Ohjelmistotuotanto

Luento10

25.4.2012

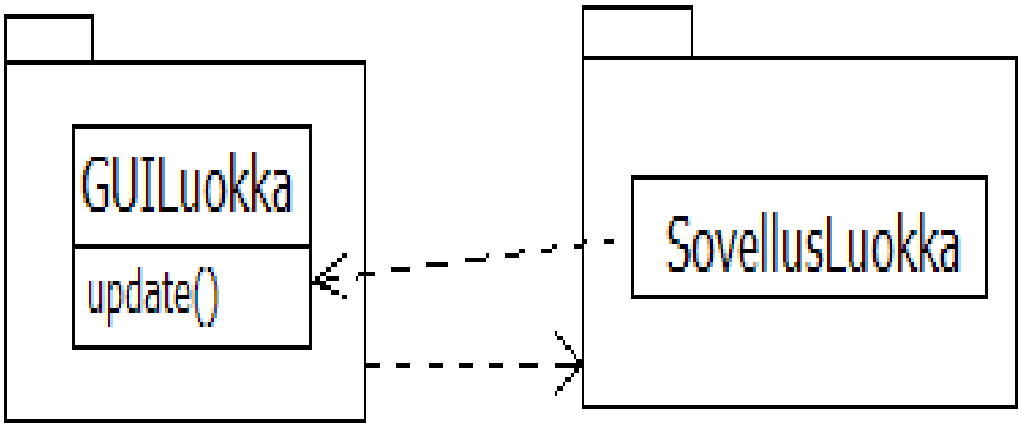
Oliosuunnittelu jatkuu

Model View Controller el MVC -malli

- MVC-mallilla tarkoitetaan periaatetta jonka avulla **malli** (model) eli liiketoimintalogiikan sisältävät olion (esim. domain-oliot) eristetään käyttöliittymän **näytöt** (view) generoivasta koodista
  - Kumpula Biershopissa on oikeastaan sovellettu WebMVC:tä, eli MVC:n www-sovelluksiin sopivaa varianttia
- Ideana on laittaa näytön/näytöt generoivan koodin ja sovelluslogiikasta huolehtivien olioiden väliin **kontrolleri** (controller)
- Kontrolleri huolehtii esim. nappien klikkaamisen tai web-sovelluksissa osoitteisiin navigoinnin tai lomakkeiden vastaanottamiseen edellyttävän toiminnallisuuden suorittamisesta kutsumalla sopivia modelin olioita
- Näytöt generoivat käyttäjälle näytettävän käyttöliittymän käyttäen joko suoraan malleissa olevaa dataa tai saamalla datan kontrollerin välityksellä (kuten WebMVC:ssä tapahtuu)
  - ks. https://wiki.helsinki.fi/display/ohtu2012/luento9 MVC
- Model ei tunne kontrollereja eikä näyttöjä ja samaan modelissa olevaan dataan voikin olla useita näyttöjä

Riippuvuuksien eliminointi

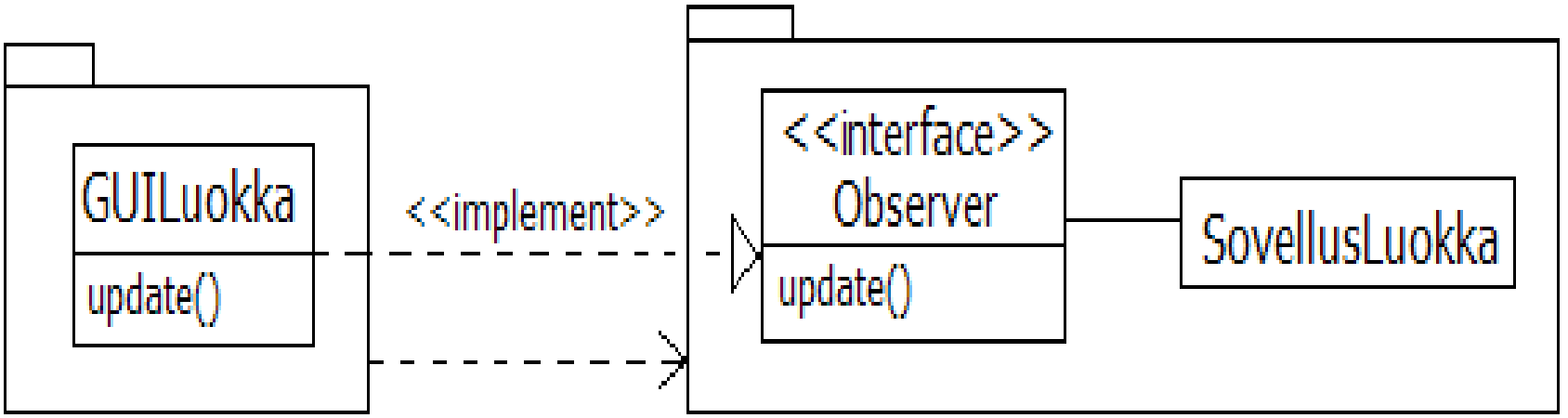
- Kerrosarkkitehtuurissa ja MVC-mallin mukaisissa sovelluksissa törmätään usein tilanteeseen, jossa sovelluslogiikan on kerrottava käyttöliittymälle jonkin sovellusolion tilan muutoksesta, jotta käyttöliittymä näyttäisi koko ajan ajantasaista tietoa
- Tästä muodostuu ikävä riippuvuus sovelluslogiikasta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



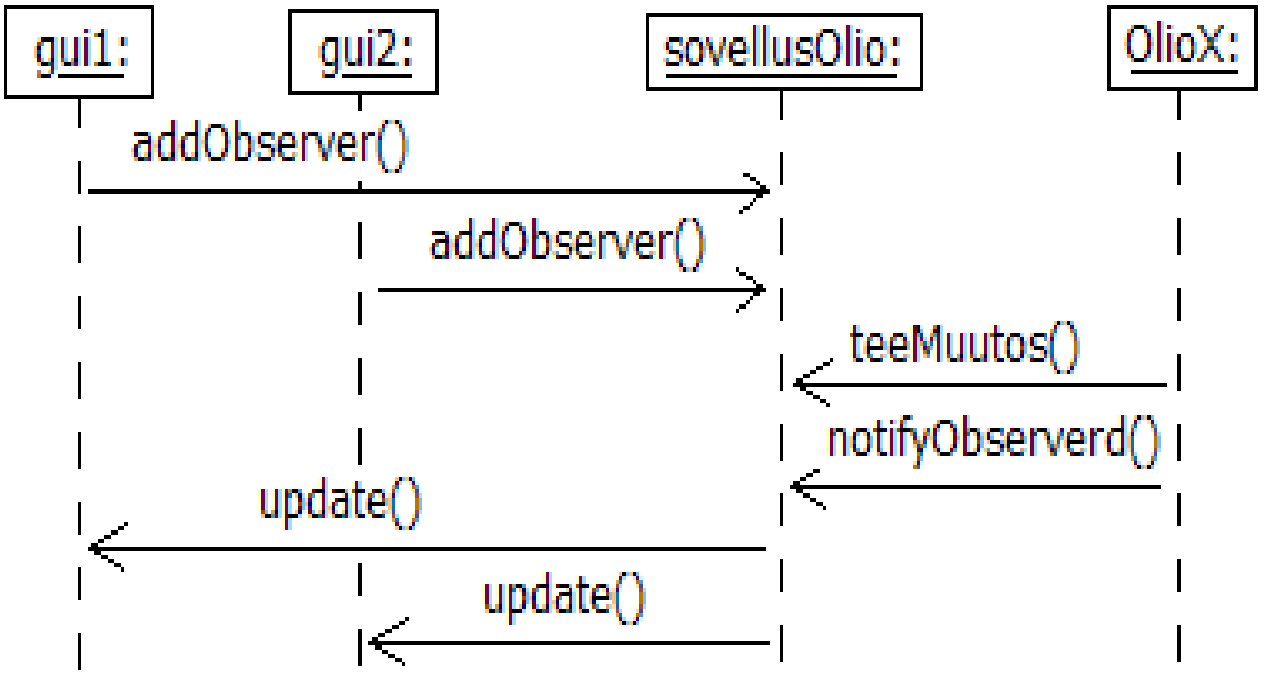
- Riippuvuus saadaan eliminoitua **observer**-suunnittelumallilla
  - Ks https://wiki.helsinki.fi/display/ohtu2012/luento9 Observer

Riippuvuuksien eliminointi observer-suunnittelumallilla

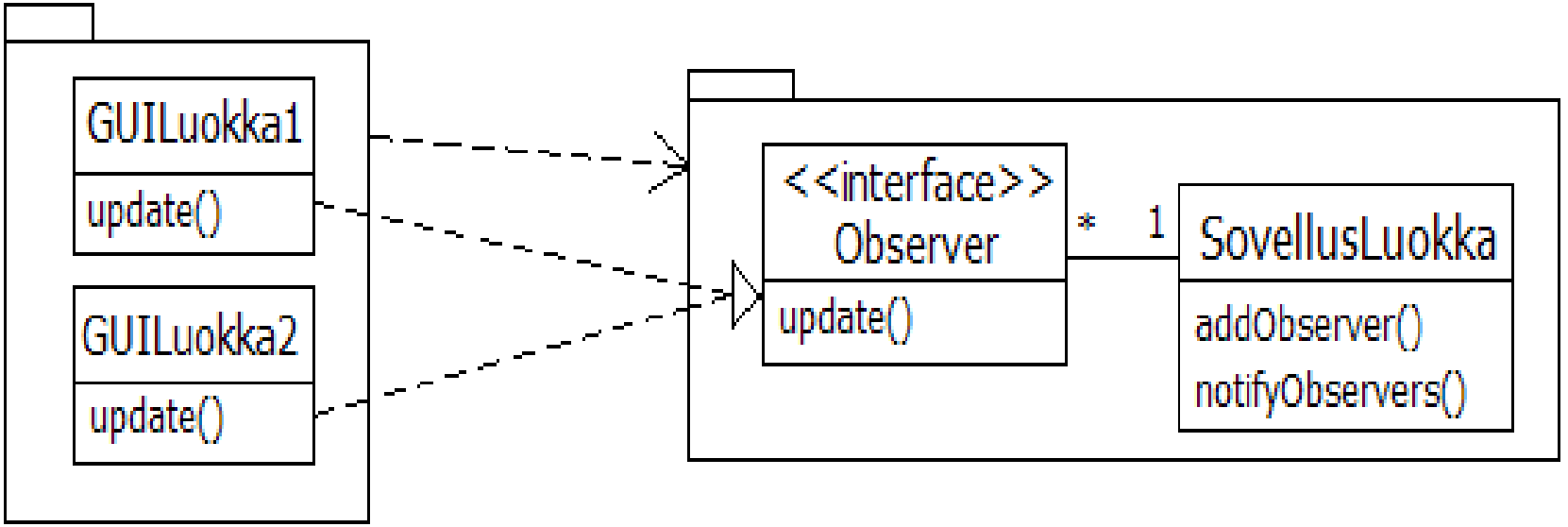
- Määritellään rajapinta, joka sisältää käyttöliittymäluokan päivitysmetodin update(), jota sovellusluokka kutsuu
  - Alla rajapinnalle on annettu nimeksi *Observer*
- Käyttöliittymäluokka toteuttaa rajapinnan, eli käytännössä toteuttaa update()-metodin haluamallaan tavalla
- Sovellusluokalle riittää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



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



Observer-suunnittelumalli




```
class Sovellusluokka{
    ArrayList<Observer> tarkkailijat;
    void addObserver(Observer o){
        tarkkailijat.add(o);
    }
    void notifyObservers(){
        for ( Observer o : tarkkailijat) o.update();
    }
    /* muu koodi */
}
```

```
Interface Observer{
    void update();
}
```

```
GUILuokka implements Observe {
    void update(){
            /* päivitetään näyttöä */
    }
    /* muu koodi*/
}
```

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 kun 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 desigina 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

Koe

Koe

- Torstaina 3.5 klo 16:00 – 19:00 B123
- 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

Reading list – eli lue nämä

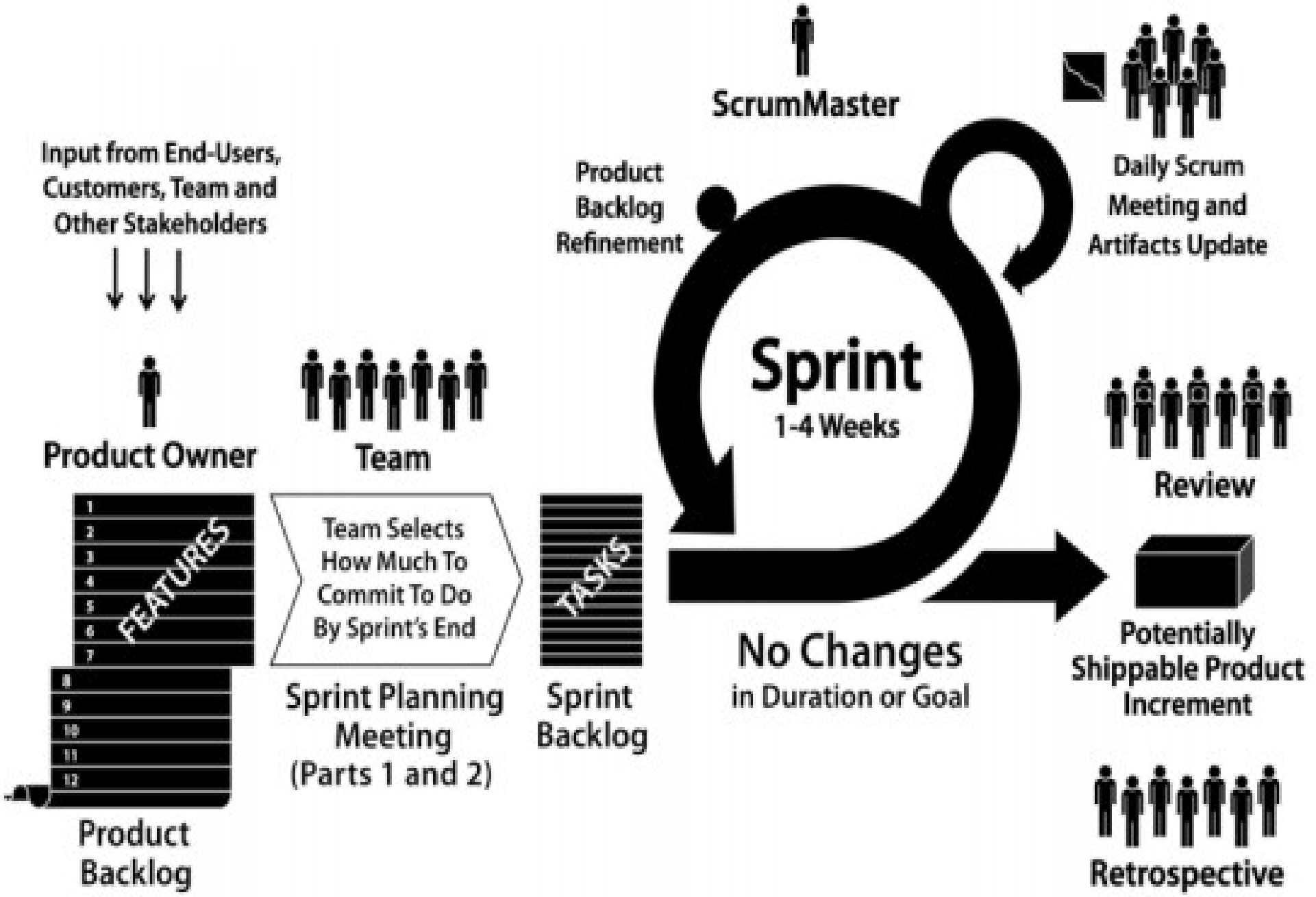
- Luentomonisteet, luentoihin 8 ja 9 liittyvät koodiesimerkit ja laskarit (paitsi edellisellä sivulla mainittujen osalta)
- http://martinfowler.com/articles/newMethodology.html
- http://assets.scrumfoundation.com/downloads/1/scrumprimer121.pdf
- http://www.infoq.com/minibooks/scrum-xp-from-the-trenches
  - 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
- Seuraavat kaksi käsiteltiin oikeastaan vasta luennolla 7
- Jatkuva integraatio
  - ”integraatiohelvetti” → Daily build / smoke test → jatkuva integraatio → continuous delivery
  - Workflow jatkuvassa integraatiossa
- 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
- Seuraava käsiteltiin vasta luennolla 8
- 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

laskaristatistiikkaa

Keskimääräinen ajankäyttö ja tehtyjen tehtävien määrä

- Lh1 – 9 tehtävää
  - 5.8 tuntia 7.8 tehtävää
- Lh2 – 7 tehtävää
  - 5.2 tuntia 5.8 tehtävää
- Lh3 – 10 tehtävää
  - 5.8 tuntia 7.6 tehtävää
- Lh4 – 11 tehtävää
  - 6.1 tuntia 7.9 tehtävää
- Lh5 – 10 tehtävää
  - 4.5 tuntia 6.8 tehtävää

- **Yhteensä – 47 tehtävää**
  - **27.4 tuntia 35.9 tehtävää**

- Lh1
  - 5.8 h
- Lh2
- Lh3
- Lh4
- Lh5
- Lh6
-