Ohjelmistotuotanto
Syksy 2017

Matti Luukkainen

assistentteina:
Juha-Pekka Moilanen, Aleksi Mustonen ja Tuomo Torppa

Luento 1

30.10.2017

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

- 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/ohjelmistotuotanto2017/wiki/Ohjelmistotuotanto-syksy-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** 1-3*2h viikossa, yleensä ma 12-14 ja ti 12-14 CK112
  - Viikolla 1 ja 2 luento myös to 16-18 CK114
  - *Viikolla 3 ei luentoja*
  - Viikon 4 maanantain luento klo 14-16 *(*tiistain luento normaalisti)
  - Viikoilla 5 ja 6 pidetään ainoastaan maanantain luento
  - Viikolla 7 ei 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
  - Viikoilla 1-3 laskareiden kuormittavuus on suurempi kuin loppukurssista
    - Viikon 1-3 laskareiden arvioitu työmäärä noin 8h/vko
    - Loppukurssista 4h/vko

- **Ylimääräiset opintopisteet**
  - **Versionhallinta 1 op**: jos et ole tehnyt kurssia, saat kurssin suoritetuksi tekemällä **kaikki** ohtun versiohallintatehtävät ja suorittamalla hyväksytysti miniprojektin

Miniprojekti

- **Viikolta 3 alkaen kurssilla siis ”miniprojekti”**
- **Kurssin läpäisyn edellytyksenä on hyväksytysti suoritettu miniprojekti**
- miniprojektissa ohjelmoidaan hiukan, mutta pääosassa on ohjelmistoprosessin kurinalainen noudattaminen
- Projekti tehdään 3-5 hengen ryhmissä. Ryhmä tapaa asiakkaan viikoilla 3-6
  - Kaikkien ryhmäläisten tulee olla asiakastapaamisissa (ensimmäisellä viikolla 90 min, muuten 30 min) paikalla
- Viimeisellä viikolla miniprojektien demotilaisuus (2h)
- Miniprojekteissa työskentelyyn tulee varata noin 6 tuntia viikossa
  - Miniprojektien aikana laskarit ovat hieman vähemmän kuormittavat kuin kurssin ensimmäisillä viikoilla
- Lisää miniprojektista parin viikon päästä

Miniprojektin hyväksilukeminen

- Miniprojektiin osallistuminen ei ole välttämätöntä jos täytät työkokemuksen perusteella tapahtuvan Ohjelmistotuotantoprojektin hyväksiluvun edellyttävät kriteerit
  - ks. kohta “Laaja suoritus” sivulta http://www.cs.helsinki.fi/opiskelu/tietotekniikka-alan-ty-kokemus-opinto suorituksena
  - jos “hyväksiluet” miniprojektin työkokemuksella, kerro asiasta välittömästi emailitse
- Jos hyväksiluet miniprojektin, joudut kuitenkin tekemään noin 5 sivuisen esseen. Tarkempia tietoja esseestä myöhemmin

Luennot – laskarit - miniprojekti

- Kurssin luennoilla 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 luennoilla
- Miniprojektin 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 https://www.computer.org/web/swebok

- Mikä on SWEBOK:
  - Ison komitean yritys määritellä mitä ohjelmistotuotannolla tarkoitetaan ja mitä osa-alueita siihen kuuluu
  - Uusin versio vuodelta 2014

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 nähden ”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 (wikipedian ohjelmistotuotanoon historiaa käsittelevästä 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

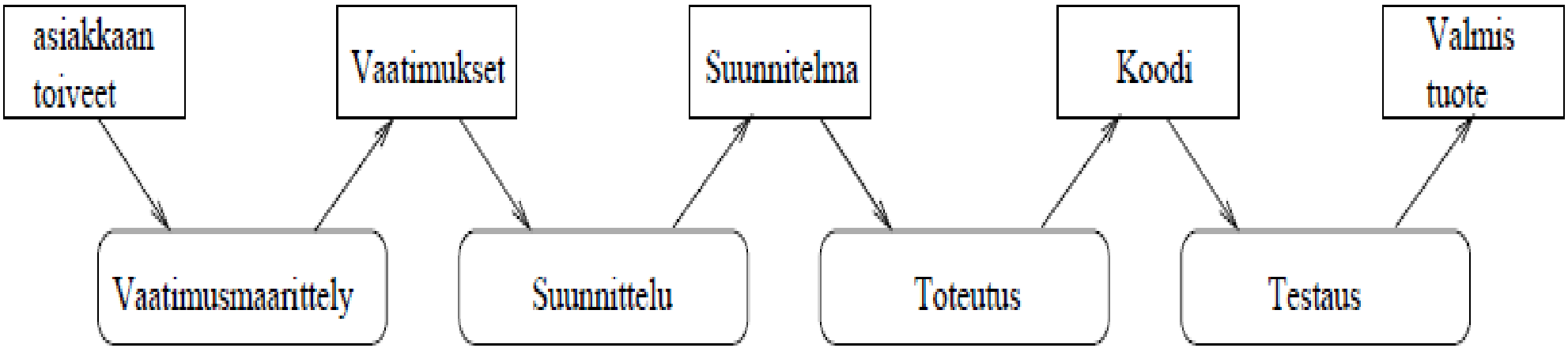
Software development as Engineering

- 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
- 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 prototyyppi ja lopullinen määrittely ja 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

Lineaarisen mallin ongelmia

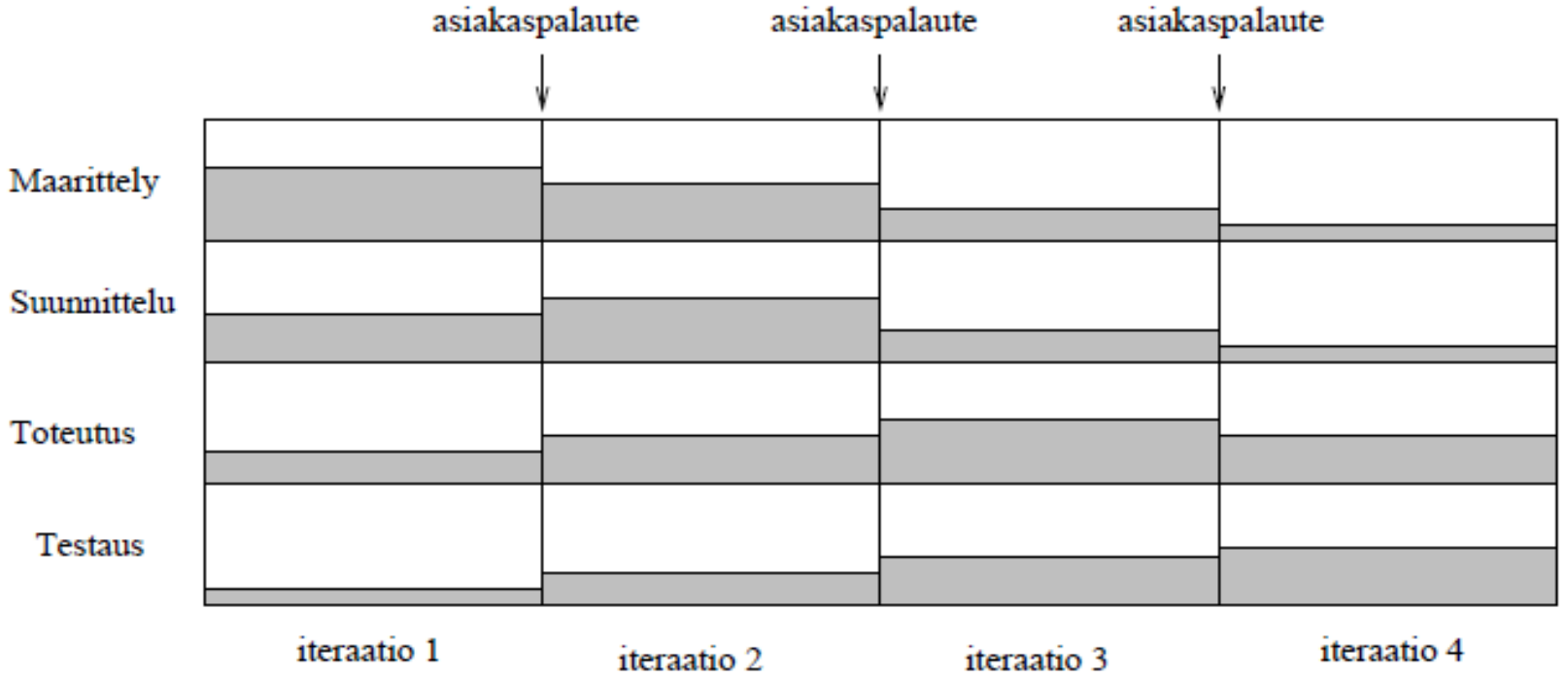
- Lineaarinen malli olettaa että ohjelmistotuotannon vaiheet tapahtuvat peräkkäin ja jokainen vaihe ainakin isoissa projekteissa eri ihmisten toimesta
- Vaatimukset kuitenkin usein muuttuvat matkan varrella:
  - Asiakas ei tiedä tai osaa sanoa mitä haluaa/tarvitsee
  - Asiakkaan tarve muuttuu projektin kuluessa
  - Asiakas alkaa haluta muutoksia kun näkee lopputuotteen
- Vaatimusmäärittelyn ja suunnittelun ja toteutuksen erottaminen on mahdotonta
  - Valittu arkkitehtuuri 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 kalliiksi 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 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

Iteratiiviset prosessimallit

- Lineaarisen mallin ongelmiin reagoinut *iteratiivinen* tapa tehdä ohjelmistoja alkoi yleistyä 90-luvulla (mm. spiraalimalli, prototyyppimalli, Rational Unified process
- Iteratiivisessa mallissa ohjelmistotuotanto jaetaan jaksoihin, eli *iteraatioihin*
  - Jokaisen iteraation aikana määritellään, suunnitellaan toteutetaan ja testataan ohjelmistoa, 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 Mercuryn* ohjelmisto kehitettiin iteratiivisesti (50-luvun lopussa)
  - *Avaruussukkuloiden* ohjelmisto tehtiin vesiputousmallin valtakaudella 70-luvun lopussa, mutta sekin kehitetiin lopulta iteratiivista prosessia käyttäen (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 pienten ja keskisuurten ohjelmistojen tekoon ne osoittautuivat usein turhan jäykiksi
- Perinteisissä 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

- Ketterät menetelmät on sateenvarjotermi useille ketterille prosessimalleille
- Näistä tunnetuimpia ovat:
  - Extreme programming eli XP
  - Scrum
- Molempiin, erityisesti Scrumiin tutustutaan kurssin aikana
- Ketterä ohjelmistotuotanto on ottanut vaikutteita 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 käsityöläisyyteen 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 kääntäminen: 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 versiota yhtä aikaa
    - Voi olla esim. tarve tehdä bugikorjauksia johonkin ohjelman jo aiemmin julkaistuun versioon
- Oikeastaan yhden hengen pieniäkään ohjelmointiprojekteja 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

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 pikkuhiljaa 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 tutustuminen aloittaa välittömästi
  - https://github.com/mluukkai/OTM2016/wiki/JUnit-ohje

Projektin riippuvuuksienhallinta ja kääntäminen – gradle

- ks. http://jamesshore.com/Agile-Book/ten_minute_build.html
- Ohjelmistoprojektissa ohjelman kääntäminen, testaaminen, paketointi suorituskelpoiseksi 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
- Riippuvuuksien hallinnan 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, Maven
- Tällä kurssilla tutustumme Gradleen

Projektin riippuvuuksienhallinta ja kääntäminen – 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 harjoitustyössä "buildaustyökaluna" mavenia.
- Gradle on uuden sukupolven buildaustyökalu, 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 pikkuhiljaa laskareissa

CI- ja Build-palvelinohjelmisto: Travis-ci

- 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 repositoriota 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 pilvessä toimivaa Travis-nimistä build-palvelinohjelmistoa
  - Keskitetyn build-palvelimen käyttöön liittyy käsite **jatkuva integraatio** (engl. Continuous integration), palaamme tähän tarkemmin myöhemmin kurssilla