, ja joskus on jopa asiakkaan kannalta tarkoitukseenmukaista tehdä "huonoa" koodia
- Huonoa oliosuunnittelua ja huonon koodin kirjoittamista on verrattu ***velan*** (engl. **design debt** tai **technical debt**) ottamiseen
  - <http://www.infoq.com/articles/technical-debt-leison>
- Piittaamattomalla ja laiskalla ohjelmoinnilla/suunnittelulla saadaan ehkä nopeasti aikaan jotain, mutta hätäinen ratkaisu tullaan maksamaan korkoineen takaisin myöhemmin jos ohjelmaa on tarkoitus laajentaa
  - Käytännössä käy niin, että tiimin velositeetti laskee, koska "teknistä velkaa" on maksettava takaisin, jotta järjestelmään saadaan toteutettua uusia ominaisuuksia

# Tekninen velka

- Jos korkojen maksun aikaa ei koskaan tule, ohjelma on esim. pelkkä prototyyppi tai sitä ei oteta koskaan käyttöön, voi "huono koodi" olla asiakkaan kannalta kannattava ratkaisu
- Vastaavasti joskus voi "lyhytaikaisen" teknisen velan ottaminen olla järkevää tai jopa välittämätöntä
  - Esim. voidaan saada tuote nopeammin markkinoille tekemällä tietoisesti huonoa designia, joka korjataan myöhemmin
  - <http://blogs.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx> (linkki rikki)
- Tekniselle velalle on yritetty jopa arvioida hintaa:
  - <http://www.infoq.com/news/2012/02/tech-debt-361>
- Kaikki tekninen velka ei ole samanlaista, Martin Fowler jaottelee teknisen velan neljään eri luokkaan:
  - Reckless and deliberate: "*we do not have time for design*"
  - Reckless and inadvertent: "*what is layering*?"
  - Prudent and deliberate: "*we must ship now and will deal with consequences*"
  - Prudent and inadvertent: "*now we know how we should have done it*"
  - <http://martinfowler.com/bliki/TechnicalDebtQuadrant.html>

# Koodi haisee: merkki huonosta suunnittelusta

- Seuraavassa alan ehdoton asiantuntija Martin Fowler selittää mistä on kysymys **koodin hajuista**:
  - **A code smell is a surface indication that usually corresponds to a deeper problem in the system.** The term was first coined by Kent Beck while helping me with my Refactoring book.
  - The quick definition above contains a couple of subtle points. Firstly **a smell is by definition something that's quick to spot** - or sniffable as I've recently put it. *A long method is a good example of this - just looking at the code and my nose twitches if I see more than a dozen lines of java.*
  - The second is that smells don't always indicate a problem. Some long methods are just fine. You have to look deeper to see if there is an underlying problem there - smells aren't inherently bad on their own - they **are often an indicator of a problem rather than the problem themselves.**
  - One of the nice things about smells is that **it's easy for inexperienced people to spot them**, even if they don't know enough to evaluate if there's a real problem or to correct them. I've heard of lead developers who will pick a "smell of the week" and ask people to look for the smell and bring it up with the senior members of the team. Doing it one smell at a time is a good way of gradually teaching people on the team to be better programmers.

# Koodihajuja

- Koodihajuja on hyvin monenlaisia ja monentasoisia
- On hyvä oppia tunnistamaan ja välttämään tavanomaisimpia
- Internetistä löytyy paljon hajulistoja, esim:
  - <http://sourcemaking.com/refactoring/bad-smells-in-code>
  - <http://c2.com/xp/CodeSmell.html>
  - <http://www.codinghorror.com/blog/2006/05/code-smells.html>
- Muutamia esimerkkejä helposti tunnistettavista hajuista:
  - Duplicated code (eli koodissa copy pastea...)
  - Methods too big
  - Classes with too many instance variables
  - Classes with too much code
  - Long parameter list
  - Uncommunicative name
  - Comments (eikö kommentointi muka ole hyvä asia?)

# Koodihajuja

- Seuraavassa pari ei ehkä niin ilmeistä tai helposti tunnistettavaa koodihajua
- **Primitive obsession**
  - Don't use a gaggle of primitive data type variables as a poor man's substitute for a class. If your data type is sufficiently complex, write a class to represent it.
  - <http://sourcemaking.com/refactoring/primitive-obsession>
- **Shotgun surgery**
  - If a change in one class requires cascading changes in several related classes, consider refactoring so that the changes are limited to a single class.
  - <http://sourcemaking.com/refactoring/shotgun-surgery>

# Koodin refaktoriointi

- Lääke koodihajuun on *refaktoriointi* eli muutos koodin rakenteeseen joka kuitenkin pitää koodin toiminnan ennallaan
- Erikoisia koodin rakennetta parantavia refaktorointeja on lukuisia
  - ks esim. <http://sourcemaking.com/refactoring>
- Muutama käyttökelpoinen nykyaikaisessa kehitysympäristössä (esim NetBeans, Eclipse, IntelliJ) automatisoitu refaktoriointi:
  - **Rename method** (rename variable, rename class)
    - Eli uudelleennimetään huonosti nimetty asia
  - **Extract method**
    - Jaetaan liian pitkä metodi erottamalla siitä omia apumetodejaan
  - **Extract interface**
    - Luodaan luokan julkisia metodeja vastaava rajapinta, jonka avulla voidaan purkaa olion käyttäjän ja olion väliltä konkreettinen riippuvuus
  - **Extract superclass**
    - Luodaan yliluokka, johon siirretään osa luokan toiminnallisuudesta

# Miten refaktorointi kannattaa tehdä

- Refaktorioiden melkein ehdoton edellytys on kattavien testien olemassaolo
  - Refaktorioiden tarkoitus ainoastaan parantaa luokan tai komponentin sisäistä rakennetta, ulospäin näkyvän toiminnallisuuden pitäisi pysyä muuttumattomana
- Kannattaa ehdottomasti edetä pienin askelin
  - Yksi hallittu muutos kerrallaan
  - Testit on ajettava mahdollisimman usein ja varmistettava että mikään ei mennyt rikki
- Refaktorointia kannattaa suorittaa lähes jatkuvasti
  - Koodin ei kannata antaa "rapistua" pitkiä aikoja, refaktorointi muuttuu vaikeammaksi
  - Lähes jatkuva refaktorointi on helppoa, pitää koodin rakenteen selkeänä ja helpottaa sekä nopeuttaa koodin laajentamista
- Osa refaktoroinneista, esim. metodien tai luokkien uudelleennimentä tai pitkien metodien jakaminen osametodeiksi on helppoa, aina ei näin ole
  - Joskus on tarve tehdä isoja refaktorointeja joissa ohjelman rakenne eli arkkitehtuuri muuttuu

# Java 8:n tuomia mahdollisuuksia

- Luennolla 8 tutustuimme jo hieman Java 8:n lambda-lausekkeiden ja stream-apin tarjoamiin mahdollisuuksiin
  - ks.  
<https://github.com/mluukkai/ohtu2017/blob/master/web/luent08.md#koodissa-olevan-epätriviaalin-copypasten-poistaminen-strategy-patternin-avulla-java-8a-hyödyntävä-versio>
- Jatketaan Java 8:aan tutustumista
  - ks.  
[https://github.com/mluukkai/ohtu2017/blob/master/web/java8\\_esimerkkeja.md](https://github.com/mluukkai/ohtu2017/blob/master/web/java8_esimerkkeja.md)

Ohjelmistotuotanto

Luento10

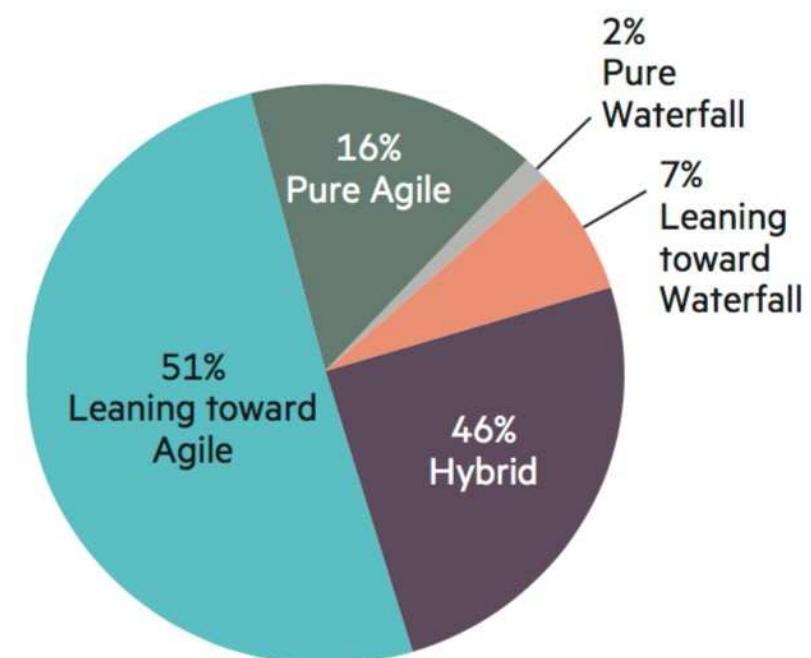
24.4.

# Mitä menetelmiä tulisi käyttää?

- Kurssilla esitelty todella suuri määrä prosessiin liittyviä asioita ja erilaisia työkaluja, mitä niistä tulisi käyttää?
- Vastausta ei ole
- Yksi hyvä lähtökohta on aloittaa seuraavasti
  - By the book Scrum
  - Mahdollisimman hyvä deployment pipeline
    - Automaattiset testit, CI ja automaattinen/helppo deployaus tuotantoypäristöön
- Tämän jälkeen ***inspect and adapt***
  - Prosessia, työskentelytapoja ja työkaluja tulee mukauttaa tarpeen mukaan
  - Oleellinen osa agilea on juuri se että prosessi taipuu tarpeiden mukaan

# Miten laajalti Agilea käytetään

- Internetistä löytyy aiheesta jossain määrin dataa, ei tosin kovin tuoreutta
- Forrester surveyed (2009) nearly 1,300 IT professionals and found that **35 percent of respondents stated that agile most closely reflects their development process**
  - <http://www.infoworld.com/d/developer-world/agile-software-development-now-mainstream-190>
- **Agile methodologies are the primary approach for 39 percent** of responding developers, making Agile development the dominant methodology in North America. **Waterfall development, is the primary methodology of 16.5 percent** of respondents (2010)
  - <http://visualstudiomagazine.com/articles/2010/03/01/developers-mix-and-match-agile-approaches.aspx>
- HP:n vuonna 2015 tekemä tutkimus julistaa "Agile is the new normal"
  - Tutkimuksessa 601 vastaajaa



# Miten laajalti Agilea käytetään

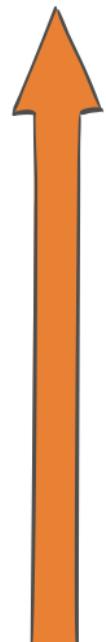
- HP:n tutkimus ei määrittele kovin hyvin käsitteitä
  - A hybrid approach: incorporate at least some Agile solutions and principles
  - Leaning towards agile jää määrittelemättä
-

# Mitä ketteriä menetelmiä käytetään?

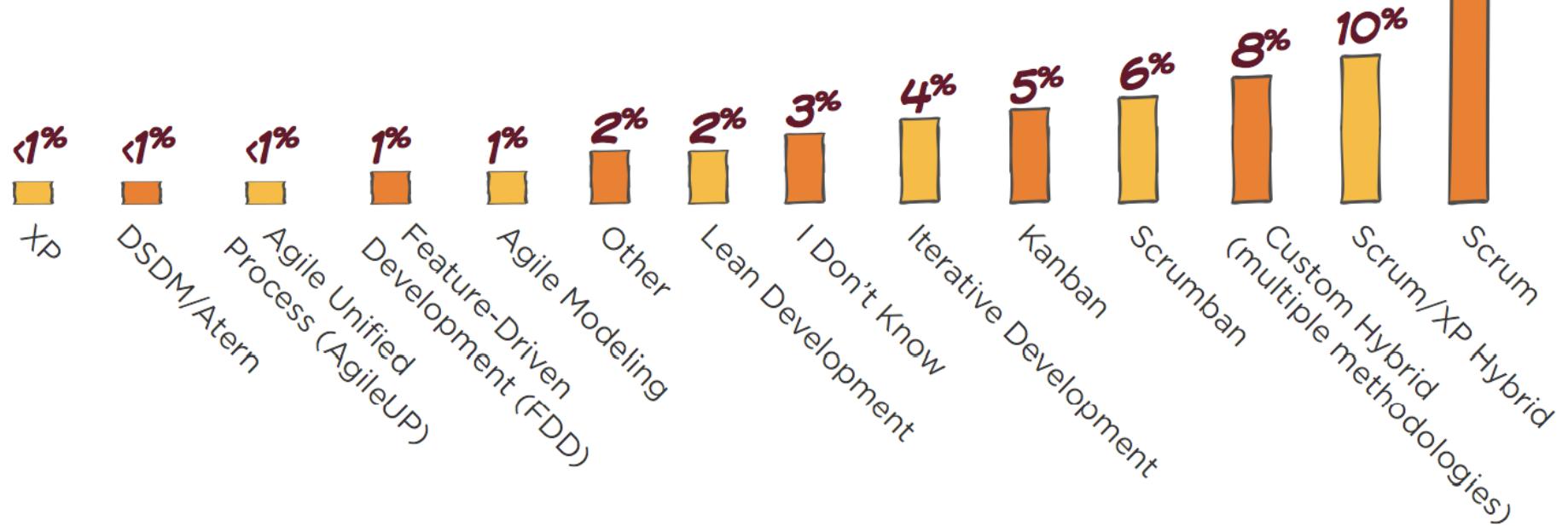


## Agile Methods and Practices

56%



### AGILE METHODOLOGY USED



- VersionOnen "internetin virallisesta" vuosiraportista
  - <http://stateofagile.versionone.com>

# Ketterät käytänteet

- VersionOne:

<b>80%</b> Daily standup	<b>38%</b> Open work area
<b>79%</b> Short iterations	<b>36%</b> Refactoring
<b>79%</b> Prioritized backlogs	<b>34%</b> Test-Driven Development (TDD)
<b>71%</b> Iteration planning	<b>31%</b> Kanban
<b>69%</b> Retrospectives	<b>29%</b> Story mapping
<b>65%</b> Release planning	<b>27%</b> Collective code ownership
<b>65%</b> Unit testing	<b>24%</b> Automated acceptance testing
<b>56%</b> Team-based estimation	<b>24%</b> Continuous deployment
<b>53%</b> Iteration reviews	<b>21%</b> Pair programming
<b>53%</b> Taskboard	<b>13%</b> Agile games
<b>50%</b> Continuous integration	<b>9%</b> Behavior-Driven Development (BDD)
<b>48%</b> Dedicated product owner	
<b>46%</b> Single team (integrated dev & testing)	
<b>43%</b> Coding standards	

- Suomen tilanne:

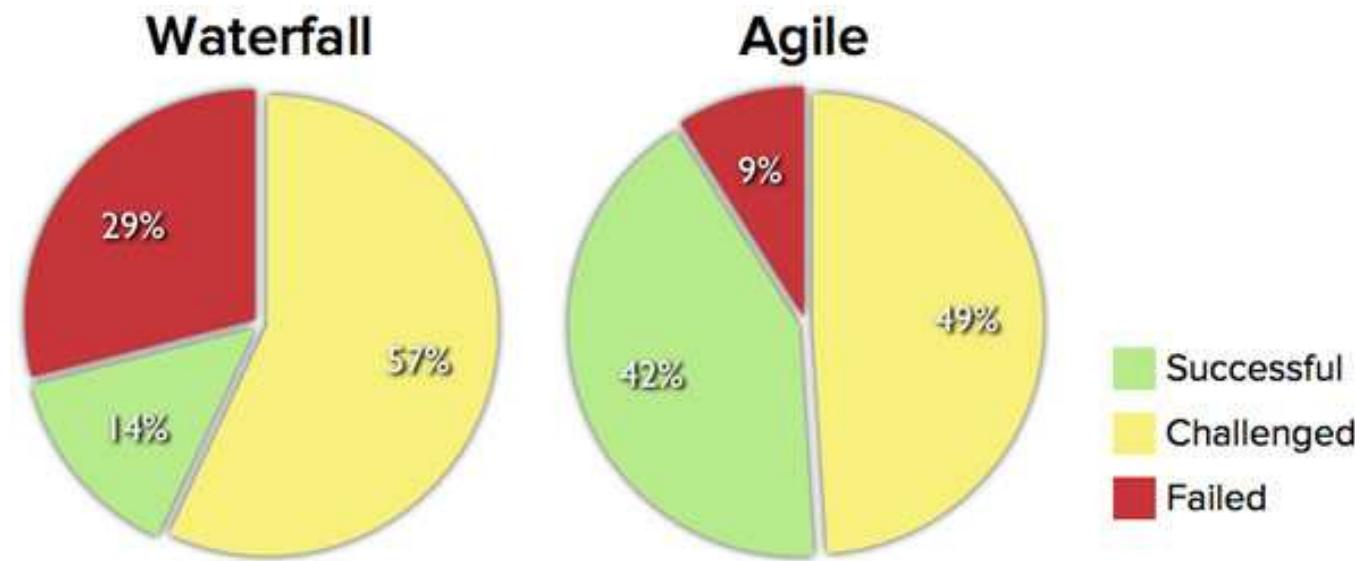
[http://esem.cs.lth.se/industry\\_public/Rodriguezetal\\_ESEM2012\\_IndustryTrack\\_1\\_0.pdf](http://esem.cs.lth.se/industry_public/Rodriguezetal_ESEM2012_IndustryTrack_1_0.pdf)

# Ketterät käytänteet Suomesta tehdynssä tutkimuksessa (n=225)

Practices	n	Mean	Median
Prioritized work list	204	4.2	4
Iteration/sprint planning	203	4.1	4
Daily stand-up meetings	209	3.7	4
Unit testing	199	3.7	4
Release planning	196	3.9	4
Active customer participation	196	3.5	4
Self-organizing teams	194	3.5	4
Frequent and incremental delivery of working software	189	4.1	4
Automated builds	185	3.5	4
Continuous integration	182	3.8	4
Test-driven development (TDD)	179	2.7	3
Retrospectives	177	3.6	4
Burn-down charts	174	3.2	3
Pair programming	174	2.4	2
Refactoring	163	3.4	3
Collective code ownership	159	3.3	3

# Projektien onnistuminen: ketterä vastaan perinteinen

- Standish CHAOS raport 2012



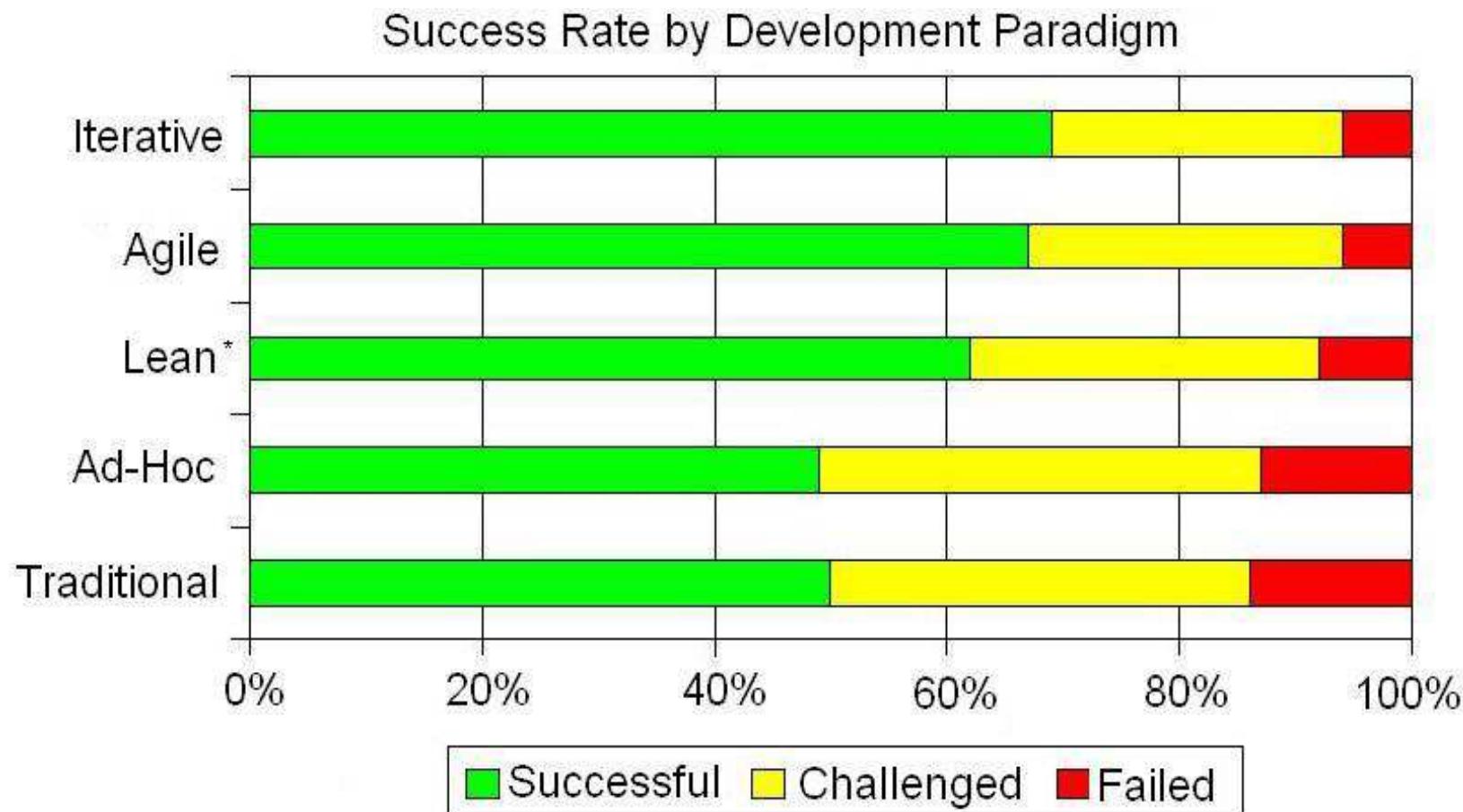
Source: The CHAOS Manifesto, The Standish Group, 2012.

- Columbus discovering Agile, laaja kyselytutkimus, alustavia tuloksia
  - Early results from the Columbus-area participants show that a typical business system comprising 50,000 lines of code is **completed 31% faster** than the industry average in the QSM industry database of completed projects. Even more remarkable is the **defect rate, which is 75% lower** than the industry norm.
  - <http://www.infoq.com/news/2012/11/success-agile-projects>

# Projektien onnistuminen: ketterä vastaan perinteinen

- Scott Ambler, Agile vs perinteinen 2011:

<http://www.drdobbs.com/architecture-and-design/how-successful-are-it-projects-really/232300110>



2011 is the first year where we asked about Lean.

We only had 40 respondents for this paradigm.

Copyright 2011 Scott W. Ambler [www.ambysoft.com/surveys/](http://www.ambysoft.com/surveys/)

# Mitä oikeastaan tarkoitetaan projektin onnistumisella?

- Ambler: Here's how respondents, on average, define success:
  - **Time/schedule**: 20% prefer to deliver on time according to the schedule, 26% prefer to deliver when the system is ready to be shipped, and 51% say both are equally important.
  - **Return on investment (ROI)**: 15% prefer to deliver within budget, 60% prefer to provide good return on investment (ROI), and 25% say both are equally important.
  - **Stakeholder value**: 4% prefer to build the system to specification, 80% prefer to meet the actual needs of stakeholders, and 16% say both are equally important.
  - **Quality**: 4% prefer to deliver on time and on budget, 57% prefer to deliver high-quality systems that are easy to maintain, and 40% say both are equally important.

# Ketteryydellä saavutettuja etuja tarkemmin eriteltyinä

- VersionOne 2016



# Ketteryydellä saavutettuja etuja Suomessa...

Effect	n	Mean	Median
Improved team communication	204	4,0	4
Enhanced ability to adapt to changes	203	3,9	4
Increased productivity	201	3,8	4
Enhanced process quality	198	3,7	4
Improved learning and knowledge creation	197	3,7	4
Enhanced software quality	196	3,8	4
Accelerated time-to-market/cycle time	192	3,7	4
Reduced waste and excess activities	190	3,5	4
Improved customer collaboration	190	3,7	4
Improved organizational transparency	187	3,5	4
Improved customer understanding	188	3,7	4

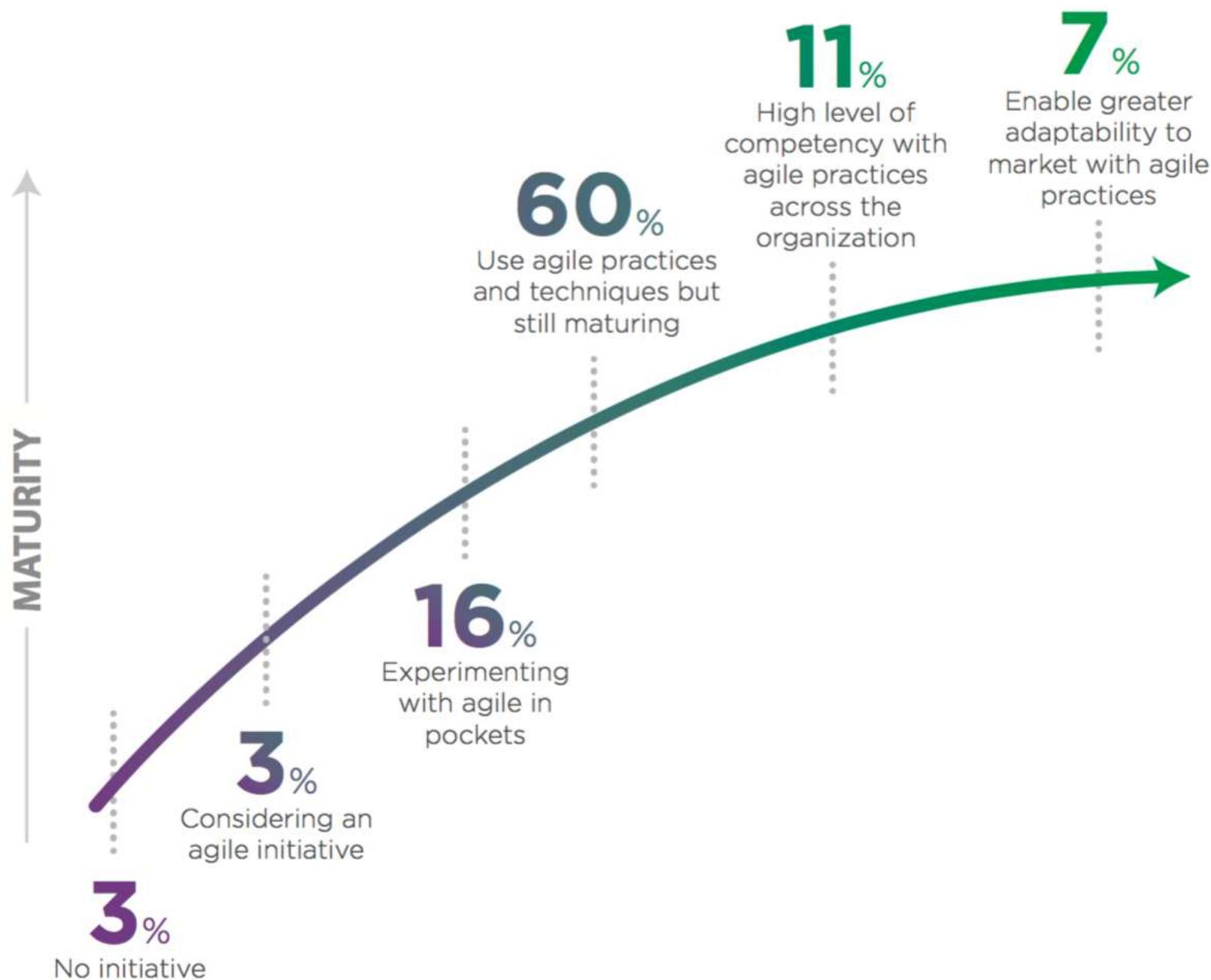
# Suurimmat haasteet ketterien menetelmien käytössä

- VersionOne 2016



\*Respondents were able to make multiple selections.

# Miten hyvin organisaatiot kokevat hallitsevansa ketterät menetelmät?



# Evidenssiä on, mutta...

- Oikeastaan kaikki edelliset olivat kyselytutkimuksia
  - käsitteitä ei ole kunnolla määritelty (esim. mitä ketteryydellä tai projektin onnistumisella tarkoitetaan)
  - Kyselyyn osallistuneet eivät vältämättä edusta tasaisesti koko populaatiota
  - Kaikkien kyselyjen tekijät eivät puolueettomia menetelmien suhteen (esim. Ambler ja VersionOne)
- Eli tutkimusten validiteetti on kyseenalainen
- Toisaalta kukaan ei ole edes yritynyt esittää evidenssiä, jonka mukaan vesiputousmalli toisi systemaattisia etuja ketteriin menetelmiin verrattuna
- Myös akateemista tutkimusta on todella paljon (mm. Markkulan ym. kyselytutkimus) ja eri asioihin kohdistuvaa. Akateemisenkin tutkimuksen systemaattisuus, laatu ja tulosten yleistettävyys vaihtelee
  - Ohjelmistotuotannossa on liian paljon muuttujia, jotta jonkin yksittäisen tekijän vaikutusta voitaisiin täysin vakuuttavasti mitata empiirisesti
  - Menetelmiä soveltavat kuitenkin aina ihmiset, ja mittaustulos yhdellä ohjelmistotiimilla ei vältämättä yleisty miinkään muihin olosuhteisiin
- Olemassa olevan evidenssin nojalla kuitenkin näyttää siltä, että ongelmaistaan huolimatta ketterät menetelmät ovat ainakin joissakin tapauksissa järkevä tapa ohjelmistokehitykseen

Koe

# Koe

- Tiistaina 9.5 klo 16:00 – salissa A111
- Kurssin pisteytys
  - Koe 20p
  - Laskarit 10p
  - Miniprojekti 10p
- Kurssin läpipääsy edellyttää
  - 50% pisteistä
  - 50% kokeen pisteistä
  - Hyväksyttyä miniprojektia
- Kokeessa on sallittu yhden A4:n kokoinen käsin, itse kynällä kirjoitettu lunttilappu

# Mitä kokeessa ei tarvitse osata

- Git
- Gradle
- Travis
- JUnit
- Mockito
- Cucumber
- Selenium
- Java 8

# Reading list – eli lue nämä

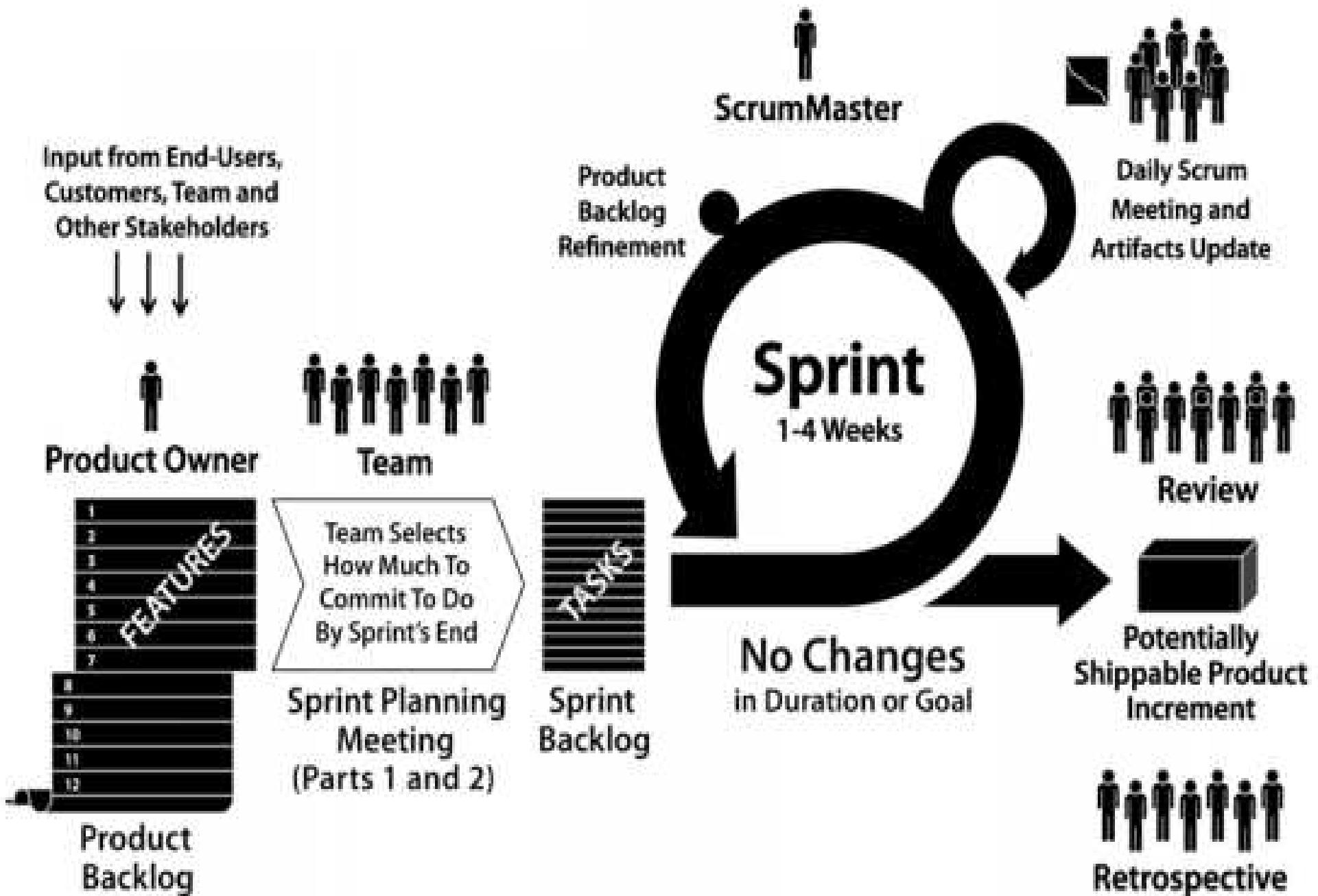
- Luentomonisteet, luentoihin 8 ja 9 liittyvät koodiesimerkit ja laskarit (paitsi edellisellä sivulla mainittujen osalta)
- <http://martinfowler.com/articles/newMethodology.html>
- <http://www.scrumguides.org/docs/scrumguide/v1/Scrum-Guide-US.pdf>
- <http://www.infoq.com/minibooks/scrum-xp-from-the-trenches>
  - Sivut 1-86 (painos 1), sivut 1-92 (painos 2)
- <http://martinfowler.com/articles/continuousIntegration.html>
- <http://martinfowler.com/articles/designDead.html>
- [http://sourcemaking.com/design\\_patterns](http://sourcemaking.com/design_patterns)
  - Tarpeellisissa määrin

Tärkeät teemat vielä pikakelauksella

# Luento 1

- Termi software engineering
  - Mitä pitää sisällään
- Prosessimallit
  - Vaiheet
    - Vaatimusmäärittely
    - Suunnittelu
    - Toteutus
    - Testaus
    - Ylläpito
  - vesiputous/lineaarinen/BUFD
  - Iteratiivinen
  - Ketterä
- Motivaatio prosessimallien kehittymiselle

# Luento 2: Scrum



# Luento 3: vaatimusmäärittely

- Vaatimukset jakautuvat
  - Toiminnallisiin
  - Ei-toiminnallisiin (rajoitteet ja laatuvaatimukset)
- Vaatimusmäärittelyn luonne ja vaiheet
  - oldschool vs. moderni
- Ketterä vaatimustenhallinta
  - User story
    - Arvoa tuottava toiminnallisuus
    - ”Card, conversation, confirmation”
    - INVEST
    - Estimointi

# Luento 4

- Ketterä vaatimustenhallinta
  - Product backlog
    - DEEP
  - Julkaisun suunnittelu
  - Velositeetti
- Sprintin suunnittelu
  - Storyjen valinta / planning game
  - Storyistä taskeihin
- Sprint backlog
  - Taskboard
  - burndown

# Luento 5

- Validointi "are we building the right product"
  - Katselointi ja tarkastukset
  - Vaatimusten validointi (ketterä vs. trad)
  - Koodin katselointi
- Verifointi "are we building the product right"
  - Vastaako järjestelmä vaatimusmäärittelyä
- Verifointi tapahtuu yleensä testauksen avulla
  - Testauksen tasot:
    - Yksikkö-, Integraatio-, Järjestelmä-, Hyväksymätestaus
  - Käsitteitä:
    - black box, white box, ekvivalenssiluokka, raja-arvo, testauskattavuus
  - regressiotestaus
  - Ohjelman ulkoinen laatu vs. sisäinen laatu

# Luento 6

- Testaus ketterissä menetelmissä
  - Automaattiset regressiotestit tärkeät
- TDD
  - Red – green – refactor
  - Enemmän suunnittelua kun testausta, testit sivutuotteena
- Storytason testaus / ATDD / BDD
- Jatkuva integraatio ja jatkuva käyttöönotto
  - "integraatiohelvetti" → Daily build / smoke test → jatkuva integraatio → continuous delivery → continuous deployment
  - CI/CD ei ole pelkkä työkalu vaan workflow ja mentaliteetti
- Tutkiva testaus
  - "Exploratory testing is simultaneous learning, test design and test execution"

# Luento 7

- Ohjelmiston arkkitehtuurin määritelmiä
- Arkkitehtuurimallit: kerrosarkkitehtuuri
- Arkkitehtuurin kuvaaminen
  - Monia näkökulmia, erilaisia kaavioita
- Arkkitehtuuri ketterissä menetelmissä
  - Ristiriita arkkitehtuurivetoisuuden ja ketterien menetelmien välillä
  - Inkrementaalinen arkkitehtuuri
    - Edut ja haitat

# Luento 8 – oliosuunnittelu

- Helposti ylläpidettävän eli sisäiseltä laadultaan hyvän koodin tunnusmerkit ja laatuatribuutit
  - kapselointi, koheesio, riippuvuuksien vähäisyys, toisteettomuus, selkeys, testattavuus
- Oliosuunnittelun periaatteita
  - Single responsibility principle
  - Program to an interface not to an implementation
  - Favour composition over inheritance
  - DRY eli Don't repeat yourself
- Suunnittelumalleja
  - Composed method
  - Static factory
  - Strategy
  - Command
  - Template method

# Luento 9

- Suunnittelumalleja
  - Dekoraattori
  - Rakentaja (builder)
  - Adapteri
  - Komposiitti
  - Proxy
  - Mvc
  - Observer
- Aiemmin kurssilla kolme suunnittelumallia
  - Riippuvuuksien injekointi (dependency injection)
  - Singleton
  - DAO, Data access object
- Domain driven design ja kerrosarkkitehtuuri
- Käsitteet tekninen velka technical/design debt, koodihaju ja refaktorointi