Testaus Korppi-kehityksessä

Panu Suominen
THK / JYU

Sisältö

- Käydään lävitse Korpin testaukseen ja laatuun vaikuttavia tekijöitä.

Mikä ihmeen Korppi?

- Jyväskylän yliopiston opintotietojärjestelmä
  - Käytännössä kehittynyt opiskeluun liittyvien asioiden portaaliksi.
- Java Servlet -tekniikkaa käyttävä isohko www-sovellus.
  - 3 133 luokkaa, 265 056 riviä koodia.
  - Noin 800 jsp-sivua, 140 000 riviä tekstiä.
- Kehittäminen aloitettu vuonna 2000.
- Kehittäminen on ollut pitkälti opiskelijaprojektien vastuulla.

Organisaation kehittyminen

1. Opiskelijaprojektit jatkoivat siitä, mihin edellinen jäi.
    - Tietotaidon nollaantuminen puolivuosittain.
2. Eri rahoista palkatut tekijät toteuttivat aikalailla itsenäisesti palkanmaksajiensa toiveita.
    - Yhteistyö hankalaa.
3. Kehittäminen siirtyi THK:lle.
    - Epäselvää kuka päättää, mitä kehitetään.
4. Tilaaja-tuottaja -mallia otetaan THK:ssa käyttöön.

Ketkä tekevät?

- Kehitys tapahtuuu tietohallintokeskuksen kehittämispalveluiden alaisuudessa.
- Projektipäällikko Petri Heinonen
- 7 kehittäjää / 2 asiakaspalvelussa:
  - Teija Alasalmi
  - Antti Hedlund
  - Kirsi Koponen
  - Pauli Kujala
  - Ilari Liukko
  - Kari Patana
  - Harri Pitkänen
  - Panu Suominen

Miten tehdään?

- Ketterää kehitystä
  - Scrum on pyrkimyksenä
    - Käytössä 1 viikon iteraatot.
  - Asiakaspalveluvuorojen vaihtaminen viikoittain
    - Kehittämisvuorossa oleville työrauha.
    - Kosketus käyttäjiin säilyy kaikilla.
    - Myös kehittäjät joutuvat käyttämään järjestelmää.

Entäs se testaus?

- Automaattiset yksikkötestit
  - TDD
  - n. 3 300
  - kattavat noin 10% koodista
- Testataan käsin
  - Päivityksen yhteydessä (2 kk välein)
  - Vie vajaan viikon koko ryhmältä

Miten tähän on tultu?

- Ohjelmiston kehityksessä on tehty iso liuta vääriä valintoja.
- Organisaatiolla on iso merkitys siihen kuinka testaus onnistuu.
- Korpin tapauksessa testausta on alettu huomioimaan vasta viimeaikoina.

Organisatio luo puitteet testaukselle

Tilaaja-tuottaja -malli

- Tilaaja tulee rahojen ja vaatimusten kanssa.
- Tuottaja vie rahat ja tuottaa asiakkaan tilaaman järjestelmän.
- Koska tilaaja on tarkka rahoistaan tuotoksesta ollaan yleensä hyvin kiinnostuneita.

Yliopisto ohjelmiston tilaajana

- Rahat ovat yhtäaikaa ei kenenkään ja yhteisiä.
  - Raha ei tule suoraan liiketoiminnasta, joten ostoksia ei ohjaa liiketoiminallinen hyöty.
  - Yleensä kunnolla tehty ominaisuus olisi yliopiston kannalta halvempi vaihtoehto, mutta yksittäisen tilaajan kannalta kalliimpi.
- Rahat tulevat lopulta vain yhdeltä asiakaalta: yliopistolta.
- Kuka edustaa opiskelijoiden tarpeita?
  - Tilaaja ja loppukäyttäjä voivat olla hyvin erillään toisistaan.

Yliopisto ohjelmiston tuottajana

- Ohjelmistokehitys ei ole ydinliiketoimintaa
  - Ulkoistamisen tai lopettamisen vaara
  - Lyhyet työsuhteet
    - Tiedon katoaminen työntekijöiden vaihtuessa.
    - Mielenkiinnon puute lopputulosta kohtaan.
- Johtamiskulttuurin puuteet
  - Kuka vastaa mistäkin?
  - Kuka hyväksyy lopputuloksen?
- Projektoinnin ongelmat
  - Työntekijöillä monta rautaa tulessa.
  - Oikea käsi ei tiedä, mitä vasen tekee.
  - Resurssien suuntaaminen oikein.

Seurauksia

- Sekava projektimalli tai sen puuttuminen kokonaan
  - Kuka määrää, mitä tehdään?
  - Ketkä ovat tekemässä samaa asiaa?
  - Milloin pitää olla valmista?
- Epäselvät vaatimukset
  - Mistä tiedän, kuinka testaan ominaisuuden X?
- Huonot käytänteet
  - Miten ehdin korjata aikaisemmat töppäykset?
- Vähäinen mielenkiinto lopputulosta kohtaan
  - Miksi miettisin testejä, jos vähempikin riittää?

Mitä tästä opitaan?

- Organisaation pitää tukea (vaatia) testausta tai se jää tekemättä.
- Suunnitteleminen pitää aloittaa testaamisesta.
- Testauksen vaatiminen on osa kehittäjän ammattitaitoa.

Testejä on monenlaisia

Jos koodi on suunitelma*

	Perinteinen teollisuus	Ohjelmistoteollisuus
Suunnittelu	Halpaa, yleensä ennustettavaa	Kallista, hidasta ja epävarmaa
Rakentaminen	Kallista, yleensä hidasta. Tehdään kerran.	Käytännössä ilmaista, erittäin nopeaa. Tehdään jatkuvasti.
Testaaminen	Suunnitelmien testaaminen (simulointi, prototyypit) tärkeää. Varmistusvirheiden eliminointi.	Testaaminen keskittyy rakennettuihin osiin. Koodia itsessään harvemmin testataan. Rakennusvaiheen valmistusvirheet erittäin harvinaisia.

Miksi kannattaa testata?

- Auton rakentamisen jälkeen tiedetään:
  - Auto näyttää siltä, miltä suunniteltiin (tai ei näytä)
- Ohjelman kääntämisen jälkeen tiedetään:
  - Suunnitelmassa ei ollut syntaksivirheitä.
  - ?
- Testaaminen on ainoa tapa tunnustella, että ohjelmassa on kaikki paikallaan.

Testaaminen käsin

- Mahdollisuus löytää aivan uusia virheitä.
- Ainoita tapoja testata käytettävyyttä.
- Jos olisi vaihtoehtoja, emme käyttäisi.
- Esiintyneitä ongelmia:
  - Vaikea toteuttaa kerta toisensa jälkeen samanlaisena.
  - Jos virheitä ei sada korjattua, motivaatio katoaa.
  - Vie tolkuttomasti aikaa.

Automaattiset testit

- Estää vanhojen vikojen uudelleen syntymisen.
- Mahdollistaa nopeat muutokset.
- Vapauttaa rutiinityöstä.
- Kehittäjän paras kaveri.
- Vaatii työtapojen ja työkalujen kehittämistä.

Tekniikka ja työkalut

Kääntäminen 1.

- Koodit alunperin samassa rakenteessa kuin ajettaessakin.
- Lisäksi epästandardeja käytänteitä (linkkejä).

```
javac WEB-INF/classes <- ei toimi
/usr/local/bin/korppi/fixLinks.sh
cp -r * /usr/local/tomcat/webapps/korppi
rm -Rf /usr/local/tomcat/webapps/korppi/doc
cd /usr/local/tomcat/bin
./startup.sh
```

Kääntäminen 2.

- Koodit eri rakenteeseen kuin ajettaessa
  - Työkalujen parempi tuki

```
ant package
jar -xf kotka.war /tmp/korppi
/usr/local/bin/korppi/fixLinks.sh /tmp/korppi
mv /tmp/korppi /usr/local/tomcat/webapps/korppi
cd /usr/local/tomcat/bin
./startup.sh
```

Kääntäminen 3.

- Siirryttiin standardiin tapaan

```
mvn tomcat:run
```

Työkalut aluksi

- CVS
- Java 1.4
- Tomcat 3.2.3
- JBuilder?

Työkalut nyt

- SVN
- Java 6.0
- Tomcat 6.0
- Eclipse / Netbeans
- TestNG (testaus)
- Maven (kääntäminen ja projektin hallinta)
- Selenium (selaimen kauko-ohjaus)
- Hudson (integrointipalvelin)
- Sonar (metriikoiden tarkkailu työkalu)

Automaattitestien kehittyminen

- Aluksi satunnaisia testejä joillekin paloille
  - Ei koottua tapaa ajamiseen.
  - Ajoajat hyvin vaihtelevia.
  - Myöhemmin näistä on ollut lähinnä vain haittaa.
- Nyt
  - Yksikkötestit.
  - Testit pyritään tekemään ennen varsinaista koodia.
  - Ensimmäiset automaattiset käyttöliittymätestit.
- Ongelmia
  - Tietokantaa käyttävät testit.

Havaintoja automaattitesteistä

- Testien tulisi olla helposti ajettavia ja nopeita.
- Lopputuloksen tulisi olla selkeä lopputulos.
- Testien käyttämiseen ja kirjoittamiseen on oltava käytänteet .
  - Jos muut sivuuttavat testit, niistä ei ole paljoa hyötyä.
- Testin pitäisi vastata selkeästi kysymykseen, mitä testattava asia tekee.
  - Koodin pitää olla luettavaa.
- Painostaa hyviin suunnittelumalleihin.
  - Testien vuoksi koodia on jossakin määrin uudelleen käytettävää.

Single Responsibility Principle

- Yksi olio ohjelmoinnin periaatteista Robert C. Martin mukaan.
- Luokalla vain yksi vastuu ja syy muuttua.
  - Monta vastuuta altistaa luokan muutoksille useammin.

SRP: Ei vastuita ollenkaan

```
class {
    private int x;
    public int getX(){ return x; }
    public void setX(int newX){ this.x=newX; }
}
```

- Voidaan käytetään tiedonsiirtoon ohjelman eri osien välillä.
- Mitä etuja?
  - Ei tarvitse testata?
  - Vähentää metodien parametrien määrää.
    - Selkeyttää koodia
  - ...
- Mitä ongelmia?

SRP: Ei vastuita ollenkaan

- Luokan sisältämän datan eheyden tarkistaminen siirtyy helposti muiden vastuulle.
  - Copy-paste -koodi
    - Testeissä samat tarkistukset eri luokille.
    - Ajettavissa luokissa samaa koodia.
- Luokka vuotaa todennäköisesti ulospäin toteutuksen yksityiskohtia.
- Voidaan yleensä välttää rajapintojen käytöllä.

SRP: Monta vastuuta

```
/* Represents addition of multiple integers. */
class IntegerAddition {
    Collection<Integer> terms = ..;
    public void addTerm(int number){..}
    public void int doTheMath(){..}

    /* Save to the database */
    public void save(){..}

    /* Load from the database */
    public static IntegerAddition load(int id){..}
}
```

- Etuja?
- Haittoja?

SRP: Monta vastuuta

- Lopputuloksena on monimutkainen luokka
  - Useat vastuut lisäävät koodirivien määrää.
  - Samaan asiaan liittyvät osat hajaantuvat helposti ympäriinsä (esim. tietokannan käsittely yhden paikan sijasta monessa)
- Testaaminen hankalaa.
  - Luokka on riippuvainen tietokanan käsittelyyn liittyvästä koodista.

Muita testauspainajaisia

- Riippuvuudet luokkien toteutuksista rajapintojen sijaan.
  - Ja kaikki, mikä rikkoo SOLID-sääntöjä.
- Suorat ulkoisten resurssien käyttämiset, staattiset luokat, muuttujat yms.
- Moniajoon liittyvät ongelmat.

Lopuksi

- Testauksen kehittäminen on organisaation, työtapojen, työkalujen sekä koodin kehittämistä. Testaus ei elä yksinään.
- Testien kirjoittaminen ei käy luonnostaan.
- Ohjelma ei ole valmis ilman kunnollisia testejä.