Ohjelmistotuotanto

Luento 9

28.11.2017

Komponentti/oliosuunnittelusta

Kertausta viime viikolta

- Erityisesti ketterissä menetelmissä tarkempi olio/komponenttisuunnittelu tapahtuu usein vasta ohjelmoitaessa
- Ohjelmistosuunnitteluun ei ole mitään helposti noudatettavaa menetelmää
  - enemmän ”taidetta kuin tiedettä”, kokemus ja hyvien käytänteiden opiskelu toki auttaa
- 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 hidastua
- Sisäisesti laadukkaan koodin tekemistä edesauttaa huomion kiinnittäminen seuraaviin *laatuattribuutteihin*
  - Kapselointi
  - Koheesio
  - Riippuvuuksien vähäisyys
  - Toisteettomuus
  - Testattavuus
  - Selkeys

Suunnittelumalleja

- Suunnittelumallit tarjoavat hyviä kooditason ratkaisuja siitä, miten koodi kannattaa muotoilla, jotta siitä saadaan sisäiseltä laadultaan hyvää
- Kurssin itseopiskelumateriaalissa tutustutaan seuraaviin suunnittelumalleihin
  - Factory
  - Strategy
  - Command
  - Template method
  - Komposiitti
  - Proxy
  - Model view controller
  - Observer
- Sunnittelumallien soveltamista harjoitellaan viikon 5-7 laskareissa
- Itseopiskelumateriaali löytyy osoitteesta https://github.com/mluukkai/ohjelmistotuotanto2017/blob/master/web/oliosuu

Tekninen velka

- Koodi ja oliosuunnittelu ei ole aina hyvää, ja joskus on jopa asiakkaan kannalta tarkoituksenmukaista 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-levison
- 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
- 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

Tekninen velka

- Vastaavasti joskus voi lyhytaikaisen teknisen velan ottaminen olla järkevää tai jopa välttämätöntä
  - Esim. voidaan saada tuote nopeammin markkinoille tekemällä tietoisesti huonoa designia, joka korjataan myöhemmin
- 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 inadverent*: ”what is layering”?*
  - Prudent and inadverent: ”*now we know how we should have done it*”
  - Prudent and deliberate: *”we must ship now and will deal with consequences”*
  - http://martinfowler.com/bliki/TechnicalDebtQuadrant.html
- Teknisen velan takana voi siis olla monenlaisia syitä, esim. holtittomuus, osaamattomuus, tietämättömyys tai tarkoituksella tehty päätös

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 tha**t 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 monen tasoisia
- 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 refaktorointi

- Lääke koodihajuun on *refaktorointi* eli muutos koodin rakenteeseen joka kuitenkin pitää koodin toiminnan ennallaan
- Erilaisia koodin rakennetta parantavia refaktorointeja on lukuisia
  - ks esim. http://sourcemaking.com/refactoring
- Muutama käyttökelpoinen nykyaikaisessa kehitysympäristössä (esim NetBeans, Eclipse, IntelliJ) automatisoitu refaktorointi:
  - **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

- Refaktoroinnin melkein ehdoton edellytys on kattavien testien olemassaolo
  - Refaktoroinninhan on 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

Lean

Taustaa

- Kurssin aikana on jo muutamaan kertaan sivuttu käsitettä *Lean*, katsotaan nyt hieman tarkemmin mistä on kysymys
- Lean on syntyisin Toyotan tuotannon ja tuotekehityksen menetelmistä
- Toisen maailmansodan jälkeen Japanissa oli suuri jälleenrakennuksen buumi, mutta pääomaa ja raaka-aineita oli saatavissa niukalti
  - Havaittiin, että laadun parantaminen nostaa tuottavuutta: mitä vähmmän tuotteissa ja tuotantoprosesseissa on virheitä ja ongelmia, sitä enemmän tuottavuus kasvaa, ja se taas johtaa markkinaosuuden kasvuun ja sitä kautta uusiin työmahdollisuuksiin
  - Japanilaisiin yrityksiin tuli vahva laatua korostava kulttuuri
  - Resurssien niukkuus johti siihen, että Toyota kehitti ns. *Just In Time (JIT)* -tuotantomallin, missä ideaalina oli aloittaa tuotteen valmistus vasta kun ostaja oli jo tilannut tuotteen
    - Vastakohtana tälle on perinteinen massatuotanto, missä tehdään paljon tuotteita varastoon
  - Pyrkimys oli saada tuote tilauksen jälkeen mahdollisimman nopeasti kuluttajalle: lyhyt *läpimenoaika (cycle time)* tilauksesta toimitukseen

JIT-tuotanto

- JIT-tuotantomallista oli monia hyötyjä
  - Asiakkaiden muuttuviin tarpeisiin oli helppo valmistautua toisin kuin massatuotannossa, missä varastoon tehdyt tuotteet oli saatava myydyksi vaikka ne eivät olisi enää asiakkaan mieleen
  - Koska tuotteen läpimenoaika (tilauksesta asiakkaalle) oli lyhyt, laatuongelmat paljastuivat nopeasti
    - Toisin kuin mahdollisesti kuukausia varastossa olevilla tuotteilla
- Massatuotanto pyrki optimoimaan yksittäisten työntekijöiden ja koneiden työpanosta (ideaalina että koneiden käyttöaste on koko ajan 100%)
- Toyotan JIT-tuotantomallissa optimoinnin kohteeksi tuli tuotteen läpimenoaika
  - Pyrittiin eliminoimaan kaikki mahdollinen *jäte tai hukka* (waste), joka ei edesauttanut työn (eli tuotannon alla olevan tuotteen) nopeampaa *virtausta (flow)* tilauksesta asiakkaalle
  - Virtausta estäviin tekijöihin, esim. laatuongelmiin puututtiin heti
    - Käytännössä jokainen työntekijä oli oikeutettu ja velvollinen pysäyttämään tuotantolinjan havaitessaan ongelmia
- Työntekijöitä kunnioittava, kuunteleva ja vastuuttava (empowering) tuotannon optimoimisen kulttuuri
  - Toimintatapojen kaikilla tasoilla tapahtuva jatkuva parantaminen

Toyota production system (TPS)

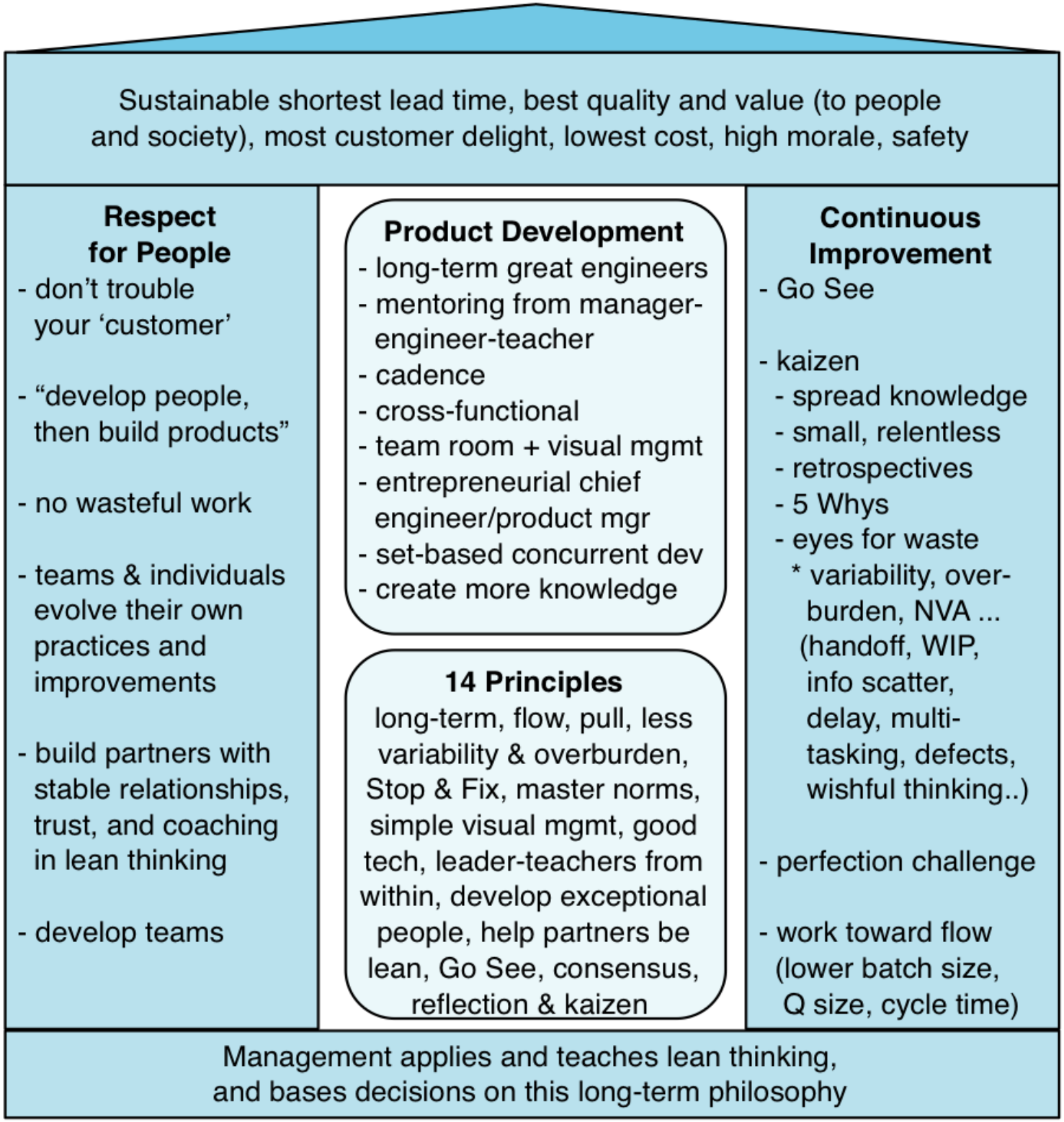
- Vuodesta 1965 alkaen Toyota alkoi kutsua toimintatapaansa Toyota Production Systemiksi
- Ensimmäiset englanninkieliset julkaisut aiheesta ovat vuodelta 1977
- Toyotan menestys herättää kiinnostusta länsimaissa ja MIT:in tutkijat alkavat 1980-luvun lopussa kartoittamaan ja dokumentoimaan tarkemmin Toyotan tuotantojärjestelmää
  - Tutkijat lanseeraavat nimikkeen *lean (eli virtaviivainen) tuotanto*
  - Vuonna 1990 ilmestyi kirja *The Machine That Changed the World,* joka toi leanin laajaan länsimaalaiseen tietoisuuteen
  - Toyota alkaa 2000-luvulla käyttämään tuotantojärjestelmästään myös sisäisesti nimitystä lean
- Viimeisen 25 vuoden aikana on ilmestynyt suuri määrä kirjoja, jotka kuvaavat Toyotan tuotantojärjestelmää, eräs kuuluisimmista ja vaikutusvaltaisimmista näistä on Jeffrey Likerin The Toyota Way (2001)

Lean tuotanto ja tuotekehitys

- Alun perin lean oli Toyotalla autojen tuotantoa (production) optimoiva toimintatapa, sittemmin leania on ruvettu hyödyntämään myös tuotekehityksessä (development)
  - Tuotanto ja tuotekehitys ovat luonteeltaan hyvin erilaisia ja niihin sovellettavat lean-käytänteet eroavatkin paikoin
- Leania on sovellettu lukuisille eri aloille, ohjelmistotuotantoon sen lanseerasi 2003 ilmestynyt Mary ja Tom Poppendieckin kirja *Lean software development, an agile toolkit*
  - Klassikon asemastaan huolimatta kirja on jo paikoin vanhentunut ja tulkitsee leania osin melko suppeasti
  - Scrumin kehittäjät Ken Schwaber ja Jeff Sutherland tunsivat lean-ajattelun, ja monet Scrumin piirteet ovat saaneet vaikutteita leanista
- Leania on ruvettu soveltamaan yhä suurempaan määrään asioita, aina terveydenhoidosta, pankkitoimintaan
- Lean tai mitä erilaisemmat lean-nimikkeen alla olevat (ja myytävät) asiat ovatkin alkaneet elämään omaa, Toyota producton systemistä erillistä elämäänsä ja nykyään on välillä vaikea sanoa tarkemmin mistä on kyse kun joku puhuu leanista

Lean production-delvelopment-thinking

- Käsittelemme nyt tarkemmin leania Craig Larmanin ja Bas Vodden mainioon kirjaan *Scaling Lean & Agile Development* perustuen
  - Leania esittelevä luku osoitteessa http://www.leanprimer.com/downloads/lean_primer.pdf
  - Luku esittelee nimenomaan Toyota production systemin modernia muotoa
- Leania havainnollistetaan useissa lähteissä *lean thinking houseksi* nimitettävänä kaaviona joka on esitelty seuraavalla kalvolla
- Kaavio havainnollistaa, että leanilla on
  - tavoite (goal)
  - perusta (foundation)
  - kaksi peruspilaria (pilars) ja
  - joukko näitä tukevia periaatteita (14 principles ja product development -periaatteet)
- Näiden lisäksi on olemassa joukko leania tukevia *työkaluja*
  - Työkaluista kuuluisin lienee kurssinkin aikana mainittu Kanban


Sustainable shortest lead time, best quality and value (to people and society), most customer delight, lowest cost, high morale, safety
Respect for People
- don't trouble your 'customer'
- "develop people, then build products"
- no wasteful work
- teams & individuals evolve their own practices and improvements
- build partners with stable relationships, trust, and coaching in lean thinking
- develop teams
Product Development
- long-term great engineers
- mentoring from manager-engineer-teacher
- cadence
- cross-functional
- team room + visual mgmt
- entrepreneurial chief engineer/product mgr
- set-based concurrent dev
- create more knowledge
14 Principles
long-term, flow, pull, less variability & overburden, Stop & Fix, master norms, simple visual mgmt, good tech, leader-teachers from within, develop exceptional people, help partners be lean, Go See, consensus, reflection & kaizen
Continuous Improvement
- Go See
- kaizen
- spread knowledge
- small, relentless
- retrospectives
- 5 Whys
- eyes for waste
* variability, over-burden, NVA ... (handoff, WIP, info scatter, delay, multi-tasking, defects, wishful thinking..)
- perfection challenge
- work toward flow (lower batch size, Q size, cycle time)
Management applies and teaches lean thinking, and bases decisions on this long-term philosophy

Lean: tavoite ja perusta

- Leanin tavoite (**goal**)
  - *Sustainable shortest lead time, best quality and value (to people and society), most customer delight, lowest cost, high morale, safety*
  - Tavoitteena saada aikaan pysyvä nopea tapa edetä ”ideasta asiakkaalle myytyyn tuotteeseen” (minimaalinen läpimenoaika)
  - siten että tämä tapahtuu työntekijöitä ja yhteistyökumppaneita riistämättä, ylläpitäen korkea laatutaso ja asiakastyytyväisyys
- Leanin perusta (**foundation**)
  - *Management applies and teaches lean thinking, and bases decisions on this long-term philosophy*
  - Jotta tavoite on mahdollinen, tulee taustalla olla syvälle yrityksen kaikille tasoille juurtunut pitkälle tähtäävä lean-toimintatapa, mitä johtajat soveltavat ja opettavat alaisille
- Lean rakentuu kahden **peruspilarin** varaan
  - **jatkuvan parantaminen** (continuous improvement)
  - **ihmisten kunnioittaminen** (respect for people)

Leanin peruspilarit

- **jatkuvan parantaminen** (continuous improvement)
  - Defines Toyota's basic approach to doing business
  - The true value of continuous improvement is in creating an atmosphere of continuous learning and an environment that not only accepts, but actually embraces change
  - The root of the Toyota Way is to be dissatisfied with the status quo; you have to ask constantly, "Why are we doing this?"
  - Such an environment can only be created where there is respect for people
- **ihmisten kunnioittaminen** (respect for people)
  - Includes concrete actions and culture within Toyota
  - not making people do wasteful work but instead real teamwork
  - mentoring to develop skillful people
  - humanizing the work and environment, safe and clean environment, and philosophical integrity among the management team
- Leanin tavoitetta ja peruspilareja tukevat konkreettisemmat toimintaa ohjaavat **lean-periaatteet** (lean principler) joita käsittelemme pian

Jatkuva parantaminen – arvo ja jäte

- TPS:n kehittäjä Taiichi Ohno kuvaa jatkuvan parantamisen periaatetta seuraavasti:
  - All we are doing is looking at the time line, from the moment the customer gives us an order to the point where we collect the cash
  - And we are reducing the time line by **reducing the non-value-adding wastes**
- Keino päästä nopeampaan tuotantosykliin on siis eliminoida arvoa (value) tuottamattomia asioita eli jätettä (waste)
- Mitä lean tarkalleen ottaen tarkoittaa arvolla ja jätteellä?
- Arvolla tarkoitetaan niitä asioita ja työnteon vaiheita, mistä *asiakas* on valmis maksamaan, eli mitkä tuottavat asiakkaalle hyötyä
- Jätteellä taas tarkoitetaan kaikkea tuotantoon liittyvää, mikä ei tuota asiakkaalle arvoa
- Lean tunnistaa alunperin 7 lähdettä jätteelle (muda)
  - Over-prodution, In-process inventory, extra processing, transportation, motion, waiting, defects

Lean-jäte ohjelmistotuotannon näkökulmasta

- **Over-prodution** ylituotanto
  - Tuotteita tulee valmistaa ainoastaan siinä määrin mitä asiakas on niitä tilannut, eli ei kannata tehdä varastoon
  - Ohjelmistojen ylimääräiset toiminnallisuudet voidaan ajatella olevan ylituotantoa
    - tutkimuksien mukaan 64% ohjelmistojen toiminnallisuuksista on joko ei ollenkaan tai ainoastaan erittäin harvoin käytettyjä
- **In-process inventory** välivarastointi
  - Tähän kategoriaan kuuluu osittain tehty työ ja sen säilöminen
  - Ilmentymiä ohjelmistotuotannossa
    - Tarkka vaatimusmäärittely ominaisuuksille joita ei vielä toteuteta
    - Valmis koodi mikä ei ole vielä testattu tai otettu käyttöön
    - Koodi joka toteuttaa asiakkaan ehkä tulevaisuudessa haluamia toiminnallisuuksia
- **extra processing** liikatyö
  - prosessin pakottamat turhat työvaiheet
  - ”Pyörän keksiminen uudelleen”

Lean-jäte ohjelmistotuotannon näkökulmasta

- **Transportation** tarpeeton materiaalin siirtely
  - Ohjelmistojen kontekstissa ns ”handoff”, eli jos ohjelmistotuote esim.
    - määritellään ja toteutetaan erillisten tiimien toimesta tai
    - toteutetaan ja viedään tuotantoon erillisten tiimien toimesta
- **Motion** työntekijöiden tarpeeton liikkuminen
  - Ohjelmistotuotannossa *task switching*: eli liian nopea vaihtelu eri työtehtävien välillä, esim. työskentely yhtäaikaa monessa projektissa
- **Waiting** turha odotus
  - Esim. aika, joka joudutaan odottamaan että yrityksen johto hyväksyy vaatimusmäärittelyn, testaajat ehtivät testaamaan ohjelman uuden version, ylläpito vie sovelluksen uuden version tuotantoon, joku mergeää pullrequestin..
- **Defects** viat
  - Järjestelmässä on väkisinkin jossain tuotannon vaiheessa vikoja, mutta testaaminen ja vikojen havaitseminen vasta tuotannon loppuvaiheessa on asiakkaan arvon tuottamisen kannalta erittäin epäoptimaalista

Lean-jäte ohjelmistotuotannon näkökulmasta

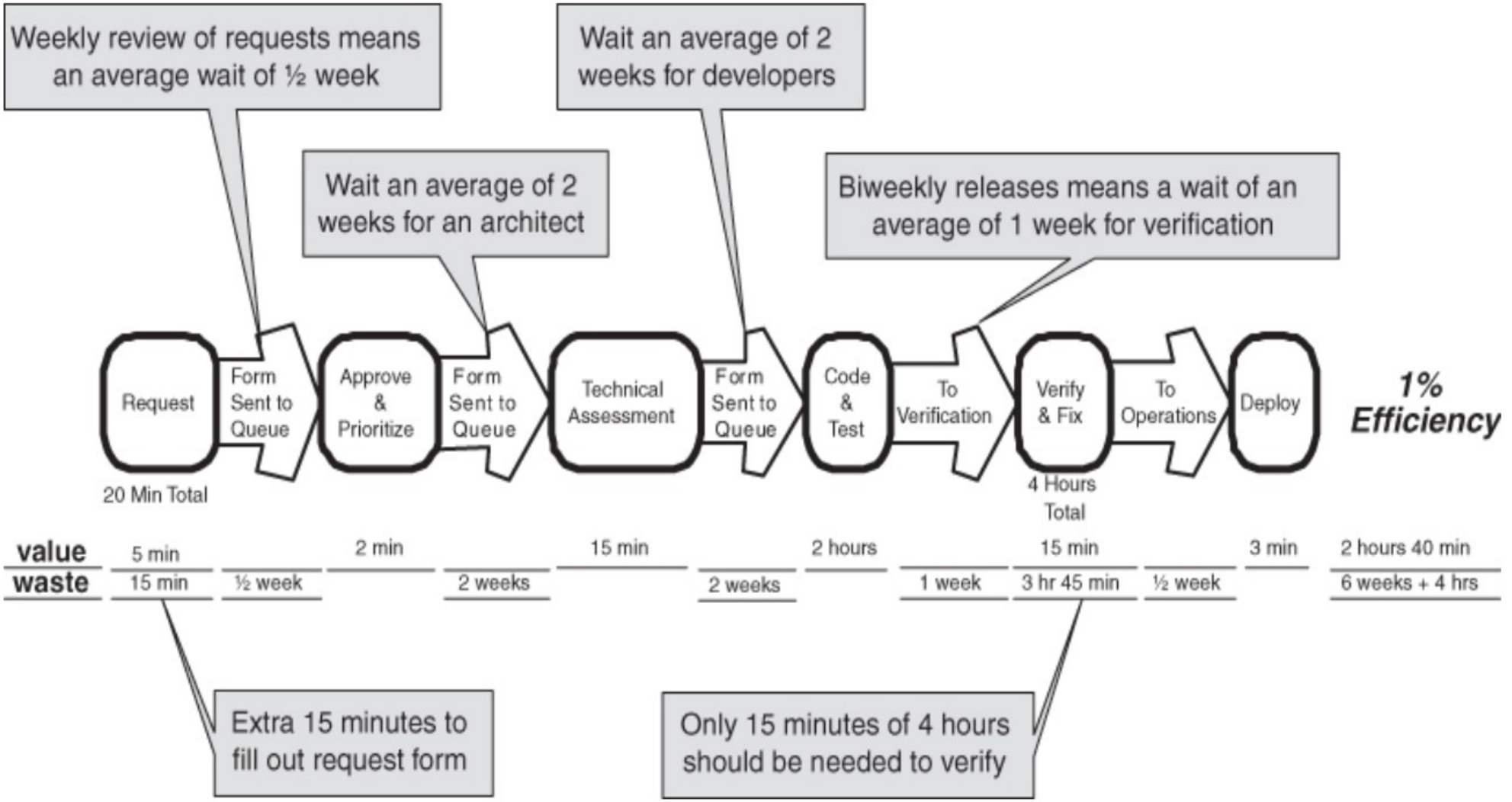
- Myöhemmin on ehdotettu alkuperäisten jätteiden lisäksi uusia, mm.
  - *Under-realizing people's potential and varied skill, insight, ideas, suggestion*
- Lean jakaa jätteet kahteen eri luokkaan
  - **Pure waste**
    - These are non-value adding actions that in principle can and should be eliminated now
    - For example, if a group is participating in the waste of defects with test-last and correct-last, that can be replaced with acceptance test-driven development
  - **Temporarily necessary waste**
    - These are things we must do because of some constraint, but still correctly classify as waste.
    - Eg. banks must do "regulatory compliance" activities that are required by law, but they are not value-adding actions in the eyes of the paying customer
    - These are temporary wastes because in the future it may be possible to change things so they are no longer necessary

Jatkuva parantaminen

- Jatkuvassa parantamisessa on siis tarkoitus optimoida toimintaa eliminoimalla asiakkaalle arvoa tuottamatonta jätettä
- Jatkuvasta parantamisesta käytetään usein sen japaninkielista nimitystä **kaizen**
- Kaizen on kaikkia työntekijöitä koskeva kattava toimintafilosofia:
  - As a mindset, it suggests “My work is to do my work and to improve my work” and “continuously improve for its own sake.”
- Kaizen on myös konkreettinen tapa toimia
  - choose and practice techniques/processes “by the book” that the team and/or product group has agreed to try, until they are well understood and mastered
  - experiment until you find a better way, then make that the new temporary ‘standard’
  - repeat forever
- Kaizeniin liittyvä syklinen parannusprosessiin saattavat liittyä tasaisin väliajoin pidettävät tilaisuudet ”kaizen event”:it
  - Scrumin retrospektiivit ovat klassinen esimerkki kaizen eventeistä

Value stream mapping

- Jätteen kartoittamisessa käytetään usein *value stream mappingia*
  - Ideana on kuvata tuotteen kulku käytetyn prosessin työvaiheiden läpi ja visualisoida tuotteelle arvoa tuottavat työvaiheet suhteessa tuotteen koko valmistuksen elinkaareen



Perimmäisen syyn analyysi - five whys

- Jos esim value steream mapping paljastaa prosessista jätteitä, eli arvoa tuottamattomia asioita (ks 7 jätetyypin lista), tulee niistä hankkiutua eroon
- Kaizenissa ei kuitenkaan ole tarkoitus lääkitä pelkkää oiretta, vaan tehdä jätteelle perimmäisen syyn ongelma (**root cause analysis**) ja pyrkiä näin kestävämpiin ja vaikuttavampiin parannuksiin
- Eräs root cause analysis -tekniikka on ”five whys”, esim.
  - **Miksi** koodin valmistumisesta menee 1.5 viikkoa sen tuotantoon saamiseen?
  - QA-osaston on vielä varmistettava, että koodi toimii staging-ympäristössä
  - **Miksi?** Ohjelmoijilla ei ole aikaa testata koodia itse staging-ympäristössä
  - **Miksi?** Ohjelmoijilla on kiire sprintin tavoitteena olevien user storyjen tekemisessä
  - **Miksi?** Edellisten sprinttien aikana tehtyjen storyjen bugikorjaukset vievät yllättävän paljon aikaa
  - **Miksi?** Laadunhallintaa ei keritä koskaan tekemään kunnolla siinä sprintissä missä storyt toteutetaan
- Näin kysymällä toistuvasti *miksi* on mahdollista päästä ongelman perimmäisen syyn lähteille, eli sinne mitä korjaamalla jäte saadaan toivon mukaan eliminoitua

Leanin periaatteita

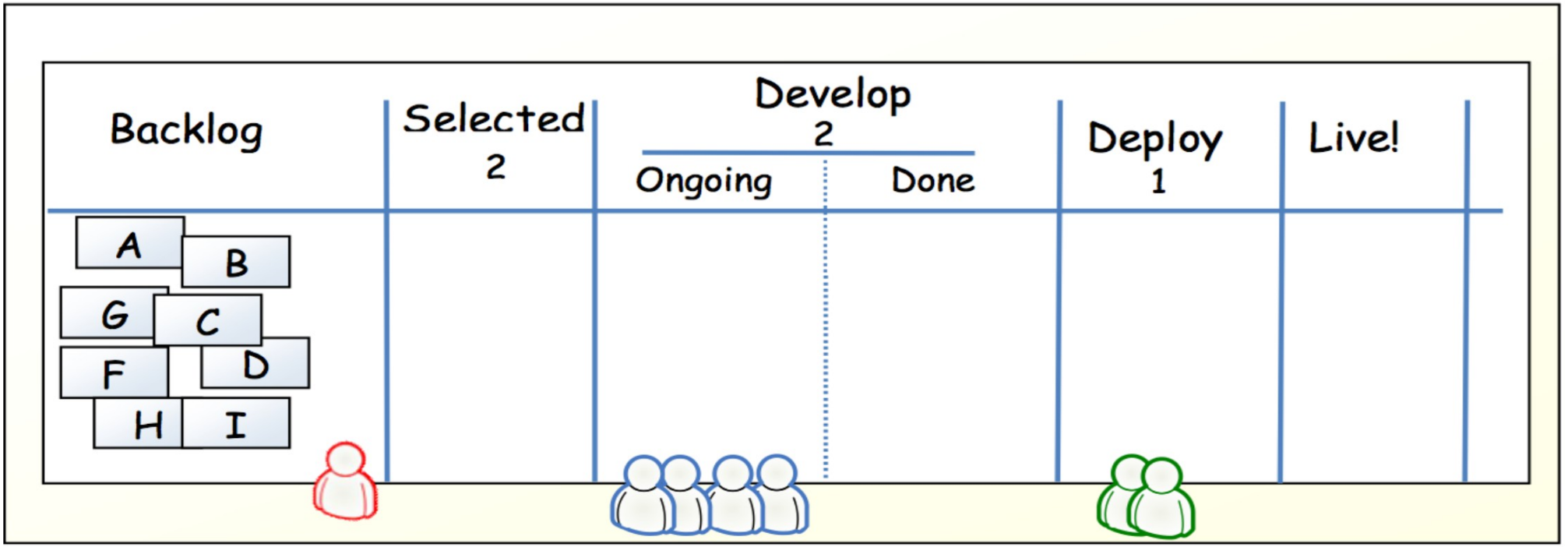
- Leanissa on siis tarkoitus optimoida aikaa, mikä kestää tuotteen suunnittelusta sen asiakkaalle toimittamiseen
- Arvo pyritään saamaan *virtaamaan* (flow) asiakkaalle ilman turhia viiveitä ja työvaiheita
  - Value stream map visualisoi arvon virtausta
- Leanin mekanismi virtauksen optimointiin on *pull-systeemien* käyttö
- Pull-systeemillä tarkoitetaan tuotannonohjaustapaa, missä tuotteita, tai tuotteiden tarvitsemia komponentteja tehdään ainoastaan asiakkaan tilauksen saapuessa
  - Näiden vastakohta on push-systeemi, missä tuotteita ja komponentteja tehdään etukäteen varastoon ja toivotaan sitten että tuotteet ja komponentit menevät kaupaksi
  - Esim. pizzeriat toimivat pull-periaatteen mukaan, pizza valmistetaan vasta kun asiakas tilaa sen. Unicafe taas on push-systeemi, lounaita tehdään varastoon ja toivotaan että ne menevät kaupaksi asiakkaille.
- Pull-systeemi toteutetaan usein kanbanin avulla
  - Kanban tarkoittaa *signaalikorttia*
  - Kanban toteuttaa *visuaalisen ohjauksen*, minkä avulla työntekijöiden on helppo tietää miten seuraavaksi tulee toimittaa

Kanban

- Kun asiakas tilaa tuotteen, viedään tilausta vastaava kanban-kortti tehtaalle
- Jos tuotteen valmistaminen edellyttää esim. viittä eri komponenttia, ”tilataan” komponentit niitä valmistavilta työpisteiltä viemällä niihin kunkin komponentin tilausta vastaava kanban-kortti
  - Jos komponenttien valmistus edellyttää jotain muita komponentteja, tilataan nekin samalla periaatteella
- Kun komponentti on valmis, viedään se tilaajalle, samalla kanban-kortti palautetaan tulevien tilauksien tekemistä varten
- Kanban-kortteja on käytössä vain rajallinen määrä, tällä kontrolloidaan sitä että liikaa työtä ei pääse kasautumaan mihinkään tuotannon vaiheeseen
- Näin kanbanin avulla ”vedetään” (pull) tarvittavat komponentit, sensijaan että komponentteja olisi etukäteen valmistettu varalta suuret määrät varastoon
- Varastoon tehdyt komponentit muodostaisivat riskin
  - Niitä ei välttämättä tarvittaisi jos tilauksia ei tulisi tarpeeksi
  - Jos komponenteissa olisi valmistusvika, saattaisi kestää kauan kunnes vika paljastuisi ja viallisia komponentteja olisi ehkä ehditty tekemään suuri määrä
- Käytännössä pull-periaatteella toimiva tuotanto saattaa ylläpitää pieniä välivarastoja saadakseen tuotteen valmistamiseen kuluvan sykliajan optimoitua

Kanban ohjelmistotuotannossa

- Kanban on otettu laajalti käyttöön myös ketterässä ohjelmistokehityksessä, luennon 4 kalvoissa puhutaan asiasta enemmän
  - Ohjelmistopuolen kanban on periaatteiltaan hieman erilainen
  - Toteutettavaa toiminnallisuutta, esim. user storyä vastaa kanban-kortti
  - Kortti kulkee eri työvaiheiden kautta
  - virtaus, eli yksittäisen storyn nopea valmistuminen saadaan aikaan rajoittamalla tietyissä työvaiheissa kesken olevan työn määrää WIP (work in progress) -rajoitteilla



Leanin periaatteita

- Jotta pull-järjestelmä toimii hyvin, eli asiakkaan arvo *virtaa* tasaisesti, on edullista jos eri työvaiheiden kestoon ei liity liikaa varianssia, tähän liittyy leanin periaate **level the work**
- Yksi varianssin aiheuttaja ovat viat. Leanin periaatteita ovatkin
  - Stop and fix
  - build quality in
- **Stop and fix** viittaa Toyotan vanhaan periaatteeseen, missä kuka tahansa on velvollinen pysäyttämään tuotantolinjan vian, esim. vaurioituneen komponentin havaitessaan

  vian perimmäinen syy (root cause) tulee selvittää mahdollisimman nopeasti ja pyrkiä eliminoimaan vian mahdollisuus tulevaisuudessa eli tuotantossa tulee olla laatu sisäänrakennettua **build quality in**
- Ohjelmistotuotannon käytänteistä Continuous integration ja automaattinen testaus voidaan nähdä suoraan stop and fix -ja build quality in -periaatteiden ilmentymänä

Leanin periaatteita

- Perinteisessä massatuotannossa keskitytään pitämään tuotantolaitteistot käynnissä maksimikapasiteetilla ja työntekijät koko ajan työllistettyinä
  - Yksittäisten työntekijöiden palkkauskin perustuu usein suorituskohtaisiin bonuksiin
- näin ajatellaan että tuotteiden yksikköhinta saadaan mahdollisimman alhaiseksi ja yrityksen tuottavuus maksimoituu
- Yksittäisten työntekijöiden ja koneiden tehokkuuden tarkastelun sijaan lean keskittyy arvoketjujen optimoinnin avulla järjestelmien kokonaisvaltaiseen kehittämiseen ja olettaa, että se on pidemmällä tähtäimellä yritykselle kannattavampaa (long term philosophy)
  - Yksittäisen koneen suuri käyttöaste voi olla *lokaalia optimointia*, joka voikin yrityksen kannalta olla jopa haitallista
    - Esim. valmistetaan paljon komponentteja, mitä ei lopulta tarvita
  - Keskittymällä arvon virtaukseen pyritään toiminnan parannuksessa ottamaan huomioon koko tuotantosysteemiä koskevat pullonkaulat
- Surullisen kuuluisia esimerkkejä lokaalista optimoinnista on paljon, mm. yliopistojen eri laitosten säästöt tilakustannuksista
  - Yliopiston rakennuksista maksama vuokra on edelleen sama vaikka jokin laitos ”säästää” jättämällä tiloja käyttämättä

Leanin periaatteita

- **Decide as late as possible**
  - Pull-systeemeissä ei ole tapana tehdä tuotantoon liittyviä päätöksiä (esim. miten paljon tuotetta ja sen tarvitsemia komponentteja tulee valmistaa) aikaisessa vaiheessa, vaan vasta tarpeen vaatiessa
  - Englanniksi tätä myöhäistä päätöksen tekemistä luonnehditaan myös seuraavasti **commit at the last responsible moment**, eli päätöksiä viivytetään, mutta ei kuitenkaan niin kauaa että viivyttely aiheuttaa ongelmia
- Kun päätös tehdään myöhään on tästä se merkittävä etu, että päätöksen teon tueksi on käytettävissä maksimaalinen määrä tietoa
  - Toisin kuin etukäteen tehtävissä päätöksissä mitkä ovat enemmän spekulatiivisia
- Kun päätökset on tehty, toimitaan pull-systeemin hengessä mahdollisimman nopeasti
  - **implement rapidly** tai **deliver as fast as possible**
  - näin arvo saadaan virtaamaan asiakkaalle ilman turhia viiveitä
- Mitä nopeammin arvo saadaan virtaamaan, sitä enemmän päätöksiä on mahdollista viivyttää ja päätökset voidaan tehdä entistä paremman tiedon valossa

Arvon virtaaminen ketterässä ohjelmistotuotannossa

- Edellisen kalvon periaatteiden soveltaminen näkyy selkeästi ketterässä ohjelmistotuotannossa
- Vaatimuksia hallitaan product backlogilla, joka on parhaassa tapauksessa DEEP
  - Detailed aproproately, emergent, estimated, prioritized
  - Tarkkoja vaatimuksia ei määritellä spekulatiivisesti vaan **at the last responsible moment**
  - Alhaisen prioriteetin user storyjä ei ole kovin tarkkaan määritelty
  - Kun product owner valitsee storyn seuraavaan sprinttiin toteutettavaksi määritellään storyn hyväksymäkriteerit ja suunnitellaan se toteutuksen osalta
  - Ja **deliver as fast as possible** eli toteutetaan valmiiksi seuraavan sprintin aikana
- Scrum voidaankin nähdä leanin mukaisena pull-systeeminä, missä jokaiseen sprinttiin otetaan kerrallaan asiakkaan edustajan viime hetkellä viimeistelemät tilaukset, jotka toteutetaan nopeasti, eli sprintin aikana
  - Arvo (eli toimivaksi asti toteutetut toiminnallisuudet) virtaavat asiakkaalle sprinttien määrittelemässä rytmissä

Arvon virtaaminen ketterässä ohjelmistotuotannossa

- Ketterässä ohjelmistotuotannossa on viime aikoina ruvettu tehostamaan arvon virtausta usein eri menetelmin
  - Alunperin Scrumin pyrkimys viedä uusia ominaisuuksia tuotantoon sprinteittäin
- Viime aikojen trendinä on ollut tihentää sykliä
  - Jatkuva tuotantoonvienti eli **continuous deployment** voi tarkoittaa sitä, että jopa jokainen commit johtaa tuotantoonvientiin
- Scrum rajoittaa kesken olevan työn määrää (joka on siis eräs lean waste) siten, että sprinttiin otetaan vaan tiimin velositeetin verran user storyjä
- Kaikissa konteksteissa, esimerkiksi jatkuvaa tuotantoonvientiä sovellettaessa aikarajoitettu sprintti ei ole mielekäs
- Paikoin onkin siirrytty ”puhtaampaan” pull-systeemiin, missä user storyjä toteutetaan yksi kerrallaan niin nopeasti kuin mahdollista
  - Kun tuotantokapasiteettia vapautuu, valitsee product owner tärkeimmän storyn
  - story määritellään, suunnitellaan ja sitten toteutetaan välittömästi alusta loppuun
  - Virtaus varmistetaan sillä, että yhtä aikaa työn alla ei ole kuin 1 tai korkeintaan muutama story
  - Luennolla 4 mainittu Scrumban-menetelmä toimii pitkälti juuri näin

Leanin johtajat

- Toyotalla useimmat uudet työntekijät koulutetaan huolellisesti perehtymään käytännön tasolla lean-ajattelun periaatteisiin
  - Useiden kuukauden koulutuksen aikana uudet työntekijät työskentelevät monissa eri työtehtävissä
  - heidät opetetaan tunnistamaan lean-jäte eri muodoissaan
  - tarkoituksena on sisäistää jatkuvan parantamisen (kaizen) mentaliteetti
- Johtamiskulttuurissa keskiössä on lean-ajattelun opettajina, mentoreina ja työn valmenajana toimivat johtajat/managerit
- Periaate **grow leaders** kuvaa Toyotan tapaa kasvattaa lean-toimintafilosofian sisäistäviä johtajia
  - Periaate *my manager can do my job better than me* kuvaa sitä, että johtajat ovat ovat firman sisällä eri työtehtävien kautta uusiin vastuisiin kasvavia ihmisiä, jotka hallitsevat myös työntekijöiden vastuulla olevan *hands on* -työn
  - Johtajat ovat ensisijaisesti toiminnan etulinjassa toimivia lean-käytänteiden opettajia ja mentoreita

Leanin johtajat

- Eräs tärkeä johtamisen periaate on **go see** (genchi genbutsu)
  - Työntekijöiden, erityisesti managerien tulee ”nähdä asiat omin silmin” eikä pelkästään istua työpöydän ääressä lukemassa muiden raportoimia faktoja
  - Tämä liittyy siihen ideaaliin, että johtajien oletetaan johtavat etulinjassa (**gemba**) eli siellä missä työ tosiasiallisesti tehdään
- Toyota production systemsin kehittäjän T. Ohnon sanoin:
  - You can’t come up with useful kaizen sitting at your desk... We have too many people these days who don’t understand the workplace. They think a lot, but they don’t see. I urge you to make a special effort to see what’s happening in the workplace. That’s where the facts are.
- Scrum masterin rooli on osin leanin ideaalien mukainen. Tosin kaikkiin scrum mastereihin ei päde periaate *my manager can do my job better than me*
  - Monissa ohjelmistoyrityksissä teknistä puolta johtajuudesta edustaa edustavat esim, *lead developer* - tai *senior developer* -nimikkeellä olevat kokeneemmat mentorin roolissa toimivat työntekijät

Lean tuotekehityen periaatteita

- Sovellettaessa leania *tuotantoon* (*production*), pääasiallisena fokuksena on toiminnan parantaminen jätettä eliminoimalla
- Sovellettaessa leania tuotannon optimoinnin sijaan *tuotekehitykseen* (*development)* esim. kokonaan uusien automallien suunnitteluun, nousee esiin uusia periaatteita
- Toyotalla periaatteena tuotekehityksessä on
  - *out-learn the competitors, through generating more useful knowledge and using and remembering it effectively*
- Fokukseen nousee toiminnan tehostamisen lisäksi *oppimisen kiihdyttäminen* (**amplify learning**)
- Kannattaa pyrkiä mahdollisimman arvokkaaseen tietoon (**high-value information),** mm. kiinnittämällä huomio asioihin, mihin sisältyy paljon epävarmuutta **(focus on uncertain things)**
  - Epävarmat ja suuren teknisen riskin sisältävät ideat tulee toteuttaa/testata nopeasti, niiden suhteen viivästyneellä tiedolla on korkea hinta (cost of delay)

Lean tuotekehityen periaatteita

- Eräs leanin mekanismi oppimisen nopeuttamiseen on **set based concurrent development**
  - Jos tarkoituksena on kehittää esim. uusi moottorin jäähdytysjärjestelmä, aletaan yhtä aika kehittämään useita vaihtoehtoisia ratkaisuja eri tiimien toimesta
  - Tasaisin väliajoin kehitettäviä ratkaisuja vertaillaan, ja osa niistä karsitaan
  - Lopulta parhaaksi osoittautuva ratkaisu valitaan käytettäväksi lopputuotteessa
- Set based -menetelmä on melko erilainen kuin useimmiten sovellettava iteratiivinen kehitysmenetelmä, missä lähtökohtana on yksi askeleittain paranneltava ratkaisu
  - Ohjelmistotuotannossa set based -menetelmää sovelletaan aika harvoin, lähinnä käyttöliittymäsuunnittelussa esittelemällä asiakkaalle aluksi useita rinnakkaisia ehdotelmia mahdollisesta käyttöliittymäratkaisusta
- Toyotalla tuotekehitystä johtaa **chief technical engineer**
  - Vastuussa sekä tuotteiden teknisestä että liiketoiminnallisesta menestyksestä
  - Kyseessä tyypillinen leanin etulinjassa toimiva johtaja joka tuntee tarkasti käytännön työn, mutta on myös erittäin lähellä asiakasta
  - Poikkeaa Scrumin Product Ownerista teknisen taustansa takia

Leanin soveltaminen eri aloille

- 90-luvulta alkaen lean on herättänyt maailmalla suurta kiinnostusta ja sitä on pyritty soveltamaan lähes kaikilla aloilla ohjelmistojen kehittäminen mukaan lukien
- Lean-periaatteet ovat olleet hyvin esim. Scrumin kehittäjien tiedossa ja vaikka Scrumin alkuperäiset lähteet eivät käytäkään leanin terminologiaa, on Scrumissa monin paikoin piirteitä leanista
  - Viime aikainen ketterien menetelmien kehitys on vienyt tiettyjä ideoita (mm. arvoketjun optimoimista user storyjen läpimenoaikoja minimoimalla) huomattavasti Scrumin ja ketterän alkuaikojen käytänteitä pidemmälle
  - Nykyään puhutaan paljon leanista ohjelmistokehityksestä
- Sekä ketterissä menetelmissä, että leanissa on sama fundamentaali periaate, toimintojen jatkuva kehittäminen
  - Rajanveto leanin ja ketterän välillä ei olekaan ollenkaan selvä ja oikeastaan täysin keinotekoista
  - Esim. Scrumin kehittäjät eivät ole tarkoittaneet Scrumia staattiseksi rakennelmaksi, jota noudatetaan kirjaimellisesti tästä ikuisuuteen, sellainen toiminta ei olisi ketteryyttä.
  - Ketteryyttä on läpinäkyvyyden mahdollistava toimintojen parantamiseen keskittyvä inspect-and-adapt-sykli
  - Käytännössä tämä on täsmälleen sama idea kuin leanin *kaizen*

Leanin soveltamisen vaikeus

- Leanin soveltamisessa on kohdattu myös paljon ongelmia
- Lean on ajattelumalli, joka on kehitetty Toyotan tarpeisiin, malli on jalostunut ja muuttunut aikojen kuluessa
  - on osin epäselvää miten Toyotan käytänteet siirretään eri aloille
- Valitettavan usein Lean tulkitaan mekanistisesti, keskittyen tiettyihin työkaluihin (esim. kanban ja value stream mapping) jättämättä leanin taustalla olevat periaatteet (jatkuva parantaminen ja ihmisten kunnioittaminen) huomioimatta
  - Saadaan ehkä aikaan hetkellisiä parannuksia tuotantoketjussa, mutta parannukset eivät välttämättä ole kauaskantoisia jos ne eivät vaikuta koko organisaation ajatteluun ja toimintatapoihin
- Kuten agile, myös lean ei ole joukko työkaluja vaan jatkuva toimintatapa, Toyotan CEO:n sanoin
  - *The root of the Toyota Way is to be dissatisfied with the status quo; you have to ask constantly, “Why are we doing this?”*
  - *In Toyota and in lean thinking, the idea is to repeat cycles of improvement experiments forever.*