Ohjelmistotuotanto

Luento 10

Oliosuunnittelu jatkuu

Java 8:n tuomia mahdollisuuksia

- Luennolla 8 tutustuimme jo hieman Java 8:n lambda-lausekkeiden ja stream-apin tarjamiin mahdollisuuksiin
  - ks. https://github.com/mluukkai/ohtu2014/blob/master/web/luento8.m
- Jatketaan Java 8:iin tutustumista
  - ks. https://github.com/mluukkai/ohtu2014/blob/master/web/luento10.

Tekninen velka

- Edellisen kahden luennon 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 priniciple, 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 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
  - http://msdn.microsoft.com/en-us/magazine/ee819135.aspx

Tekninen velka

- 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
- Tekniselle velalle on yritetty jopa arvioida hintaa:
  - http://www.infoq.com/news/2012/02/tech-debt-361
- Toisaalta 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
  - http://martinfowler.com/bliki/DesignStaminaHypothesis.html
- Vastaavasti joskus voi ”lyhytaikaisen” teknisen velan ottaminen olla järkevää
  - Esim. voidaan saada tuote nopeammin markkinoille tekemällä tietoisesti huonoa designia joka korjataan myöhemmin
  - http://blogs.construx.com/blogs/stevemcc/archive/2007/11/01/technical-de

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 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://wiki.java.net/bin/view/People/SmellsToRefactorings
  - 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 parametre list
  - Uncommunicative name
  - Comments (hetkinen, 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

Miten refaktorointi kannattaa tehdä

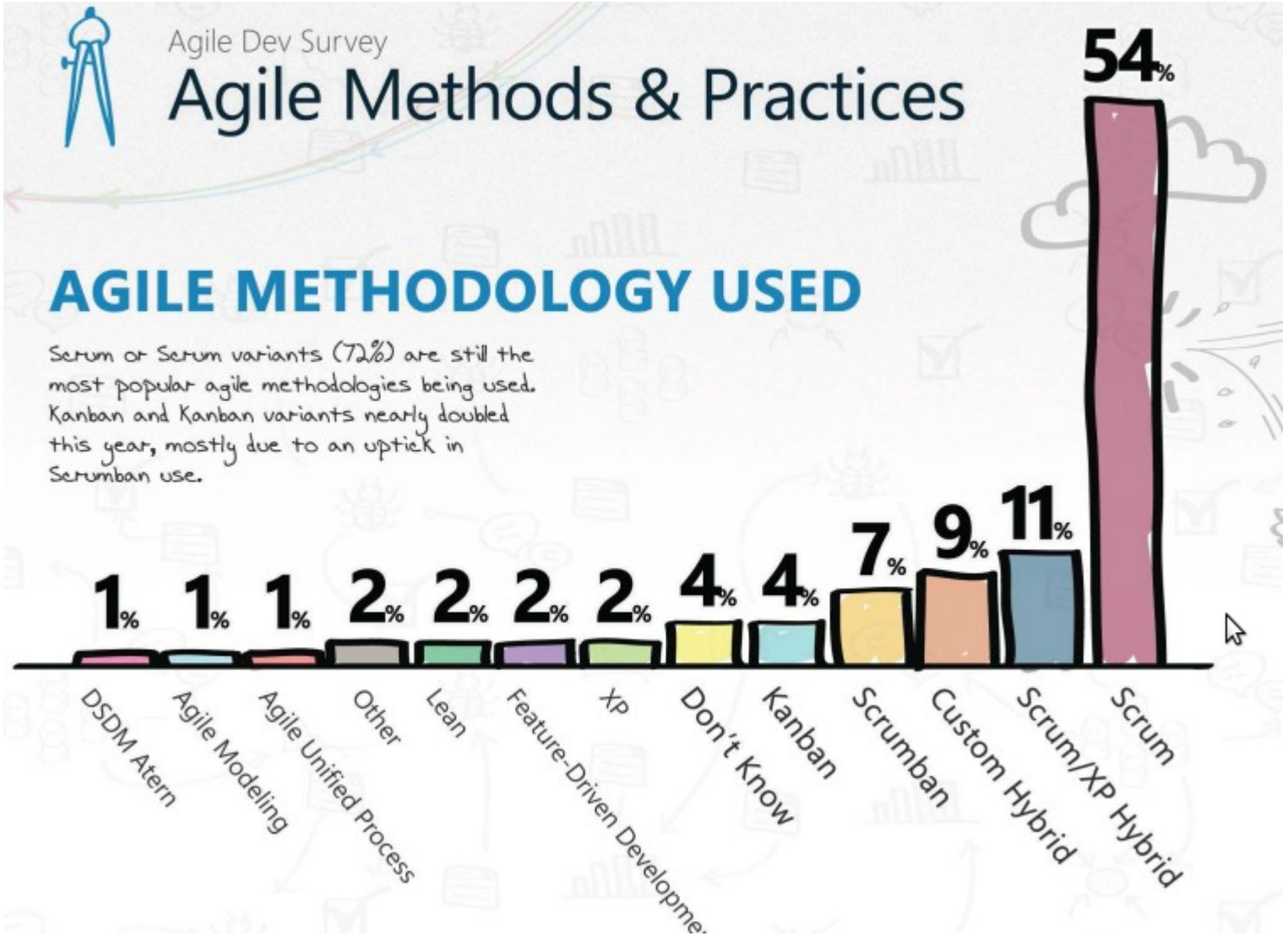
- Refaktoroinnin melkein ehdoton edellytys on kattavien yksikkötestien olemassaolo
  - Refaktoroinninhan on tarkoitus ainoastaan parantaa luokan tai komponentin sisäistä rakennetta, ulospäin näkyvän toiminnallisuuden pitäisi pysyä muuttumattomana
- Kannattaa ehdottomasti edetä pienin askelin
  - Yksi 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

Käytetäänkö ketteriä menetelmiä
ja toimivatko ne?

Miten laajalti Agilea käytetään

- 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-n
- **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 on Suomessa suosittua:
  - The results of the survey reveal that a majority of respondents' **organizational units are using agile and/or lean methods (58%)**
  - Markkula ym.: Survey on Agile and Lean usage in Finnish software industry, ESEM 2012 (ks. ACM digital library)
  - http://esem.cs.lth.se/industry_public/Rodriguezetal_ESEM2012_IndustryTrack_

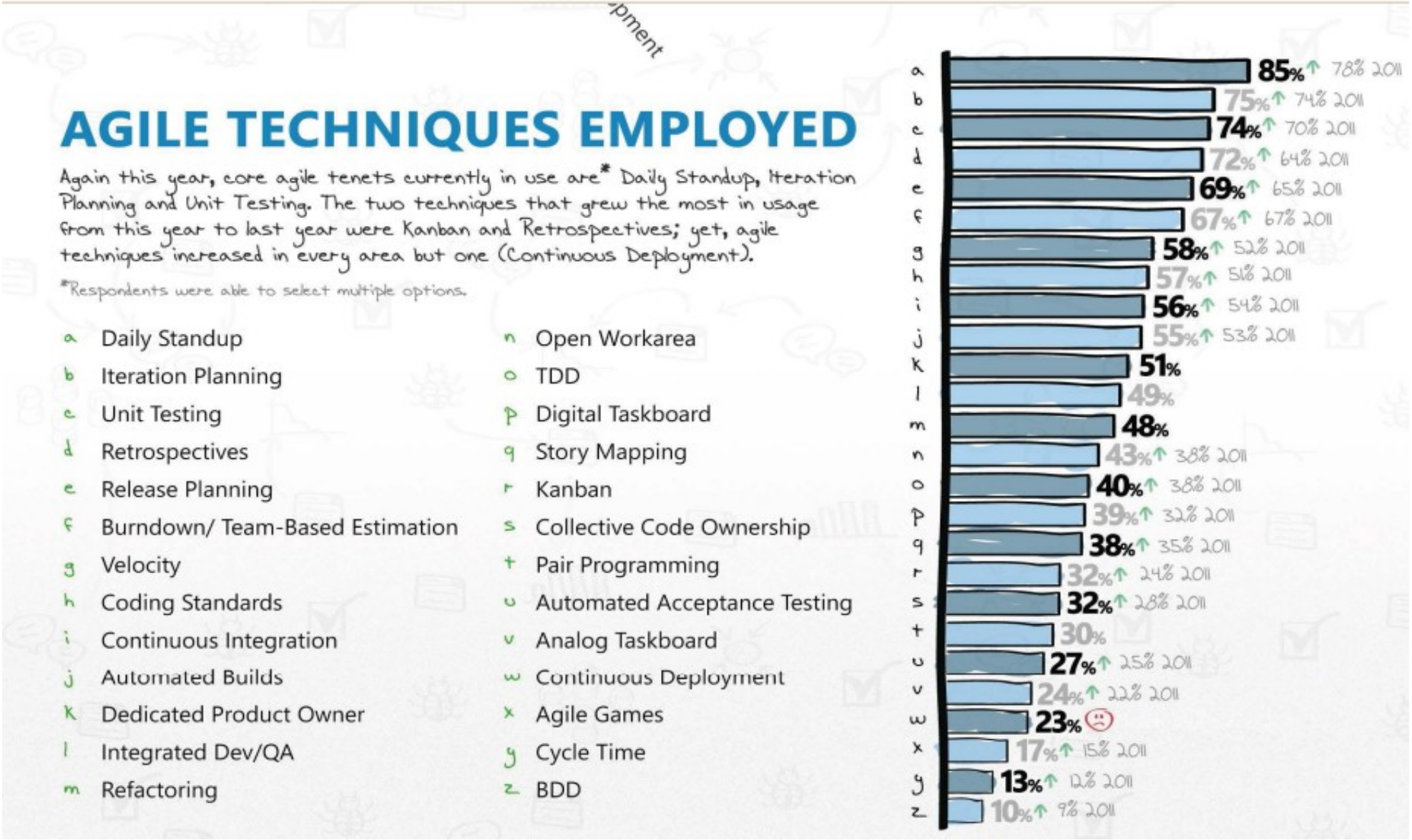
Mitä ketteriä menetelmiä käytetään?



- VersionOnen ”internetin virallisesta” vuosiraportista
  - http://www.versionone.com/pdf/7th-Annual-State-of-Agile-Development-S

Ketterät käytänteet

- VersionOne:



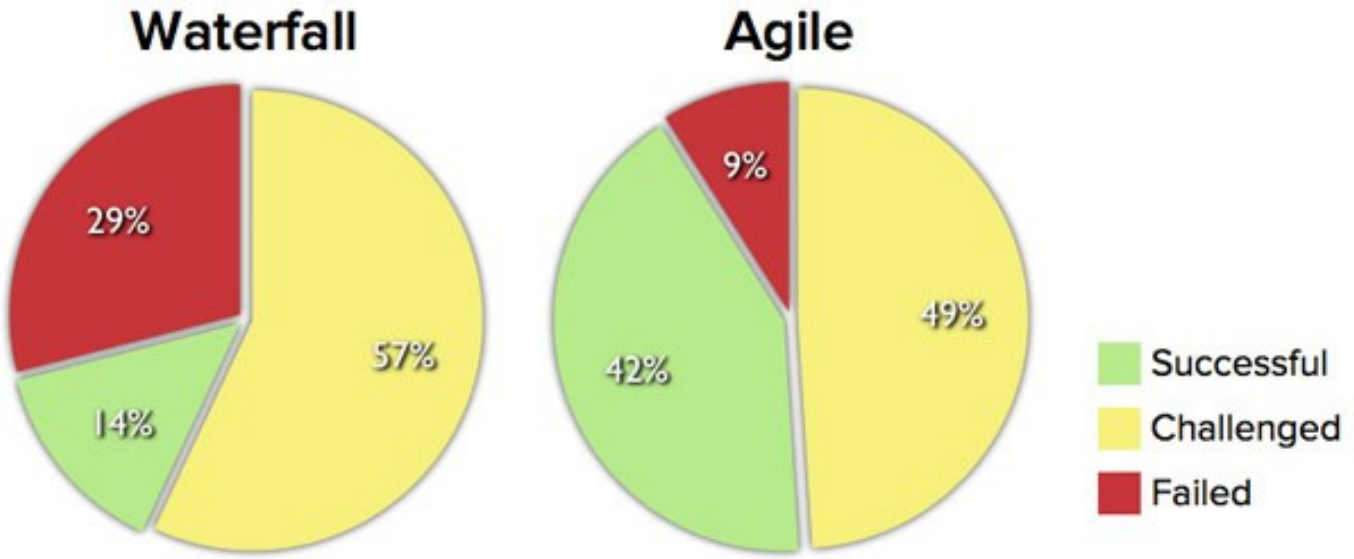
- Suomen tilanne: http://esem.cs.lth.se/industry_public/Rodriguezetal_ESEM2012_IndustryTrack_1_0.pd

Ketterät käytänteet Suomesta tehdyssä 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



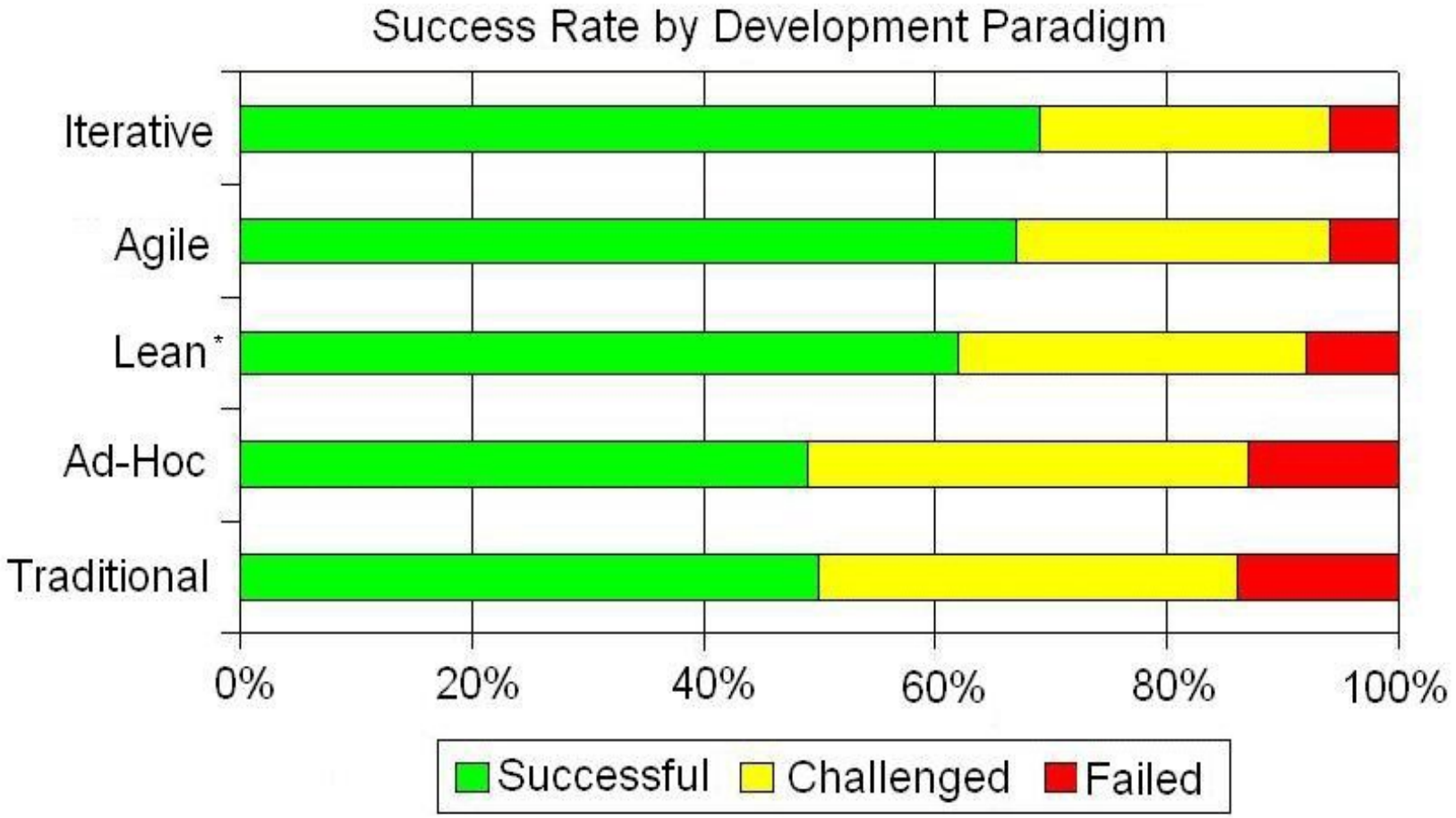
Source: The CHAOS Manifesto, The Standish Group, 2012.

- 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



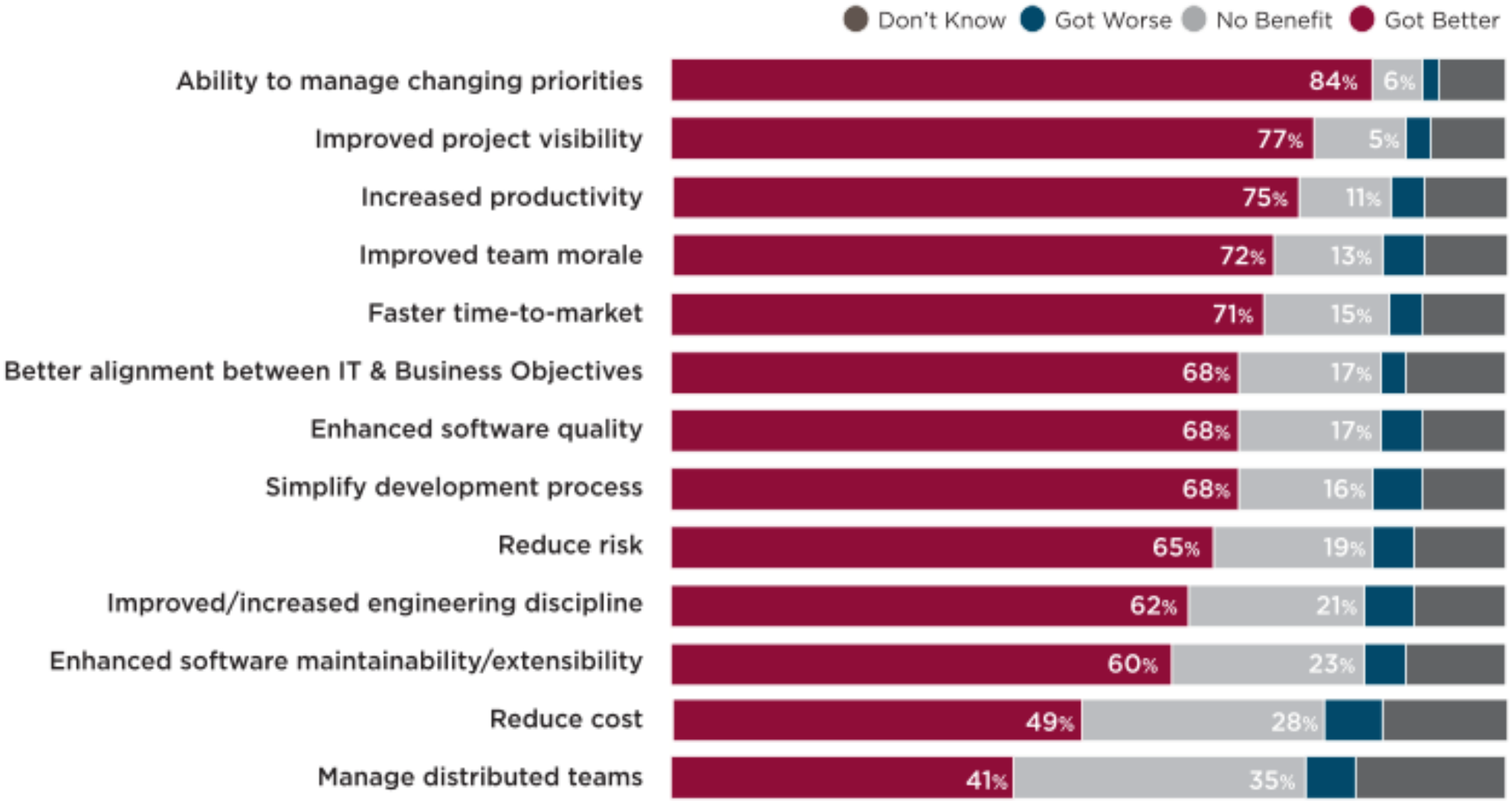
2011 is the first year where we asked about Lean.
We only had 40 respondents for this paradigm.

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 eriteltynä

- VersionOne 2011
  - http://www.versionone.com/pdf/2011_State_of_Agile_Development_Su



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

Evidenssiä on, mutta...

- Oikeastaan kaikki edelliset olivat kyselytutkimuksia
  - käsitteitä ei ole kunnolla määritelty (esim. mitä ketteryydellä tai projektin onnistumisella tarkoitetaan)
  - Kyselyyn osallistuneet eivät välttämättä edusta tasaisesti koko populaatiota
  - Kaikkien kyselyjen tekijät eivät puolueettomia menetelmien suhteen (esim. Ambler ja VersionOne)
- Eli tutkimusten validitetti on kyseenalainen
- Toisaalta kukaan ei ole edes yrittänyt 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 ohjelmistotiimillä ei välttämättä yleisty mihinkään muihin olosuhteisiin
- Olemassa olevan evidenssin nojalla kuitenkin näyttää siltä, että ongelmistaan huolimatta ketterät menetelmät ovat ainakin joissakin tapauksissa järkevä tapa ohjelmistokehitykseen

Koe

Koe

- Tiistaina 21.10 klo 09:00 – 11:30 salissa A111
  - Tarkista kurssikoelistauksesta ja kurssisivulta paria päivää ennen tenttiä
- 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
- Maven
- Jenkins
- JUnit
- Mockito
- EasyB
- Selenium
- Ebean
- Java 8

Mitä kokeessa **ei** tarvitse osata

- Git
- Maven
- Jenkins
- JUnit
- Mockito
- EasyB
- Selenium
- Ebean
- Java 8

Reading list – eli lue nämä

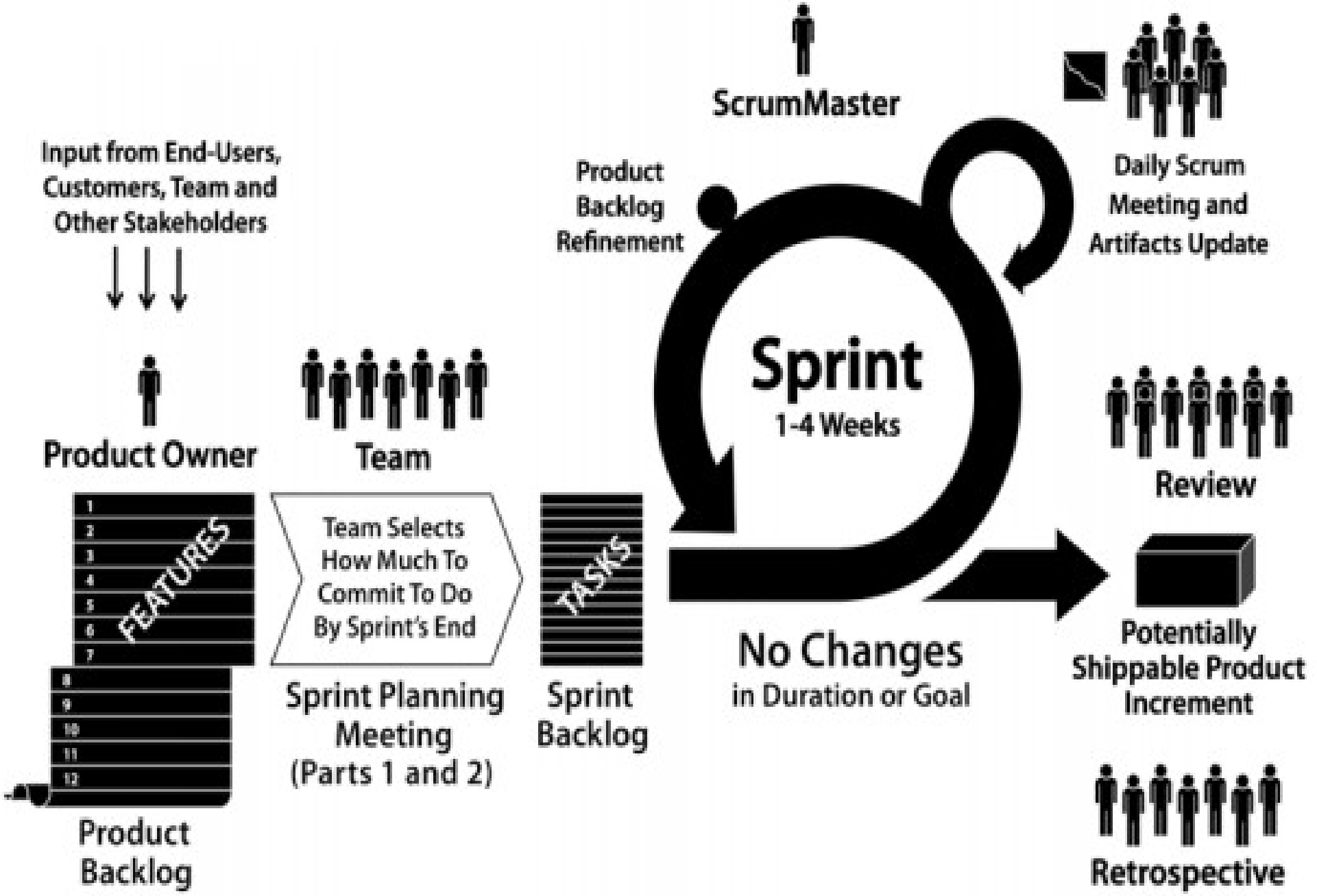
- Luentomonisteet, luentoihin 8 ja 9 liittyvät koodiesimerkit ja laskarit (paitsi edellisellä sivulla mainittujen osalta)
- http://martinfowler.com/articles/newMethodology.html
- http://www.scrum.org/Portals/0/Documents/Scrum%20Guides/Scrum_Guid
- http://www.infoq.com/minibooks/scrum-xp-from-the-trenches
  - Sivut 1-86
- http://martinfowler.com/articles/continuousIntegration.html
- http://martinfowler.com/articles/designDead.html
- 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”
  - Katselmointi ja tarkastukset
  - Vaatimusten validointi (ketterä vs. trad)
  - Koodin katselmointi
- Verifiointi ”are we building the product right”
  - Vastaako järjestelmä vaatimusmäärittelyä
- Verifiointi 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
  - ”integraatiohelvetti” → Daily build / smoke test → jatkuva integraatio → continuous delivery
  - Workflow jatkuvassa integraatiossa
- Seuraava käsiteltiin oikeastaan vasta luennolla 7
- 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 laatuattribuutit
  - 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 ja 10 alku

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