

# Ohjelmistotuotanto Kevät 2013

Matti Luukkainen

assistentteina:  
Kasper Hirvikoski ja Tero Huomo

Luento 1

11.3.2012

WE'RE GOING TO  
TRY SOMETHING  
CALLED AGILE  
PROGRAMMING.



www.dilbert.com scottadams@aol.com

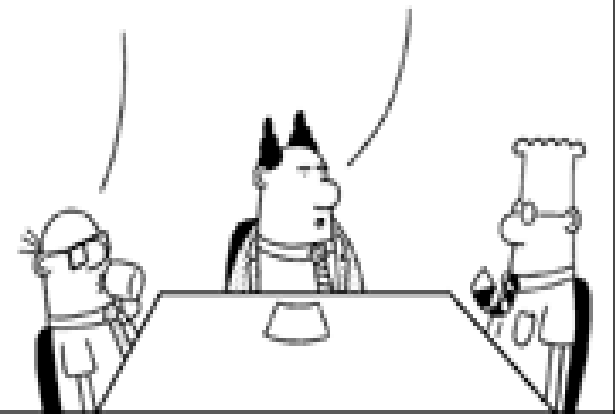
THAT MEANS NO MORE  
PLANNING AND NO MORE  
DOCUMENTATION. JUST  
START WRITING CODE  
AND COMPLAINING.



11-24-07 ©2007 Scott Adams, Inc./Dist. by UFS, Inc.

I'M GLAD  
IT HAS A  
NAME.

THAT  
WAS YOUR  
TRAINING.



# Kurssin tavoite

- Primäärinen 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, päivitetään kurssin aikana
- Aihepiirin hallitseville ihmisille suuri tarve, ks:
  - <http://www.projectmanagement.com/blog/Agility-and-Project-Leadership/6>

# Sisältö ja kurssimateriaali

- Sisältö ks.
  - kurssisivu  
<https://github.com/mluukkai/ohtu2013/wiki/Ohjelmistotuotanto-kevat-2013>
- ”teoria” perustuu mm. seuraaviin lähteisiin
  - Jonathan Rasmusson: The Agile Samurai
  - Henrik Kniberg: Scrum and XP from the trenches (ilmainen pdf)
  - James Shore: The Art of Agile development (osittain online)
- 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
  - Viikoilla 5 ja 6 pidetään (todennäköisesti) ainoastaan maanantain luento
- **Laskarit:**
  - Ohjelmointi/versionhallinta/konfigurointitehtäviä
  - Laskareista yhteensä 10 kurssipistettä
- **Coding dojo** viikolla 3, osallistumisesta 6 laskarirastia (käytännössä tämä tarkoittaa yhtä kurssipistettä)
- **Miniprojekti** viikoilla 2-6
  - 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
  - Ohjausta torstaina ja perjantaina klo 14-16 salissa BK107
  - Tehtävien palautus: ks. ensimmäinen laskari
- **Ylimääräiset opintopisteet**
  - **Versionhallinta 1 op:** jos et ole tehnyt kurssia, saat kurssin suoritetuksi tekemällä **kaikki** ohtun versiohallintatehtävät ja suorittamalla hyväksytysti miniprojektin
  - Tekemällä 95% kurssin muista kuin versionhallintaa käsittelevistä laskaritehtävistä, saat kurssista normaalin neljän opintopisteen sijaan **viisi opintopistettä**

# Miniprojekti

- **Viikolta 2 alkaen kurssilla siis ”miniprojekti”**
- miniprojektissa ohjelmoidaan hiukan, mutta pääosassa on ohjelmistoprosessin kurinalainen noudattaminen
- Projekti tehdään 3-5 hengen ryhmissä. Projekti tapaa asiakkaan viikoilla 2, 4, 5.
  - Kaikkien ryhmäläisten tulee olla asiakastapaamisissa (30-60 min) paikalla
- Viimeisellä viikolla miniprojektien demotilaisuus (2h)
- Lisää miniprojektista viikon päästä
- ”Vihaan Javaa. Onko miniprojekti pakko tehdä Javalla?”
- Käytetty ohjelmointikieli on kurssin suhteen täysin toisarvoinen asia. Ohjelmointiin liittyvien asioiden kannalta tärkeää on harjoitella mm. versionhallintaa, buildinhallintaa, yksikkötestaus- ja BDD-frameworkkeja, jatkuvaa integraatiota ja IoC-containerien käyttöä. Jos tämä kaikki infra löytyy omasta suosikkikielystä ja on halukas näkemään ekstravaivaa, voi käyttää mitä kieltä vaan.



# 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
  - Software design
  - Software construction
  - Software testing
  - Software maintenance
  - Software configuration management
  - Software engineering management
  - Software engineering process
  - 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. OhMa:n/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 nähden ”triviaaleja”
  - Ohjelmointi konekielellä
  - Usein sovelluksen käyttäjä ohjelmoi itse ohjelmansa
- Vähitellen ohjelmistot alkavat kastaa ja kehitettiin korkean tason ohjelmointikieliä (Fortran, Cobol, Algol)
- Pikkuhiljaa homma alkaa karata käsistä:
  - 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

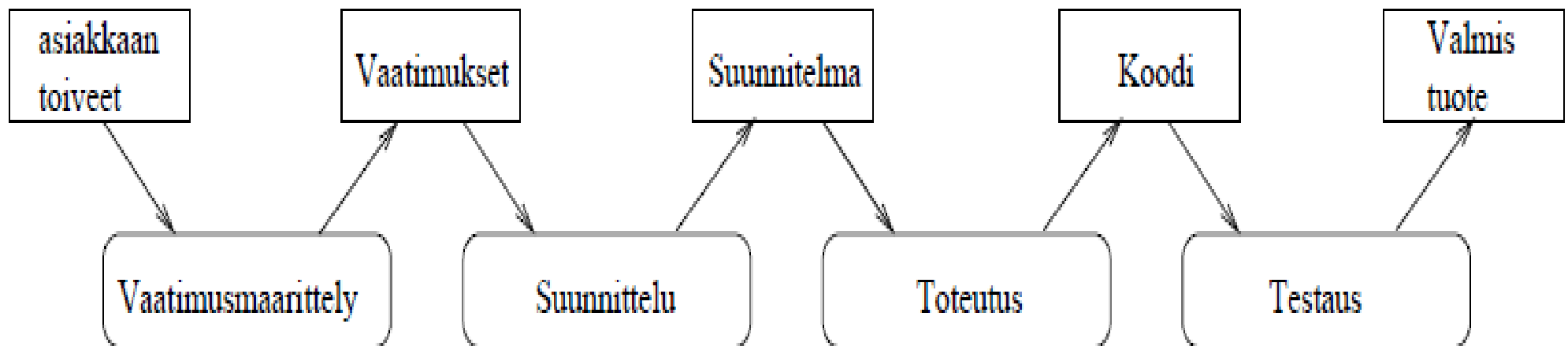
- Termi **Software crisis** lanseerataan kesällä 1968
  - The term was used to describe the impact of rapid increases in computer power and the complexity of the problems that could be tackled. In essence, it **refers to the difficulty of writing correct, understandable, and verifiable computer programs**. 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.

# 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 insinööritö
- Eli kuten esim siltojen rakentamisessa, tulee ensin rakennettava artifakti määritellä ja suunnitella (design) aukottomasti, tämän jälkeen rakentaminen (construction) on triviaali vaihe

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

- Paradoksaalista kyllä, Royce *ei suosittale* artikkelissaan suoraviivaisen lineaarisen mallin käyttöä, vaan esittelee mallin, jossa järjestelmästä tehdään ensin prototyyppi ja lopullinen suunnittelu tehdään vasta prototyyppiin 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

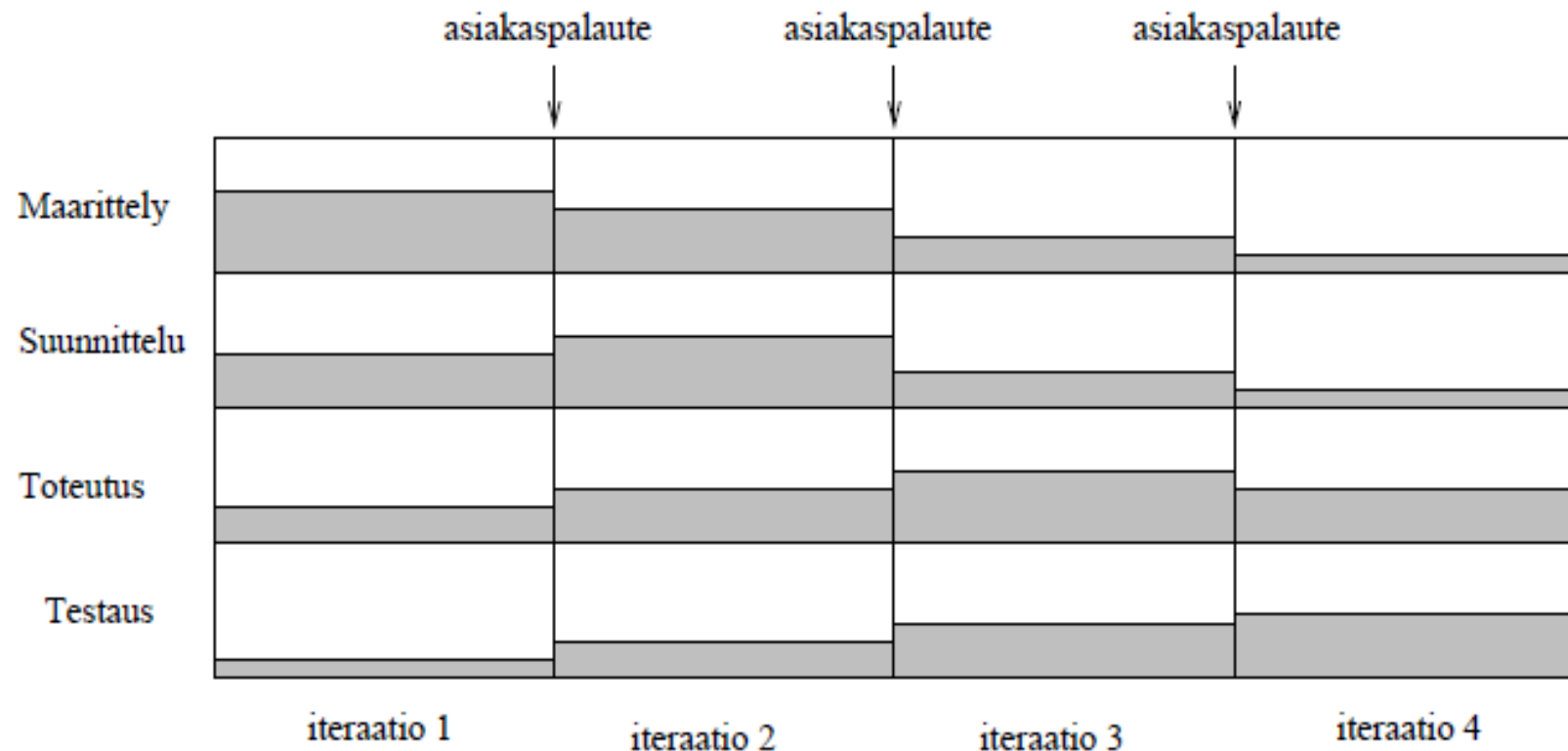


# Vesiputousmalli eli lineaarinen malli eli Plan based process tai Big Design Up Front

- Vesiputousmalli perustuu vahvasti siihen että eri vaiheet ovat erillisten tuotantotiimien tekemiä
  - Tämän takia kunkin 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 suositteli artikkelissaan ohjelmien tekemistä kahdessa iteraatiossa
  - Roycen mukaan ensin kannattaa tehdä prototyyppi ja vasta siitä saatujen kokemusten valossa suunnitellaan ja toteutetaan lopullinen ohjelmisto

# Iteratiiviset prosessimallit

- Iteratiivinen tapa alkoi yleistyä 90-luvulla (mm. spiraalimalli, prototyypimalli, Rational Unified process)
  - Itseasiassa iteratiivinen ohjelmistokehitys on paljon vanhempi idea kun lineaarinen malli, ks. <http://c2.com/cgi/wiki/wiki?HistoryOfIterative>
- Iteratiivisissa prosessimalleissa ohjelmistoja tehdään yleensä myös inkrementaalisesti, eli jokaisen iteraation aikana ohjelmistoon lisätään uusia ominaisuuksia



# Lineaarisen mallin ongelmia

- Martin Fowlerin artikkeli The New Methodology käsittelee laajalti lineaarisen mallin ongelmia
  - ks. <http://martinfowler.com/articles/newMethodology.html>
- 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 ohjelmointi on rinnastettu triviaalina pidettyyn ”rakentamisvaiheeseen” ja kaiken haasteen on ajateltu olevan määrittelyssä ja suunnittelussa
  - Tätä rinnastusta on kuitenkin ruvettu kritisoimaan, sillä ohjelmistojen suunnittelu sillä tarkkuudella että suunnitelma voidaan muuttaa suoraviivaisesti koodiksi on osoittautunut mahdottomaksi
- Onkin esitetty että perinteisen insinööritiedeanalogian triviaali rakennusvaihe ei ohjelmistoprosessissa olekaan ohjelmointi, vaan ohjelmakoodin kääntäminen eli että **ohjelmakoodi on itseasiassa ohjelmiston lopullinen suunnitelma** siinä mielessä kuin insinööritieteet käsittävät suunnittelun (design)
  - ks. <http://www.bleading-edge.com/Publications/C++Journal/Cpjour2.htm>

# 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 pienten ja keskisuurten ohjelmistojen tekoon ne osoittautuivat usein turhan jäykiksi
- Perinteisissä prosessimalleissa on pyritty työtä tekevän yksilön merkityksen minimoimiseen
  - 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 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
- Molempiin, erityisesti Scrumiin tutustutaan kurssin aikana
- Viime aikoina on puhuttu paljon siitä, että ohjelmistojen tekemisen rinnastaminen perinteiseen insinööriyöhön eli koko termi *Software engineering* metaforana väärä
- Ohjelmistojen tekemistä on alettu rinnastamaan käsityöläisyyteen ja on syntynyt ns. **Software craftsmanship** ”liike”, jolla on jopa oma manifestinsä, ks.
  - <http://manifesto.softwarecraftsmanship.org/>
  - <http://ofps.oreilly.com/titles/9780596518387/introduction.html>



# Ohjelmistokehityksen työkalut

- Nyt on aika siirtyä teoriasta käytäntöön ja tarkastella alustavasti muutamaa ohjelmistokehityksen käytännön työkalua
  - Versionhallinta: git
  - Testaus: JUnit
  - Projektin riippuvuuksienhallinta ja kääntäminen: Maven
  - CI- ja Build-palvelinohjelmisto: Jenkins

# Versionhallinta – Git

- ks. [http://jamesshore.com/Agile-Book/version\\_control.html](http://jamesshore.com/Agile-Book/version_control.html)
- Versionhallinta välttämätön monen hengen projektissa:
  - 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
    - Esim. voidaan palauttaa tiedostosta sen edellisen päivän tilanne
- Oikeastaan yhden hengen pieniäkään ohjelmointiprojekteja ei ole järkevää tehdä ilman versionhallintaa
- Mitä versionhallintaan talletetaan?
  - Jos mahdollista, **kaikki ohjelmistoon liittyvä**: ohjelmakoodi, dokumentit, kirjastot, konfiguraatietiedostot, jopa työkalut

# Versionhallinta – Git

- Kurssilla käytössä Git
  - Linus Torvaldsin kehittämä hajautettu versionhallinta (älä välitä vielä tästä termistä, sen merkitys selviää myöhemmin)
  - Tämän hetken eniten käytetty versionhallinta, syrjäyttänyt 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 pikkuhiljaa laskareissa, muutama hyvä lähtökohta
  - <https://we.riseup.net/debian/git-development-howto>
  - <http://www.ralfebert.de/tutorials/git/>
  - <http://progit.org/book/>
- Git saattaa tuntua aluksi sekavalta. Peruskäyttö on kuitenkin hetken totuttelun jälkeen helppoa

# Testaus – JUnit

- Ohjelmiston kehittämisessä lähes tärkein vaihe on laadunvarmistus, laadun varmistuksen 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 tutustuminen aloittaa välittömästi
  - <https://wiki.helsinki.fi/display/ohma/JUnit>

# Projektin riippuvuuksienhallinta ja kääntäminen – Maven

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

# Projektin riippuvuushallinta ja kääntäminen – Maven

- What Maven is:
  - The answer to this question depends on your own perspective. **The great majority of Maven users are going to call Maven a “build tool”: a tool used to build deployable artifacts from source code.** Build engineers and project managers might refer to Maven as something more comprehensive: a project management tool. What is the difference? A build tool such as Ant is focused solely on preprocessing, compilation, packaging, testing, and distribution. A project management tool such as Maven provides a superset of features found in a build tool. **In addition to providing build capabilities, Maven can also run reports, generate a web site, and facilitate communication among members of a working team.**
  - Maven on laaja ja pelottavakin työkalu, väärin konfiguroituna painajaismainen, mutta oikein konfiguroituna ohjelmistokehittäjän unelma!
    - Maven mm. hoitaa riippuvuuksien (eli Javassa jar-tiedostojen) lataamisen ohjelmoijan puolesta
- Tutustumme kurssin aikana maveniin pikkuhiljaa
- Lue esim:
  - <https://www.ibm.com/developerworks/java/tutorials/j-mavenv2/>

# CI- ja Build-palvelinohjelmisto: Jenkins

- Käännöksen automatisoinin 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 repositorioa ja kun siellä huomataan muutoksia, kääntää käännöspalvelin koodin ja suorittaa sille testit
  - Käännöspalvelin raportoi havaituista virheistä
- Erillisen käännöspalvelimen avulla varmistetaan, että ohjelmisto toimii muuallakin kuin muutokset tehneen ohjelmistokehittäjän koneella
- Kurssilla käytämme Jenkins-nimistä käännöspalvelinohjelmistoa
  - Keskitetyn build-palvelimen käyttöön liittyy käsite **jatkuva integraatio** (engl. Continuous integration), palaamme tähän tarkemmin myöhemmin kurssilla