Ohjelmistojen testaus

Juha Taina

1. Johdanto (P&Y:1-4)

- Kurinalainen insinöörityö sisältää suunnittelun ja rakentamisen lisäksi välttämättä tehtäviä, joiden tarkoitus on tunnistaa ja poistaa keskeneräisestä tai valmiista tuotteesta vikoja (defects).
- Ohjelmistotuotanto on kurinalaista insinöörityötä, ja samoin kuin kovissa insinööritieteissä, myös ohjelmistotuotannossa tuotteen vikoja etsitään ja korjataan. Tehtävää ohjelmistoa *verifioidaan* ja *validoidaan.*

Verifiointi ja validointi

- *Verifiointi* (verification) tarkoittaa työtä, jolla varmistetaan, että tehtävä ohjelmisto vastaa sille tehtyä määrittelyä (specification).
  - Verifiointi tiivistetään usein lauseeksi ”Rakennammeko tuotetta oikein?” (Are we building the product right?)
- *Validointi* (validation) tarkoittaa työtä, jolla varmistetaan, että tehtävä ohjelmisto täyttää asiakkaan sille asettamat todelliset tarpeet.
  - Validointi tiivistetään usein lauseeksi ”Rakennammeko oikeaa tuotetta?” (Are we building the right product?)

Ohjelmistojen testaus / Taina 3

Verifiointi ja validointi 2

- Verifiointi ja validointi (V&V) ovat mukana ohjelmistoprosessin alusta alkaen. Jo kelpoisuusselvityksessä (feasibility study) tehtävää tuotetta täytyy arvoida ja analysoida. Arviot ja analyysit pitää varmentaa – siis verifioida.
- Verifiointiin ja validointiin on lukuisia tekniikoita, joista pääosa voidaan luokitella kahteen ryhmään: *staattinen analyysi* ja *(dynaaminen) testaus*.

Ohjelmistojen testaus / Taina 4

Staattinen analyysi ja (dynaaminen) testaus

- *Staattinen analyysi* (static analysis) on yleisnimi verifiointi- ja validointitekniikoille, jotka eivät vaadi ohjelmakoodin suoritusta.
  - Tekniikoita kutsutaan kirjallisuudessa myös staattisen testauksen tekniikoiksi, mutta kurssilla emme kuormita suotta testaus-termiä.
- *(Dynaaminen) Testaus* ([dynamic] testing) tarkoittaa verifiointi- ja validointitekniikoita, jossa ohjelmaa tai sen osaa suoritetaan tietyllä syötteellä ja saadut tulokset analysoidaan.

Ohjelmistojen testaus / Taina 5

Staattisen analyysin ja (dynaamisen) testauksen suhde

- Sekä staattista analyysia että testausta tarvitaan verifioinnissa ja validoinnissa:
  - Tekniikoilla löydetään eri tyyppisiä virheitä
    - Staattisella analyysilla löydetään isoja kokonaisuuksia, testauksella pieniä yksityiskohtia.
  - Tekniikoita käytetään eri vaiheissa projektia
    - Testaus vaatii ohjelmakoodin, jota ei ole projektin alussa saatavilla.
  - Tekniikoiden tavoitteet eroavat toisistaan
    - Testaus sopii virheiden etsintään, staattinen analyysi ominaisuuksien varmentamiseen.
  - Tekniikoilla voidaan tehdä kompromisseja kustannusten ja laadun välillä.
    - Staattinen analyysitekniikka *tarkastukset* (inspections) on erinomainen mutta kallis tekniikka. Kustannuksia voidaan säästää käyttämällä tarkastuksia kriittisissä työvaiheissa.
- Oppikirja käsittelee staattista analyysia ja testausta yhdessä. Siksi testauksen lisäksi kirjassa puhutaan *analyysista ja testauksesta* (analysis and testing, A&T).

Ohjelmistojen testaus / Taina 6

Analyysin ja testauksen tehtävät

- A&T:lla on kaksi tehtävää:
  - varmentaa, että ohjelmisto on riittävän laadukas ja
  - parantaa ohjelmiston laatua löytämällä vikoja.
- Laadun varmennus tarkoittaa, että A&T:n tekniikoilla tarkastetaan ohjelmiston täyttävän ennalta määritellyt laatuvaatimukset.
  - Yleensä laatuvaatimukset määritellään laatuattribuutteina, joille on määrätty vaaditut arvot.
- Laadun parannus tarkoittaa, että vikoja löytämällä tuotteen *luotettavuutta* (dependability) parannetaan.

Ohjelmistojen testaus / Taina 7

Lisää validoinnista ja verifioinnista

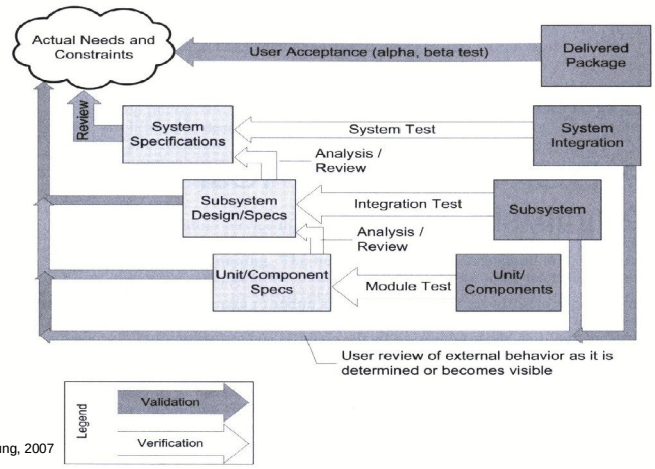
- Validoinnin avulla siis varmennetaan, että ohjelmisto on sitä, mitä haluttiin. Tämä ei ole sama asia kuin verifiointi: kirjattujen vaatimusten toteutumisen tarkistus.
  - Vaatimusmäärittelydokumentti (Software requirements specification, SRS) on kuvaus ehdotetusta ratkaisusta annettuun ongelmaan.
  - Ratkaisu voi kuitenkin olla vajaa tai virheellinen, joten ohjelmiston varmentaminen sen mukaiseksi ei vielä takaa laadukasta lopputulosta.
- Ohjelmisto, joka täyttää sille asetetut tarpeet, on *hyödyllinen* (useful).
- Ohjelmisto, joka on yhdenmukainen määrittelynsä kanssa, on *luotettava* (dependable).

Ohjelmistojen testaus / Taina 8

Kehitettävän tuotteen verifiointi ja validointi

- Seuraavalla kalvolla on kuva verifioinnin ja validoinnin suhteesta kehitettävään tuotteeseen.
- Kaavio ei tarkoita lineaarista prosessia, vaikka sen pohjana on kuuluisa *V-malli* (V-model).
  - V-malli lienee peräisin Myersiltä vuodelta 1979. Se on malli, joka kertoo testauksen ja vesiputousmallin välisen yhteyden.

Verifioinnin ja validoinnin suhde kehitettävään ohjelmistoon



Kuvalla (c) Pezzè & Young, 2007

Subjektiivinen validointi vs. objektiivinen verifiointi

- Validointi on subjektiivista, koska validoitava termi ”hyödyllinen” on moniselitteinen. Validoinnissa on iso väärinkäsitysten riski.
- Vastaavasti verifiointi on lähes objektiivista ja yksiselitteistä. Ohjelmistolla on joukko kirjattuja ominaisuuksia, joita varmennetaan verifioinnilla.
  - Sana ”lähes” tulee ns. implisiittisistä vaatimuksista, joita ei kirjata ylös, mutta joiden oletetaan olevan tuotteessa. Myös nämä pitää verifioida.

A&T:n rajat

- On luonnollista olettaa, että täsmällisesti määritelty ohjelmisto on mahdollista verifioida täydellisesti. Valitettavasti näin on vain harvoin.
  - Muutamissa erikoistapauksissa on mahdollista käyttää staattisen analyysin tekniikkaa *ohjelmiston todistamista oikeaksi*, jonka avulla ohjelmisto voidaan verifioida täydellisesti.
- Dijkstra esitti tämän jo 1972 lauseessaan: ”Testauksella voidaan löytää virheitä, mutta ei voida osoittaa ohjelmistoa virheettömäksi.”
- Dijkstran lause pätee pääosin yhtä hyvin staattiseen analyysiin.

A&T:n rajat 2

- A&T:ssa kombinatoriikka asettaa nopeasti rajat täydelliselle verifioinnille.
  - Esimerkiksi testauksessa käytännössä yhtään ohjelmaa ei voida testata kaikilla syötteillä.
    - Ohjelma, joka laskee yhteen kaksi 32-bittistä kokonaislukua, sisältää noin 10^21 eri syötettä. Testaamalla miljardi testiä sekunnissa syötteiden testaukseen menee 30.000 vuotta.
- Ongelmaa kierretään määrittelemällä täydellistä verifiointia heikompi analysoitavissa oleva ehto.
  - Esimerkiksi edellä voidaan kaikkien kombinaatioiden sijaan valita sopiva osajoukko kombinaatioita. Hyvin valitulla osajoukolla päästään melko lähelle laajan testijoukon hyötyä.

A&T:n periaatteet

- Oppikirja listaa kuusi analyysin ja testauksen *perusperiaatetta* (basic principles). Periaatteet ovat ominaisuuksia, jotka A&T:lla on riippumatta sovelluksesta ja prosessista.
- Periaatteet ovat:
  - Herkkyys: kaadutaan mieluummin aina kuin toisinaan
  - Toisteisuus: tehdään aikeet selkeiksi.
  - Rajoittaminen: yksinkertaistetaan ongelmaa.
  - Ositus: käsitellään pienempiä osaongelmia.
  - Näkyvyys: helpotetaan tietojen saatavuutta.
  - Palaute: käytetään kokemusta hyväksi.

A&T:n periaatteet 2

- Herkkyys:
  - Ohjelmistossa olevat viat aiheuttavat virheellisen suorituksen, mutta eivät välttämättä jokaisella suorituskerralla. Puhutaan *virheherkkyydestä* (fault sensitivity).
  - 100% virheherkkä koodi on helppo testata, koska ohjelma kaatuu jokaisella suorituksella.
  - Mitä pienempi on virheherkkyys, sitä suuremmalla todennäköisyydellä virhe jää huomaamatta.
  - Mitä myöhemmin virhe huomataan, sitä kalliimmaksi sen korjaus tulee.

Ohjelmistojen testaus / Taina 15

A&T:n periaatteet 3

- Toisteisuus:
  - Kun kaksi osaa ohjelmistotuotteesta ja/tai dokumentaatiosta on toisistaan riippuvaisia, niiden välillä on toisteisuutta.
  - Kun toisteisuuden tyyppi tunnetaan, sitä voidaan testata analyysin ja testauksen menetelmin.
- Rajoittaminen:
  - Usein ohjelmiston tai dokumentaation ominaisuus voi olla niin yleinen, että sitä ei voida varmentaa helposti
  - Toisinaan tällainen yleinen ominaisuus voidaan korvata lähes yhtä ilmaisuvoimaisella ominaisuudella, joka on kuitenkin varmennettavissa helposti.
  - Stattinen analyysitekniikka *ohjelman käännös* käyttää paljon rajoittamista.

Ohjelmistojen testaus / Taina 16

A&T:n periaatteet 4

- Ositus:
  - Useimmat ongelmat ovat niin monimutkaisia, että niiden ratkaisuja ei voida verifioida kerralla.
  - Ongelma voidaan osittaa pienemmiksi osaongelmiksi, joiden ratkaisut ovat verifioitavissa.
  - Myös staattinen tekniikka *mallinnus* on osittamista. Siinä rakennetaan yksinkertaistus (malli), joka voidaan verifioida.
- Näkyvyys:
  - Mitä paremmin ohjelmiston toiminta tunnetaan, sitä helpompi se on verifioida: ohjelmiston rakenne on näkyvä.
  - Näkyvyyttä voidaan parantaa suunnitteluvaiheessa esimerkiksi käyttämällä tekstipohjaista tietojen talletusta.

A&T:n periaatteet 5

- Palaute:
  - Analyysi- ja testausammattilaiset eivät keksi joka kerta samoja ideoita uudestaan. Sen sijaan he käyttävät hyväksi aiemmin kerättyä tietoa vastaavien ohjelmistojen verifioinnista ja validoinnista.
  - Hyvä palautemekanismi on rakennettu suoraan prosessiin, jolloin projekteissa tietojen keruu tapahtuu mahdollisimman pitkälle lisäämättä työntekijöiden kuormaa.

A&T:n periaatteet 6

- A&T:n periaatteet muodostavat kehyksen, jonka varaan yritykset voivat rakentaa oman testausprosessinsa yksityiskohdat.
- Vaikka periaatteet ovat yksinkertaiset ja vaikuttavat jopa triviaaleilta, niiden kirjaaminen ylös helpottaa analyysin ja testauksen suunnittelua.
- Listatut kuusi periaatetta ovat yleisiä kaikelle A&T:lle. Jokainen A&T:n tekniikka sisältää lisäksi omia periaatteita.

A&T ohjelmistoprosessissa

- A&T ei ole prosessin viimeinen työvaihe, kuten joskus ajatellaan, vaan se on mukana prosessin alusta alkaen.
  - Analyysia voidaan tehdä minkä tahansa työvaiheen aikana tai päättyessä
  - Testausta voidaan tehdä rinnan minkä tahansa työvaiheen kanssa määrittelemällä ja suunnittelemalla tehtäviä testejä.
  - Varsinainen testitapausten suoritus on vain jäävuoren huippu. Suurin osa A&T:sta on tehty aiemmin.

A&T ja laatu

- Tuotteen laatuattribuutit ovat tuotteelle asetettuja tavoitteita. Ne ovat joko sisäisiä tai ulkoisia.
- *Ulkoiset laatuattribuutit* (external qualities) ovat ominaisuuksia, jotka näkyvät suoraan asiakkalle.
  - Ulkoisia attribuutteja ovat mm. luotettavuus (dependability), viive (latency), käytettävyys (usability) ja läpäisykyky (throughput).
- *Sisäiset laatuattribuutit* (internal qualities) vaikuttavat välillisesti tuotteen laatuun.
  - Sisäisiä attribuutteja ovat mm. ylläpidettävyys (maintainability), uudelleenkäytettävyys (reusability), testattavuus (testability) ja seurattavuus (traceability).

Luotettavuuden ominaisuudet

- Luotettavuus on A&T:n kannalta mielenkiintoisin laatuattribuutti. Luotettava ohjelmisto vastaa määrittelyjään.
- Luotettavuuden vahvin aste on *oikeellisuus* (correctness). Ohjelmisto on oikeellinen, jos ja vain jos se on määritelmänsä mukainen.
- Oikeellisuutta ei yleensä voida verifioida. Tätä lievempi muoto on *käyttövarmuus* (reliability – usein myös luotettavuus), joka määrittelee, miten todennäköisesti ohjelmisto toimii tietyllä käyttökerralla tai aikavälillä määritelmänsä mukaisesti.

Luotettavuuden ominaisuudet 2

- Oikeellisuus ja käyttövarmuus ovat voimassa, vaikka käyttöympäristössä olisi vikaa.
  - Esimerkiksi tekstinkäsittelyjärjestelmä voi olla käyttövarma, vaikka se ei voi tallentaa täydelle levylle.
- Kun ohjelmisto kaatuu poikkeustilanteeseen virheen takia tai ympäristön vaikutuksesta, sillä on väliä, miten se tapahtuu.
- Luotettavuuden ominaisuus *sitkeys* (robustness) määrittelee, miten ”pehmeästi” ohjelmisto kaatuu poikkeustilanteessa.

Analyysi, testaus ja laatu

- Laatua kannattaa parantaa analyysilla:
  - Staattinen analyysi sopii mihin tahansa työvaiheeseen
  - Mitä aiemmin virhe havaitaan, sitä edullisempi se on korjata
  - Rakenteiselle dokumentille on mahdollista tehdä automaattinen staattinen analyysi.
- Laatua kannattaa parantaa testaamalla:
  - Testitapausten aikainen suunnittelu auttaa löytämään vaatimusmäärittelystä ja suunnittelusta virheitä.
  - Testaus on parhaiten hallittu V&V:n muoto.

2. Yleistä testauksesta (P&Y:9)

- Nyt jätämme toistaiseksi analyysin ja keskitymme puhtaasti testaukseen.
- Ohjelmisto, jolle on tehty systemaattinen testaus, on luultavasti luotettavampi kuin ohjelmisto, joka on testattu satunnaisesti.
  - Jokainen ohjelmiston osa pienimmästä suurimpaan pitää testata *perusteellisesti* ja *systemaattisesti* ennen sen liittämistä valmiiseen tuotteeseen.
  - Mutta mitä perusteellinen ja systemaattinen testaus tarkoittaa?

Systemaattisuus ja perusteellisuus

- Systemaattisuus on helppo määritellä. Systemaattinen testaus tarkoittaa, että testaukselle on määritelty prosessi tai prosesseja, joita noudatetaan kaikessa testauksessa.
- Perusteellisuus on vaikeampi termi. Se pitäisi määritellä tarkoittamaan testausta, joka varmistaa, että ohjelmisto on oikeellinen.
  - Valitettavasti näin sitä ei voida määritellä, sillä ohjelmiston oikeellisuutta ei voida verifioida.

Riittävyys

- Perusteellisuuden sijaan puhutaan *riittävyydestä* (adequacy). Testaus on riittävää, kun se täyttää sille asetetut ehdot.
- Riittävyys on mukava termi, sillä sen määritelmä riippuu testattavan ohjelmiston lisäksi testaavasta organisaatiosta:
  - testaus on riittävää, kun sitä on tehty yrityksen strategian kannalta tarpeeksi.
- Toki yleensä kunkin yrityksen määrittelemä testauksen riittävyys riippuu testausteoriasta, mutta sen ei tarvitse tehdä sitä.

Ohjelmistojen testaus / Taina 27

Testauksen terminologiaa

- *Ohjelma* (program): testattava suorituskelpoinen osa. Ohjelman koko voi vaihdella sovelluksesta metodiin.
- *Testitapaus* (test case): Joukko syötteitä, suoritusehtoja ja hyväksymisehto.
- *Testitapausmäärittely* (test case specification): Jokin vaatimus (requirement), jonka yksi tai usea testitapaus täyttää.
- *Testipaketti* (test suite): Testitapausten joukko, joilla yleensä on yhteinen tavoite.
- *Testi tai testin suoritus* (test or test execution): Ohjelman suorittaminen testitapauksen syötteillä ja tuloksen arviointi testitapauksen hyväksymisehdolla.
- *Riittävyysehto* (adequacy criterion): totuusarvo tosi/epätosi parille <ohjelma,testipaketti>. Riittävyysehto kertoo, onko ohjelmalle tehty testipaketti testannut ohjelmaa riittävästi.

Ohjelmistojen testaus / Taina 28

Testitapaus

- Testitapaus sisältää kaiken sen informaation, joka tarvitaan sitä vastaavan testin suoritukseen ja tulosten analysointiin.
  - Testitapauksen syöte voi olla mikä tahansa ohjelman tapahtuma, esimerkiksi keskeytys.
  - Testitapauksen suoritusehto on ehto, jonka on oltava voimassa testauksessa, esimerkiksi muuttujien alustus.
  - Testitapauksen hyväksymisehto on ehto, jonka on oltava voimassa testin suorituksen jälkeen. Hyväksymisehto kertoo, menikö testi läpi.

Testitapausmäärittely

- Jos testitapauksen analogia ohjelmistotuotannossa on valmis ohjelmisto, niin testitapausmäärittelyn analogia on ohjelmiston vaatimusmäärittely.
- Testitapausmäärittely listaa ne vaatimukset, jotka sen mukaisten testitapausten tulee täyttää.
- Yhden testitapausmäärittelyn voi täyttää usea testitapaus ja testitapaus voi täyttää usean testitapausmäärittelyn.

Riittävyysehto

- Testipaketti täyttää riittävyysehdon, jos sen jokainen suoritettu testi täytti hyväksymisehdon ja kaikki testipaketille asetetut vaatimukset täyttyivät testeillä.
  - Oppikirja puhuu *testien sitoutumisesta* (test oblication): testin sitoutuma on jokin täsmällinen ehto, jonka testitapauksen tulee täyttää.
  - Esimerkiksi jos testipaketin riittävyysehto on, että jokainen metodin lause suoritetaan ainakin kerran, niin tietyn testin sitoutuma voi määritellä, minkä osan metodin lauseista kyseisen testin tulee ainakin suorittaa.

Ohjelmistojen testaus / Taina 31

3. Toiminnallinen testaus (P&Y: 10)

- Ohjelman toiminnallinen määrittely (functional specification) on tärkein testitapaus-määrittelyjen syöte.
- Toiminnallisesta määrittelystä johdettujen testien suoritusta sanotaan *toiminnalliseksi testaukseksi* (functional testing) tai *mustalaatikkotestaukseksi* (black-box testing).
- Toiminnallisessa testauksessa vain ohjelmaa kuvaavat määrittelyt vaikuttavat testaukseen. Ohjelman rakenteella ei ole merkitystä.

Ohjelmistojen testaus / Taina 32

Miksi toiminnallista testausta?

- Toiminnallinen testaus (tt) on perustestausta:
  - Tt aloitetaan heti vaatimusmäärittelyssä suunnittelemalla testitapausmäärittelyjä.
  - Tt:lla löydetään sellaisia virheitä, joita on vaikea tai mahdoton löytää muilla keinoilla.
  - Tt:n tekniikoita voidaan soveltaa riippumatta siitä, millä tasolla ohjelman toiminnallinen määrittely on annettu.
  - Tt:n tekniikat sopivat kaikenkokoisten ohjelmien testaamiseen metodeista ohjelmistoihin.
  - Tt on yleensä halvin ja helpoin suunniteltava.

Ohjelmistojen testaus / Taina 33

Ositus

- Toiminnallinen testaus perustuu *ositukseen* (partitioning). Testattavan ohjelman mahdolliset toimintatavat ositetaan homogeenisiksi luokiksi, joista jokainen kuvaa yhtenevän tavan käyttää ohjelmaa.
- Jokaista luokkaa kohti suunnitellaan sopivat testitapausmäärittelyt ja kirjoitetaan niihin sopivat testitapaukset.
- Jos ositus on tehty hyvin, luokat ovat ulkoisesti erillisiä ja sisäisesti yhteneviä:
  - Ulkoisesti erillisiä = luokkien leikkaukset ovat tyhjiä.
  - Sisäisesti yhteneviä = luokan sisällä kaikki ohjelman toimintatavat käyttävät ohjelmaa samalla tavalla.

Ohjelmistojen testaus / Taina 34

Ositus 2

- Koska ohjelman toiminta riippuu vain syötteistä, ositus tehdään syötteille.
  - Esimerkiksi palvelupyyntö, skedulointi ja keskeytys ovat syötteitä.
- Onnistuneessa osituksessa osajoukon toiminta millä tahansa osajoukon syötteellä on yhtenevää.
  - Esimerkiksi ohjelmalla, joka palauttaa syötteenä saamansa luvun x käänteisluvun, ositus voi olla luvut x != 0 ja x=0.
  - Ehdolla x != 0 ei ole väliä, mikä luku valitaan testitapaukseksi. Ohjelman pitäisi käyttäytyä kaikilla luvuilla x != 0 samalla tavalla.
  - Jos ohjelma ei käyttäydy arvoilla x != 0 samalla tavalla, ositus (tai ohjelma) on pielessä.

Testitapaukset ja toiminnallinen määrittely

- Testitapausten johtaminen toiminnallisesta määrittelystä on monivaiheinen prosessi.
- Periaatteessa testitapaukset voitaisiin johtaa suoraan toiminnallisesta määrittelystä, mutta tällöin
  - testitapauksia tulee hyvin paljon,
  - suuri osa testitapauksista on päällekkäisiä,
  - riittävyysehtoja on vaikea arvioida ja
  - tuloksena ei ole (helposti) toistettava prosessi.

Systemaattinen toiminnallinen testaus

- Toiminnallisen testauksen prosessi on systemaattinen ja toistettava. Se sisältää seuraavat työvaiheet:
  1. Riippumattomien testattavien ominaisuuksien tunnistus.
     - Ositetaan ohjelman toiminnallisuus sellaisiksi osajoukoiksi, jotka ovat mahdollisimman itsenäisiä.
  2. Testattavan ominaisuuden testiarvojen valinta tai testattavan mallin generointi.
     - Toiminnallisuusosajoukkoa edustavien syötteiden arvot luetteloidaan. Yleensä arvoja on paljon (jopa ääretön määrä), joten pääosin arvot kuvataan joukko-opin keinoin.
     - Jos toiminnallisuusosajoukosta on olemassa käyttökelpoinen toiminnallisuutta kuvaava malli tai malli on generoitavissa helposti, käytetään mallia. Mallin parametrien arvojoukot vastaavat syötearvojoukkoja.

Systemaattinen toiminnallinen testaus 2

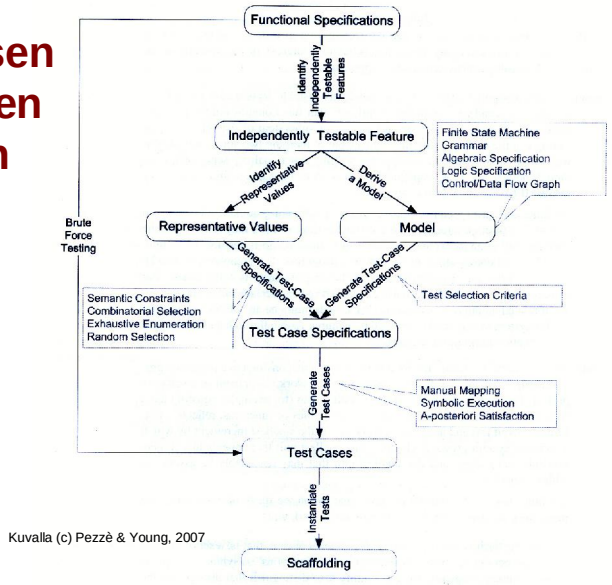
3. Testitapausmäärittelyjen generointi.
   - Kaikki testitapausmäärittelyt ovat edellisen kohdan syötearvojoukkojen tai mallin parametrien arvojoukkojen karteesinen tulo.
   - Koska karteesisen tulon koko kasvaa syötearvojoukkojen määrän suhteen eksponentiaalisesti, kaikkien testitapausmäärittelyjen joukko kasvaa nopeasti hallitsemattoman suureksi.
   - Edellisen kohdan johdosta testitapausmäärittelyiden joukkoa pitää *karsia* (prune). Karsinnassa testitapausmäärittelyistä valitaan sopivalla heuristiikalla osajoukko, jonka mukaiset testitapaukset testaavat toiminnallisuusosajoukkoa riittävän hyvin.

Systemaattinen toiminnallinen testaus 3

4. Testitapausten generointi.
   - Jokaista valittua testitapausmäärittelyä kohti generoidaan yksi tai useampi testitapaus.
5. Testien suoritus.
   - Generoidaan/toteutetaan tarvittava ohjelman toimintaympäristö. Tämä on tarpeen, jos testattava ohjelma ei pysty toimimaan yksin.
   - Suoritetaan testit ja analysoidaan tulokset.

- Tärkein työ on testitapausmäärittelyiden karsinta. Ilman sitä testaus on käytännössä mahdotonta, koska testejä tulee liikaa.

Systemaattisen toiminnallisen testauksen prosessi



Kuvalla (c) Pezzè & Young, 2007

4. Kombinaatiotestaus (P&Y: 11)

- Toiminnallinen määrittely palvelee hyvin erilaisia sidosryhmiä, joista testaajat ovat vain yksi. Näin määrittely ei yleensä ole sellaisessa muodossa, että siitä voidaan johtaa testitapauksia.
- Kombinaatiotestauksessa toiminnallisen määrittelyn riippumattomat osat jäsennetään ominaisuuksiksi tai attribuuteiksi, joiden arvoja voidaan systemaattisesti vaihdella ja joiden kombinaatiot voidaan laskea.

Kombinaatiotestauksen tekniikat

- Oppikirja listaa kolme kombinaatiotestaustekniikkaa:
  - Luokittelutestaus (Category-Partition testing)
  - Parittainen kombinaatiotestaus (Pairwise combination testing)
  - Luettelopohjainen testaus (Catalog-Based testing)
- Tekniikoista luokittelutestaus on yleisin. Parittainen kombinaatiotestaus antaa luokittelutestausta vähemmän testitapauksia. Luettelopohjainen testaus helpottaa automaattista testitapausten generointia.

Luokittelutestaus

- *Luokittelutestaus* (Category-Partition testing) on yleisin toiminnallisen testauksen tekniikka.
- Luokittelutestauksessa testattavan riippumattoman ohjelman osan syötteet ositetaan (partition) sellaisiksi luokiksi (categories), että kaikilla luokan syötealkioilla testattavan ohjelman toiminta on samanlaista.
- Kun luokat ja syötteet on valittu, lasketaan kaikkien parametrien luokkien välinen karteesinen tulo eli lasketaan kaikki mahdolliset kombinaatiot.
- Koska kombinaatioita on liikaa, niiden määrää karsitaan käyttämällä hyväksi testattavasta ohjelmasta tiedettävää semanttista informaatiota.

Luokittelutestauksen algoritmi

- Käytännössä luokittelutestaus menee näin:
  1. Katsotaan, mitä syötteitä testattava ohjelma saa.
  2. Ositetaan kunkin syötteen arvojoukko luokiksi, joiden sisällä syötteet vaikuttavat ohjelmaan samalla tavalla.
  3. Annetaan jokaisen syötteen jokaiselle luokalle säännöt, jotka rajaavat luokan muista syötteen arvojoukon luokista.
  4. Arvioidaan, montako arvoa kustakin luokasta tarvitaan testaukseen.
  5. Lasketaan, montako kombinaatiota luokkien syötteistä tulee.
  6. Jos kombinaatioita on resursseihin nähden liikaa, niitä karsitaan semanttisilla säännöillä (tai kokemukseen perustuvalla ammattitaidolla).
  7. Jäljelle jääneille kombinaatioille laaditaan testitapausmäärittelyt ja niiden mukaiset testitapaukset.

Esimerkki: Laajennettu käyttötapaus

- Robert Binder esittää kirjassaan Testing Object-Oriented Systems *laajennetun käyttötapauksen* (extended use-case) käsitteen.
- Laajennetussa käyttötapauksessa kunkin käyttötapauksen skenaarion syötteet ja tulosteet on eritelty arvojoukkoineen.
- Kaikista käyttötapauksen skenaarioista tehdään päätöstaulu, josta näkyvät syöte- ja tulostietojen lisäksi niiden välinen suhde.

Pankkiautomaattiin kirjautuminen

- Olkoon meillä esimerkiksi käyttötapaus, jolla kuvataan pankkiautomaattiin kirjautuminen.
- Käyttötapauksen syötteet ja tulosteet ovat:
  - Pankkikortin todellinen tunnus: neljä numeroa / / lopetettu kortti / jotain muuta
  - Automaattiin syötetty tunnus: oikea / väärä
  - Pankin yhteyskuittaus: yhteys ok, ei yhteyttä
  - Tilin tila: aktiivinen, suljettu
  - Vastausviesti: joukko ruudulla näkyviä viestejä.

Pankkiautomaatin käyttötapauksen päätöstaulu

N:o	Kortin tunnus	Syötetty tunnus	Pankin kuittaus	Tilin tila	Vastausviesti	Automaatin toiminta kortille
1	Virheel-linen	-	-	-	Syötä pankkikortti	Palauttaa kortin
2	Oikea	Oikea	OK	Suljettu	Ota yhteyttä pankkiin	Palauttaa kortin
3	Oikea	Oikea	OK	Avoin	Valitse toiminta	Kortti pysyy sisällä, toiminta jatkuu
4	Oikea	Oikea	Ei yhteyttä	-	Yhteys poikki	Palauttaa kortin
5	Oikea	Väärä	-	-	Syötä tunnus uudestaan	Kortti pysyy sisällä, toiminta jatkuu
6	Lopetettu	-	OK	-	Kortti lopetettu	Syö kortin
7	Lopetettu	-	Ei yhteyttä	-	Kortissa vikaa	Palauttaa kortin

Skenaarioiden suora testaus

- Käyttötapauksen mukaan kirjautumisen pitää toimia edellisellä tavalla. Periaatteessa tästä voidaan johtaa testitapaukset yksinkertaisesti valitsemalla kullekin skenaariolle sopivat parametrit.
- Saman asian voi tehdä yleisemmin kombinaatiotestauksella. Tällöin skenaarioiden sijaan katsotaan syötteiden ja tulosteiden arvojoukkoja.

Syötteiden arvojoukkojen ositus

- Mitä syötteitä testattava ohjelma saa? Tehty!
- Ositetaan syötteet luokiksi:
  - Luokittelu näkyy kalvolla 46 syötteiden perässä
  - Myös yhteyskuittaus ja tilin tila ovat syötteitä.
- Annetaan luokille säännöt:
  - numero = 0,1,2,3,4,5,6,7,8,9
  - muiden luokkien arvojen täytyy vastata vastaavaa syötettä kohti lueteltuja arvoja (esim. automaattiin syötetyn tunnuksen luokat ovat oikea ja väärä tunnus, joita vastaavat arvot ovat oikea arvo ja mikä tahansa muu arvo)

Luokkien kokojen arviointi

- Arvioidaan, montako arvoa luokista tarvitaan testaukseen:
  - Todellinen tunnus ok: 1 arvo
  - Todellinen tunnus lopetettu kortti: 1 arvo
  - Todellinen tunnus roskaa: 1 arvo
  - Syötetty tunnus ok: 1 arvo
  - Syötetty tunnus väärä: 1 arvo
  - Yhteys ok: 1 arvo
  - Ei yhteyttä: 1 arvo
  - Aktiivinen tili: 1 arvo
  - Suljettu tili: 1 arvo
  - (Vaikka kaikista luokista riitti yksi arvo, tämä ei ole yleinen sääntö. Esimerkiksi aidoista arvoväleistä kannattaa valita arvot välien raunalta ja ehkä keskeltä.)

Kombinaatioiden karsinta

- Lasketaan tarvittavien kombinaatioiden määrä:
  - Kustakin luokasta otetaan vain yksi arvo, joten kombinaatioiden määrä on luokkien karteesisen tulon koko: 3*2*2*2 = 24 kombinaatiota.
  - 24 kombinaatiota ei ole kovin paljon, joten kaikki tapaukset voitaisiin tutkia. Osa kombinaatioista on kuitenkin mahdottomia: meillä on semanttista informaatiota, jolla voimme karsia kombinaatioita.
- Karsitaan kombinaatioita semanttisilla säännöillä:
  - Jos todellinen tunnus on roskaa, niin syötetty kortti ei ole pankkikortti. Tällöin automaatti vain sylkee kortin ulos, ja muilla arvoilla ei ole väliä.
  - Jos kortti on lopetettu, syötetyllä tunnuksella ja tilin avauksella ei ole väliä. Automaatti syö kortin, jos se saa varmennettua lopetuksen.
  - Jos syötetty tunnus ei vastaa kortin tunnusta, niin pankkiin ei oteta yhteyttä. Tällöin yhteyden tilalla ja tilin aktiivisuudella ei ole väliä.
  - Jos pankkiin ei saada yhteyttä, tilin tilaa ei voida tarkastaa. Tällöin tilin laillisuudella ei ole väliä.

Testitapausmäärittelyiden ja testitapausten luonti

- Laaditaan testitapausmäärittelyt ja testitapaukset:
  - Todellinen tunnus roskaa:
    - roskaa, -, -, -: #65Ty4£,-,-,-
  - Todellinen tunnus lopetettu kortti ja yhteys ok:
    - lopetettu kortti, -, yhteys ok, -: 1234,-,OK,-
  - Todellinen tunnus lopetettu kortti ja ei yhteyttä:
    - lopetettu kortti, -, yhteys ei ok, -:1234,-,timeout,-
  - Todellinen tunnus ok, syötetty tunnus ei ok:
    - ok, väärä,-,-: 5678, 9012,-,-
  - Todellinen tunnus ok, syötetty tunnus ok, pankkiin ei saada yhteyttä:
    - ok, ok, ei yhteyttä: 5678, 5678, timeout, -
  - Todellinen tunnus ok, syötetty tunnus ok, pankkiin saadaan yhteys, tili on aktiivinen:
    - ok, ok, yhteys, aktiivinen tili: 5678, 5678, OK, Active
  - Todellinen tunnus ok, syötetty tunnus ok, pankkiin saadaan yhteys, tili on suljettu:
    - ok, ok, yhteys, suljettu tili: 5678, 5678, OK, Closed

Yhteenveto esimerkistä

- Esimerkissä sekä skenaarioilla että luokittelutestauksella päädyttiin samoihin testitapauksiin. Luokittelutestaus antoi kuitenkin enemmän informaatiota kuin skenaariopohjainen testaus:
  - Luokittelutestauksella löysimme kaikki 24 kombinaatiota.
  - Luokittelutestaus oli riippumaton löydetyistä skenaarioista. Jos jokin skenaario olisi puuttunut käyttötapauksen kuvauksesta, sitä ei olisi osattu testata skenaarioihin perustuvalla testauksella.

Luokittelutestauksen kombinaatioräjähdys

- Luokittelutestaus on erinomainen työkalu silloin, kun luokkien välille tulee luonnollisella tavalla rajoitteita. Näin juuri kävi pankkiautomaattiesimerkissä.
- Jos luonnollisia rajoitteita ei ole tarpeeksi, kombinaatioiden määrä kasvaa helposti liian suureksi. Luokittelutestauksessa saatetaan joutua käyttämään keinotekoisia rajoitteita. Tällöin testauksen laatu riippuu rajoitteiden mielekkyydestä.
- *Parittainen* ja k-*asteinen kombinaatiotestaus* (pairwise and k-way combination testing) pyrkivät ratkaisemaan edellisen ongelman.

Parittainen ja k-asteinen kombinaatiotestaus

- Luokittelutestauksen kaikkien muuttujien luokkien kombinaatioiden testaus on kombinaatiotestauksen toinen ääripää. Toinen ääripää on pitää muuttujat erillään ja testata kukin muuttujan luokka muista luokista riippumattomasti.
- Ääripäiden välissä on lukuisa joukko muita mahdollisuuksia: Voimme testata kaikki muuttujaparit, muuttujakolmikot jne.
- Yleisesti voimme testata k-asteiset kombinaatiot, missä 1 <= k <= n (n = kaikkien muuttujien lukumäärä)
  - Luokittelutestaus = n-asteinen kombinaatiotestaus
  - Erillisten muuttujien testaus = 1-asteinen kombinaatiotestaus
  - Muuttujaparien testaus = 2-asteinen kombinaatiotestaus jne.

Parittainen kombinaatiotestaus vs. luokittelutestaus

- Luokittelutestaus on luotettavampaa testausta kuin parittainen kombinaatiotestaus, sillä siinä huomioidaan kaikki kombinaatiot.
- Toisaalta parittainen kombinaatiotestaus säästää hyvin paljon testitapauksia:
  - Luokittelutestauksen vaatimien testitapausten määrä kasvaa testattavien muuttujien suhteen eksponentiaalisesti.
  - Parittaisen kombinaatiotestauksen vaatimien testitapausten määrä kasvaa testattavien muuttujien suhteen logaritmisesti.

Luokittelutestauksen esimerkin parit

- Luokittelutestauksen esimerkissä meillä olivat seuraavat muuttujat: pankkikortin tunnus, syötetty tunnus, yhteyskuittaus ja tilin tila. Näistä saadaan seuraavat parit:
  - pankkikortin tunnus, syötetty tunnus: 3*2 testiä
  - pankkikortin tunnus, yhteyskuittaus: 3*2 testiä
  - pankkikortin tunnus, tilin tila: 3*2 testiä
  - syötetty tunnus, yhteyskuittaus: 2*2 testiä
  - syötetty tunnus, tilin tila: 2*2 testiä
  - yhteyskuittaus, tilin tila: 2*2 testiä
  - Yhteensä 30 testiä.

Usean parin kattaminen yhdellä testitapauksella

- Luokittelutestauksella kombinaatioiden määrä oli 24 eli parittaisen testauksen määrää pienempi. Tämä on mahdollista, jos muuttujaa kohti ei ole kovin montaa luokkaa.
- Parittaisessa kombinaatiotestauksessa on kuitenkin mahdollista kattaa samalla testitapauksella usea pari. Syynä on se, että testitapaukseen on joka tapauksessa annettava arvo kaikille muuttujille.

Yhdistetyt testitapaukset

- Vaikkapa seuraavilla testitapausmäärittelyillä saadaan katettua kaikki luokittelutestauksen esimerkin parit:
  - Todellinen ok, syötetty ok, yhteys ok, tili ok
  - Todellinen ok, syötetty väärä, ei yhteyttä, tili suljettu
  - Lopetettu kortti, syötetty ok, ei yhteyttä, tili suljettu
  - Lopetettu kortti, syötetty väärä, yhteys ok, tili ok
  - Väärä kortti, syötetty ok, yhteys ok, tili suljettu
  - Väärä kortti, syötetty väärä, ei yhteyttä, tili ok

Parittaisen kombinaatiotestauksen heikkoudet

- Parittainen kombinaatiotestaus ei ole täydellinen, vaan siinä on heikkoutensa:
  - Jos testattavassa ohjelmassa on paljon riippuvuuksia, on mahdollista, että yhdistetty testitapaus ei testaa kuin osaa toiminnallisuudesta.
    - Esimerkiksi edellisen esimerkin lopetettu kortti –testit eivät oikein toimiessaan testaa mitään muuta kuin lopetettu kortti –toimintaa. Onneksi tällaisten rajoittavien testien löytäminen automaattisesti on mahdollista, kun arvoalueet ja rajoitteet on kuvattu formaalisti.
  - Optimaalisen testitapausmäärän löytäminen käsityönä on käytännössä mahdotonta, kun muuttujia ja ositusluokkia on paljon. Onneksi työvaihe on mahdollista automatisoida.
  - Parittainen kombinaatiotestaus jää helposti vajaaksi, jos ohjelmassa on monimutkaisia usean muuttujan välisiä riippuvuuksia.

Luettelopohjainen testaus

- Sekä luokittelutestauksessa että parittaisessa kombinaatiotestauksessa syötteiden ositus tehtiin manuaalisesti.
- Manuaalisessa osituksessa osituksen tekijän ammattitaito ratkaisee. Kaksi henkilöä tekee luultavasti erilaiset ositukset.
- Olisi hyvä, että samoilla lähtöehdoilla syntyisi aina samanlainen mahdollisimman optimaalinen ositus.
- Olisi vielä parempi, jos osituksen voisi automatisoida.
- *Luettelopohjainen testaus* (Catalog-Based testing) pyrkii tarjoamaan vastauksen näihin kysymyksiin.

Luettelopohjaisen testauksen perusteet

- Luettelopohjaisessa testauksessa laaditaan sääntöjä, joiden perusteella osataan osittaa tietyn tyyppinen muuttuja.
- Kun säännöt on suunniteltu huolella ja kuvattu formalismilla, niitä voidaan käyttää automaattiseen osituksen.
- Automaattisessa osituksessa ositusohjelmisto saa syötteenä muuttujat arvoalueineen ja luettelon ositussäännöistä. Ohjelmisto palauttaa tuloksena sääntöihin sopivan osituksen.

Luetteloesimerkki

- Esimerkiksi jos kokonaislukumuuttujan arvon täytyy olla tietyllä arvovälillä, luettelo voisi johtaa sille seuraavat testitapaukset:
  - Pienin arvoväliä edeltävä arvo x, x < alaraja
  - Arvovälin alaraja
  - Jokin arvo x väliltä alaraja < x < yläraja
  - Arvovälin yläraja
  - Pienin arvoväliä seuraava arvo x, x > yläraja
- Oleellista on, että kaikille arvovälimuuttujille generoitavat testitapaukset olisivat samojen sääntöjen mukaiset.

Yleinen luettelopohjaisen testauksen malli

- Luettelopohjaisessa testauksessa ohjelmalle tarkennetaan
  - Esiehdot: rajoitteet, joiden on oltava voimassa ennen ohjelman suoritusta.
  - Jälkiehdot: rajoitteet, joiden on oltava voimassa ohjelman suorituksen jälkeen.
  - Muuttujat: millä arvoilla ohjelmaa käytetään. Muuttujat voivat olla syötteitä, tuloksia tai väliarvoja.
  - Operaatiot: mitä ohjelmassa tehdään.
  - Määritelmät: mitä mnemonisia lyhenteitä käytetään.
- Kullekin esiehdolle, jälkiehdolle ja muuttujalle määritellään tyyppi.
- Jos luettelossa on tyyppiä vastaava kuvaus, niin kuvausta käytetään tyypin arvoalueen ositukseen.
- Oppikirjassa on yksityiskohtainen kuvaus luettelopohjaisesta testauksesta. Kurssille riittää menetelmän yleinen malli.

5. Rakenteinen testaus (P&Y:12)

- Ohjelman rakenne on itsessään merkittävä ja ennen kaikkea toteutuksen kanssa täysin yhtenäinen malli, jota voidaan käyttää testauksen apuna.
- Ohjelmakoodiin perustuvaa testausta kutsutaan *rakenteiseksi testaukseksi* (structural testing) tai *lasilaatikkotestaukseksi* (white-box testing, glass-box testing).
- Rakenteisen testauksen tärkein työkalu on metodin kuvaus verkkona: ohjausvuokaavio.

Ohjausvuokaaviot (P&Y:5.3)

- *Ohjausvuokaavio, OVK* (control flow graph) on metodista tehty suunnattu verkko, jonka solmut ovat ohjelmakoodin osia ja särmät ovat mahdollisia siirtymiä koodin osien välillä.
- Solmujen koko voi olla periaatteessa lauseita tai jopa konekielikäskyjä, mutta yleensä samaan solmuun laitetaan kaikki lauseet, jotka on pakko suorittaa peräkkäin. Tällaisia lausejoukkoja kutsutaan lauseiden *peruslohkoiksi* (basic block).
  - Peruslohkojen avulla OVK:ssa on mahdollisimman vähän solmuja: vähintään joka toisesta solmusta lähtee ainakin kaksi särmää.

Haarautumien kuvaaminen ohjausvuokaavioissa

- Kun OVK:ssa on haarauma, se tarkoittaa, että vastaavassa kohdassa ohjelmakoodia on ehto.
  - Ehto voi olla totuusarvoinen (if-lause, while-lause, do-while –lause, for-lause), jolloin solmusta lähtee kaksi särmää.
  - Ehto voi olla moniarvoinen (switch-lause), jolloin solmusta lähtee n tai n+1 särmää, missä *n* on case-vaihtoehtojen lukumäärä. Tapaus +1 tulee mahdollisesta default-haarasta.
  - Myös poikkeuskäsittelyssä voi olla monta haaraa, mutta näitä on vaikea mallintaa OVK:lla.

Atomiset ehdot

- Jos OVK:sta halutaan mahdollisimman täydellinen, ehdot pitää rikkoa *atomisiksi ehdoiksi* (basic condition (kirjan termi) tai atomic expression).
  - Atominen ehto on ehto, joka ei sisällä operaattoreita AND-, OR- , XOR, NOT, jne.
  - Esimerkiksi Java-ehdossa

    (((a < 0 || a > 100) && b != 0) || c == 0) && !e)

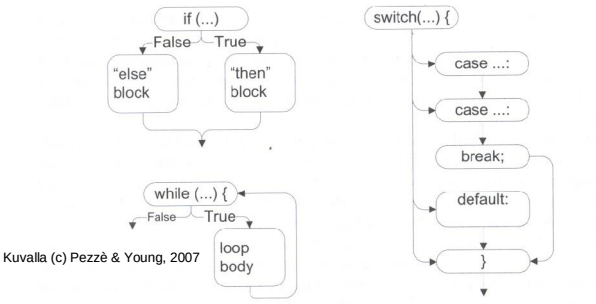
    on viisi atomista ehtoa:
    - a<0, a>100, b!=0, c==0, e

Atomiset ehdot OVK:ssa

- OVK:ssa atomiset ehdot esitetään solmuina, joiden välillä on särmät kuvaamassa evaluoinnin etenemistä atomisesta ehdosta toiseen.
  - Jos kielessä lasketaan atomisia ehtoja vain niin pitkälle, kunnes lopullinen tulos tiedetään, vain sellaiset särmät otetaan mukaan, jotka kuvaavat todellista laskentaa. Näin tehdään esim. Javassa.
  - Jos kielessä lasketaan kaikkien atomisten ehtojen arvo ennen lausekkeen arvon laskemista, jokaisesta atomisesta ehdosta lähtee kaksi särmää. Näin tehdään esim. Pascalissa.

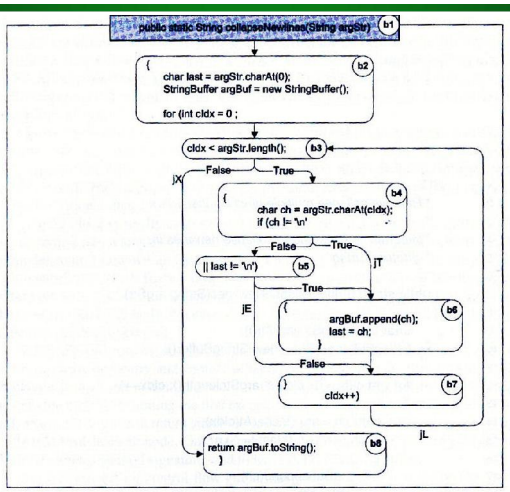
Muutama OVK:n ohjausrakenne

- Seuraavassa ovat oppikirjan versiot if-lauseesta, while-lauseesta ja switch-lauseesta:



Kuvalla (c) Pezzè & Young, 2007

OVK-esimerkki



Kuvalla (c) Pezzè & Young, 2007

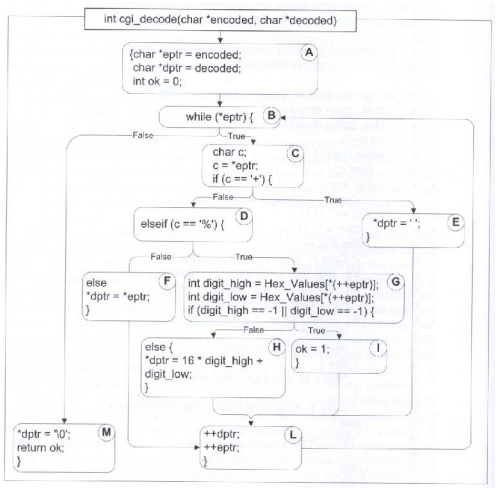
(Vastaava Java-metodi on oppikirjan sivulla 61)

Riittävyysehto ja kattavuus

- Vuokaavion avulla voimme määritellä sopivan riittävyysehdon. Riittävyysehto kertoo, onko testipaketti on testannut ohjelmaa tarpeeksi.
- Rakenteisessa testauksessa riittävyysehto ilmaistaan (rakenteisena) *kattavuutena* (coverage).
- Kattavuus on luku väliltä 0-1. Se kertoo, miten hyvin testipaketti kattaa OVK:n jollain annetulla heuristiikalla: 0 = ei kata lainkaan, 1 = kattaa täysin.

Esimerkkiohjausvuokaavio

Kuvalla (c) Pezzè & Young, 2007



Lausekattavuus

- *Lausekattavuus* (statement coverage) on intuitiivisin kattavuuksista. Siinä heuristiikkana on laskea testipaketilla suoritettuja lauseita eli OVK:n solmuja.
- Olkoon meillä testipaketti *T* ja ohjelma *P. T*:n täyttämä lausekattavuus *C* ohjelmassa *P* on
  - $C = N_T/N$, missä
    - C = lausekattavuus,
    - N_T = suoritettujen lauseiden määrä,
    - N = kaikkien lauseiden määrä.

Solmukattavuus

- OVK:ssa lausekattavuutta vastaava kattavuus on *solmukattavuus* (node coverage). Se kertoo, kuinka monessa OVK:n solmussa testipaketti käy.
- Solmukattavuus eroaa lausekattavuudesta silloin, kun solmuissa käytetään peruslohkoja.
- Esim. kalvo 73:n ohjelmalle testipaketti
  - T = {"","test","test+case%1Dadequacy"} antaa:
    - LK = 17/18 = 0,94;
    - SK = 10/11 = 0,91.

Haaraumakattavuus

- Lausekattavuus on heikoin OVK-pohjainen kattavuus. Täydellinen lausekattavuus voidaan saavuttaa ilman kaikkien haaraumien suoritusta.
- Lausekattavuutta parempi kattavuus on *haaraumakattavuus* (branch coverage). Siinä lasketaan ohjelman haaraumia.
- Olkoon meillä testipaketti *T* ja ohjelma *P*. *T*:n täyttämä haaraumakattavuus *C* ohjelmassa *P* on
  - $C = N_T/N$, missä
    - *C* = haaraumakattavuus,
    - N_T = suoritettujen haaraumien määrä,
    - *N* = kaikkien haaraumien määrä.

Vielä haaraumakattavuudesta

- Jos kalvo 73:n OVK:sta poistetaan solmu F, niin testitapauksella T={"","+%0D+%4J"} lausekattavuus LK=1, mutta haaraumakattavuus HK=7/8=0,88.
- Haaraumakattaus on hyvä ja yksinkertainen kattavuus, joka käy hyvin riittävyysehdoksi. Se on luultavasti käytetyin kattavuus.

Ehtokattavuudet

- Haaraumakattavuuden heikkous on monimutkaisissa ehdoissa. Yhtä haaraumaa kohti riittää yksi ehdon evaluointi, vaikka ehdossa olisi miten monta osaehtoa.
- Ehtokattavuuksissa monimutkaiset ehdot jaetaan atomisiksi ehdoiksi. Kattavuus lasketaan atomisten ehtojen totuusarvojen perusteella.

Perusehtokattavuus

- *Perusehtokattavuus* (basic condition coverage) on yksinkertaisin ehtokattavuus. Siinä lasketaan testipaketilla saavutettujen atomisten ehtojen tosi-epätosi-arvot.
- Olkoon meillä testipaketti *T* ja ohjelma *P. T*:n täyttämä perusehtokattavuus *C* ohjelmassa *P* on
  - $C = N_T/2*N$, missä
    - *C* = perusehtokattavuus,
    - N_T = testipaketissa suoritettujen atomisten ehtojen totuusarvojen määrä,
    - *N* = kaikkien atomisten ehtojen määrä.

Lisää perusehtokattavuudesta

- Esimerkiksi kalvo 73:n OVK:ssa on viisi atomista ehtoa: *eptr (B), c==’+’ (C), C==’%’ (D), digit_high==-1 (G), digit_low==-1 (G).
- Täyteen perusehtokattavuuteen tarvitaan testipaketti, jonka testeillä jokainen viidestä atomisista ehdoista evaluoidaan ainakin kerran sekä tosi- että epätosi-arvoiseksi.
- Esimerkiksi kalvo 73:n OVK:ssa testipaketti T={”first+test%9Ktest%K9”} riittää täyteen perusehtokattavuuteen.

Perusehtokattavuus ja haaraumakattavuus

- Täysi perusehtokattavuus ei takaa täyttä haaraumakattavuutta (ja toisin päin).
  - Esimerkiksi ehto digit_high == -1 || digit_low == -1 voidaan perusehtokattaa seuraavilla testeillä:
    - digit_high = -1, digit_low = -1 (tosi, tosi);
    - digit_high = 0, digit_low = -1 (epätosi, tosi);
    - digit_high = -1, digit_low=0 (tosi, epätosi)
  - Silti kaikilla kolmella ehdolla ehto evaluoituu arvoksi tosi, joten epätosi-haaraa ei suoriteta koskaan.
- Ongelma on helppo ratkaista yhdistämällä perusehto- ja haaraumakattavuudet. Tällöin täyteen kattavuuteen vaaditaan sekä täysi perusehtokattavuus että täysi haaraumakattavuus.

Moniehtokattavuus

- Edellisiä kattavuuksia laajempi kattavuus on *moniehtokattavuus* (compound condition coverage). Siinä testataan rakenteisen ehdon kaikkien atomisten ehtojen tosi-epätosi-kombinaatiot.
- Olkoon meillä testipaketti *T* ja ohjelma *P. T*:n täyttämä moniehtokattavuus *C* ohjelmassa *P* on
  - $C = N_T/\Sigma_i 2^{N_i}$, missä
    - *C* = moniehtokattavuus,
    - N_T = testipaketissa suoritettujen atomisten ehtojen totuusarvojen määrä,
    - Σ_i = kaikkien rakenteisten ehtojen määrä
    - N_i = i:nnen rakenteisen ehdon atomisten ehtojen määrä.

Lisää moniehtokattavuudesta

- Moniehtokattavuus sisältää sekä haaraumakattavuuden että perusehtokattavuuden.
- Joskus kaikkia moniehtokattavuuden vaatimia kombinaatioita ei ole mahdollista generoida. Mahdottomat tapaukset voidaan jättää pois listasta:
  - Esimerkiksi ehto a>=0 && a<=1 vaatisi testin, jossa a<0 ja a>1. Tämä on mahdottomuus.
- Jos kielessä lasketaan atomiset ehdot vain niin pitkälle, että tulos tiedetään, loppujen ehtojen evaluointi ei vaikuta tulokseen. Nämä merkitään viivalla '-'.
  - Edellisen esimerkin atominen ehto a>=0 on arvoilla a<0 aina epätosi, joten toista testiä 'a<=1' ei tarvitse evaluoida. Samalla päästään eroon mahdottomasta tapauksesta

Moniehtokattavuusesimerkki

- Esimerkiksi kalvo 73:n OVK:ssa tarvitaan testipaketti, jonka testit täyttävät seuraavat ehdot:
  - *eptr = 0
  - *eptr != 0
  - c = '+'
  - c != '+'
  - c = '%'
  - c != '%'
  - digit_high = 1
    - Ennakkoevaluoinnin johdosta ehtoa digit_low == 1 ei tarvitse evaluoida, kun digit_high = 1
  - digit_high != 1 && digit_low = 1
  - digit_high != 1 && digit_low != 1

Polkukattavuudet

- Tähän asti käsitellyt kattavuudet ovat perustuneet ohjelman osien erilliseen evaluointiin. Tämä ei aina riitä, vaan toisinaan tarvitaan peräkkäisten osien yhteistä evaluointia. Tällöin puhutaan *polkukattavuuksista* (path coverage).
- Jokainen ohjelman läpikäynti alusta loppuun on *polku* (path). Polku siis alkaa ohjelman alusta ja päättyy johonkin ohjelman lopuista.
  - Esimerkiksi kalvo 73:n OVK:ssa on polku (Start,A,B,M).

Yleinen polkukattavuus

- Yleinen polkukattavuus, tai vain *polkukattavuus* (path coverage), vertaa testipaketissa käytyjen polkujen määrää kaikkien polkujen määrään.
- Olkoon meillä testipaketti *T* ja ohjelma *P. T*:n täyttämä polkukattavuus *C* ohjelmassa *P* on
  - $C = N_T/N$, missä
    - C = polkukattavuus,
    - N_T = Testipaketissa läpikäytyjen polkujen määrä
    - N = kaikkien mahdollisten polkujen määrä
- Valitettavasti kaikkien mahdollisten polkujen määrä voi olla ääretön, joten kattavuutta ei voi käyttää.

Osapolut

- Kaikkien polkujen sijaan voidaan listata osapolkuja. *Osapolku* (subpath) on jokin polun osa. Se ei siis välttämättä ala ohjelman alusta eikä pääty sen loppuun.
- Osapolkujen joukko on tietenkin polkujen tapaan ääretön, mutta osapolkujen avulla on helpompaa määritellä sopivia osajoukkoja kuin kokonaisten polkujen.
- Tavallisimpia osapolkukattavuuksia ovat *silmukkakattavuus* (loop coverage) ja *tietovuokattavuudet* (data-flow coverage)

Silmukkakattavuudet

- Silmukkakattavuuksissa lasketaan osapolkuja, jotka liittyvät silmukoihin. Osapolku alkaa silmukan alusta ja päättyy silmukan loppuun.
- Jos silmukat riippuvat toisistaan, silmukkakattavuus voidaan määritellä kaikille toisistaan riippuville silmukoille.
  - Sisäkkäiset silmukat ovat toisistaan riippuvaisia.
  - Peräkkäiset silmukat ovat toisistaan riippuvaisia, jos jälkimmäisen silmukan poistumisehto riippuu edellisen silmukan poistumisarvosta.

Yksinkertainen silmukkakattavuus

- *Yksinkertainen silmukkakattavuus* (loop boundary coverage) määritellään ohjelman silmukoille. Kullekin silmukalle lasketaan
  - osapolku, jossa silmukan lopetusehto toteutuu ensimmäisellä suorituskerralla.
  - osapolku, jossa silmukka suoritetaan täsmälleen kerran.
  - osapolku, jossa silmukka suoritetaan useammin kuin kerran.
- Olkoon meillä testipaketti *T* ja ohjelma *P*. *T*:n täyttämä yksinkertainen silmukkakattavuus *C* ohjelmassa *P* on
  - $C = N_T/N$, missä
    - C = yksinkertainen silmukkakattavuus,
    - N_T = Testipaketissa läpikäytyjen silmukkaosapolkujen määrä
    - N = kaikkien mahdollisten silmukkaosapolkujen määrä

Tietovuokattavuudet (P&Y:13.1, 13.5)

- Vaikka mahdollisia (osa)polkuja on ääretön määrä, vain harva niistä paljastaa ohjelman virheen.
- Koska käytännössä koko ohjelman logiikka perustuu muuttujiin, on luultavaa, että virhe esiintyy kohdassa, missä väärin alustettua muuttujan arvoa käytetään.
- Tähän ideaan perustuvat *tietovuokattavuudet* (data flow coverage). Niissä lasketaan osapolut, jotka johtavat sovitulla tavalla muuttujan alustuksesta sen käyttöön.
- Tietovuokattavuudet ovat teoriassa tehokkaita, mutta käytännössä ne vaativat huomattavasti haarauma- ja ehtokattavuuksia kehittyneempiä – siis kalliimpia - työkaluja. Ainakin toistaiseksi on katsottu, että pieni saavutettu lisäetu ei ole kustannusten arvoinen.

6. Mallipohjainen testaus (P&Y:14)

- Luvussa 4 esitelty kombinaatiotestaus perustuu toisistaan riippumattomien syötteiden kombinaatioiden karsintaan.
- Vaikka syötteiden välillä on riippuvuuksia, niiden huomiointi on testaajan vastuulla. Riippuvuudet lisätään käsityönä SRS:n vapaamuotoisten tai osittain puoliformaalien kuvausten perusteella.
- Jos SRS:ssa käytetään formaaleja malleja, niitä voidaan käyttää kätevänä apuna syötteiden tunnistamisessa, rajoitusten määrittämisessä ja merkittävien kombinaatioiden valinnassa.

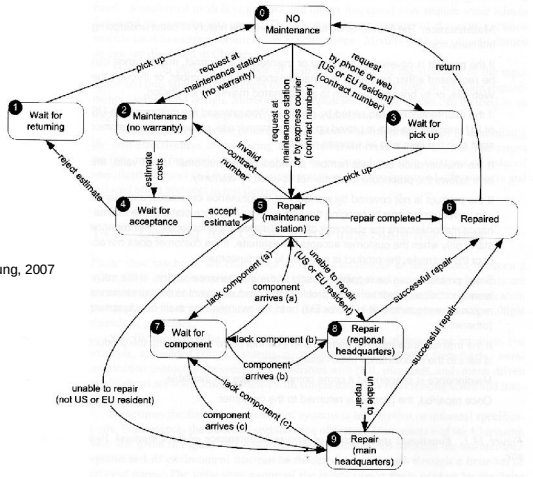
Formaalit ja puoliformaalit mallit testauksessa

- Formaalit mallit sisältävät niin paljon informaatiota, että niiden avulla voidaan tehdä automaattista testausta.
  - Formaaleja malleja ovat esimerkiksi äärelliset automaatit, päätöstaulut ja kieliopit.
- Puoliformaalit mallit vaativat manuaalista analyysia, mutta tarjoavat silti paremman lähtökohdan testitapausmäärittelyille kuin vapaamuotoiset kuvaukset.
  - Puoliformaaleja malleja ovat esimerkiksi luokkakaaviot, tietovuokaaviot ja laajennetut käyttötapaukset.

Äärellisten automaattien testaus

- Äärelliset automaatit ovat näppärä tapa kuvata järjestelmän ja ympäristön välistä yhteistyötä peräkkäisten tapahtumien sarjana.
  - Äärellinen automaatti rakentuu *tiloista* (states) ja *siirtymistä* (transitions).
  - Äärellisen automaatin kuvaama järjestelmä siirtyy tilasta toiseen saamansa herätteen avulla.
- Äärellisiä automaatteja voidaan käyttää sekä testien valinnassa että testin suorituksen validoinnissa.

Äärellisen automaatin esimerkki



Kuvalla (c) Pezzè & Young, 2007

Esi- ja jälkiehdot äärellisessä automaatissa

- Äärellisestä automaatista saadaan johdettua testitapauksia monella tekniikalla, mutta kaikki ne perustuvat tavalla tai toisella siirtymien läpikäyntiin.
  - Järjestelmän siirtyminen äärellisessä automaatissa tilasta toiseen on esi- ja jälkiehdon toteutumista. Kun tilan jälkiehto aktivoituu, siirtymä aktivoi saman esiehdon täyttävän tilan.
- Virheellinen järjestelmä voi rikkoa esi- ja jälkiehtoja eli aiheuttaa väärän siirtymän.

Historiaherkkyys

- Jos äärellinen automaatti on deterministinen, tietty siirtymä ei riipu aiemmista siirtymistä. Tällöin mikä tahansa solmu käy alkutilaksi, josta testaus etenee solmu kerrallaan.
- Valitettavasti kaikki äärelliset automaatit eivät ole deterministisiä. Aktivoituva siirtymä riippuu siitä, mitä edellisiä siirtymiä on aktivoitu. Tätä siirtymän riippuvuutta edellisistä siirtymistä kutsutaan *historiaherkkyydeksi* (history sensitivity).
  - Esimerkiksi kalvo 94:n automaatti on historiaherkkä. Tilasta 7 lähtevä siirtymä ”component arrives” riippuu siitä, minkä siirtymän kautta tilaan on tultu.
- Jos äärellinen automaatti on historiaherkkä, pelkkä yhden tilan tai siirtymän tarkastelu kerralla ei riitä. Tarvitaan siirtymäpolkuja
- Siirtymäpolkuja kannattaa tarkastella myös deterministisissä äärellisissä automaateissa, jos ei olla varmoja, että automaatilla on kuvattu kaikki mahdolliset tilat. Puuttuvien tilojen johdosta toteutettu äärellinen automaatti voi olla epädeterministinen.

Äärellisen automaatin kattavuudet

- Äärellinen automaatti on suunnattu verkko, kuten OVK. Jos äärelliselle automaatille määritellään alkusolmu, sille voidaan määritellä pitkälti vastaavat kattavuudet kuin OVK:lle.
  - Tilakattavuus (state coverage): kaikissa tiloissa on käytävä (vastaa solmukattavuutta)
  - Siirtymäkattavuus (transition coverage): kaikkien siirtymien kautta on kuljettava (vastaa haaraumakattavuutta)
  - Silmukkakattavuus (boundary interior loop coverage): kaikki silmukat on käytävä minimikierrosten, keskivertojen kierrosten ja maksimikierrosten verran (vastaa silmukkakattavuutta)

Kalvo 92:n kattavuuksien testitapauksia

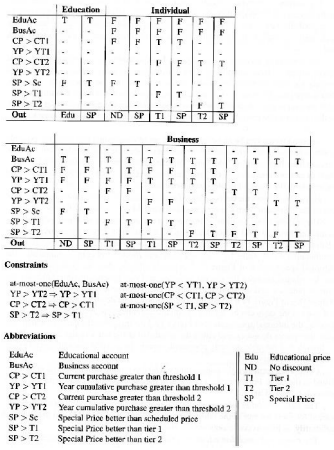
- Tilakattavuus:
  - T1: 0-2-4-1-0
  - T2: 0-3-5-7-9-6-0
  - T3: 0-5-8-6-0
- Siirtymäkattavuus:
  - T1: 0-2-4-1-0
  - T2: 0-5-2-4-5-6-0
  - T3: 0-3-5-7-9-6-0
  - T4: 0-3-5-7-5-8-7-8-9-7-9-6-0
- Silmukkakattavuus: silmukat ovat:
  - S1: 0-2-4-1-0
  - S2: 0-5-2-4-1-0
  - S3: 0-5-2-4-5-6-0
  - S4: 0-5-6-0
  - S5: 0-5-7-8-6-0
  - S6: 0-5-7-9-6-0
  - S7: 0-5-7-9-7-5-6-0
  - S8: … (kaikkien silmukoiden lukumäärän laskenta jätetään harjoitustehtäväksi (eli en jaksanut laskea niitä kaikkia – tällaiseen tarvitaan laskentaohjelma)

Päätöstaulujen testaus

- *Päätöstaulut* (decision tables) ovat hyvä formaali menetelmä kuvata monimutkaisia päätöksiä.
- Päätöstaulu on kaksiulotteinen taulukko, jossa
  - rivit kuvaavat atomisia ehtoja,
  - sarakkeet kuvaavat ehtojen kombinaatioita,
  - solmut kuvaavat atomisen ehdon totuusarvon (tosi, epätosi, ei väliä) ja
  - alin rivi kuvaa kombinaatiolla saatavan tuloksen.
- Taulukon lisäksi päätöstauluun voidaan liittää rajoitteita, jotka supistavat atomisten ehtojen kombinaatioiden määrää.

Päätöstauluesimerkki

Kuvalla (c) Pezzè & Young, 2007



	Education			Individual				
EduAc	T	T	F	F	F	F	F	F
BusAc	-	-	F	F	F	F	F	F
CP > CT1	-	-	F	F	T	T	-	-
YP > YT1	-	-	-	-	-	-	-	-
CP > CT2	-	-	-	-	F	F	T	T
YP > YT2	-	-	-	-	-	-	-	-
SP > Sc	F	T	F	T	-	-	-	-
SP > T1	-	-	-	-	F	T	-	-
SP > T2	-	-	-	-	-	-	F	T
Out	Edu	SP	ND	SP	T1	SP	T2	SP

	Business									
EduAc	-	-	-	-	-	-	-	-	-	-
BusAc	T	T	T	T	T	T	T	T	T	T
CP > CT1	F	F	T	T	F	F	T	T	-	-
YP > YT1	F	F	F	F	T	T	T	T	-	-
CP > CT2	-	-	F	F	-	-	-	-	T	T
YP > YT2	-	-	-	-	F	F	-	-	-	-
SP > Sc	F	T	-	-	-	-	-	-	-	-
SP > T1	-	-	F	T	F	T	-	-	-	-
SP > T2	-	-	-	-	-	-	F	T	F	T
Out	ND	SP	T1	SP	T1	SP	T2	SP	T2	SP

Constraints

at-most-one(EduAc, BusAc)	at-most-one(YP < YT1, YP > YT2)
YP > YT2 ⇒ YP > YT1	at-most-one(CP < CT1, CP > CT2)
CP > CT2 ⇒ CP > CT1	at-most-one(SP < T1, SP > T2)
SP > T2 ⇒ SP > T1	

Abbreviations

EduAc	Educational account	Edu	Educational price
BusAc	Business account	ND	No discount
CP > CT1	Current purchase greater than threshold 1	T1	Tier 1
YP > YT1	Year cumulative purchase greater than threshold 1	T2	Tier 2
CP > CT2	Current purchase greater than threshold 2	SP	Special Price
YP > YT2	Year cumulative purchase greater than threshold 2		
SP > Sc	Special Price better than scheduled price		
SP > T1	Special Price better than tier 1		
SP > T2	Special Price better than tier 2		

Päätöstaulukattavuuksia

- Päätöstaulu on yhtenevä rakenteisen testauksen ehdon kanssa. Näin sille voidaan määritellä samat ehtokattavuudet kuin rakenteisen testauksen ehdoille.
  - Täysi *perusehtokattavuus* (basic condition coverage) vaatii testitapausmäärittelyn jokaista päätöstaulun saraketta kohden. Ei väliä –arvot voivat olla mitä tahansa, kunhan rajoitteita ei rikota
  - Täysi *moniehtokattavuus* (compound condition coverage) vaatii testitapausmäärittelyn jokaista atomisen ehdon kombinaatiota kohden.

Muokattu ehto-/päätöskattavuus

- Sekä perusehtokattavuus että moniehtokattavuus kärsivät päätöstauluissa heikkouksista:
  - perusehtokattavuus testaa tunnistetut atomisten ehtojen kombinaatiot, mutta ei löydä puuttuvia kombinaatioita.
  - moniehtokattavuus testaa kaikki atomisten ehtojen kombinaatiot, mutta eksponentiaalisen kasvun johdosta vähänkään isommassa päätöstaulussa kombinaatioiden määrä kasvaa hallitsemattomaksi.
- Edellisiä parempi kattavuus on *muokattu ehto-/päätöskattavuus* (modified condition/decision coverage).

Päätös ja vastapäätös

- Muokatun ehto-/päätöskattavuuden idea on, että jokaista positiivista päätöstä kohti on oltava vastapäätös. Vastapäätös kertoo, mitä tehdään, kun ehdot eivät toteudu.
  - Vastapäätökset käsittelevät virhetilanteita: arvot eivät olleetkaan odotetulla tavalla.
  - Useimmat vastapäätökset ovat jo mukana päätöstaulussa, mutta m e-/p:n avulla etsintä on systemaattista.
- Vastapäätökset saadaan jokaisesta sarakkeesta kääntämällä yksi totuusarvo. Jos käännetyn totuusarvon saraketta ei löydy päätöstaulusta, se lisätään uutena sarakkeena.
- Uusien sarakkeiden lisäyksen jälkeen muokattuun päätöstauluun käytetään perusehtokattavuutta.

Esimerkki: Kalvo 45:n laajennettu käyttötapaus päätöstauluna sekä sarakkeiden 1 ja 2 vastapäätökset

	1	2	3	4	5	6	7
Kortti voimassa	-	T	T	T	T	F	F
Kortin tunnus ok	F	T	T	T	T	T	T
Syötetty tunnus ok	-	T	T	T	F	-	-
Pankiin tunnus ok	-	T	T	F	-	T	F
Tili avoin	-	F	T	-	-	-	-
Tulos	PK1	PK2	TJ1	PK3	TJ2	SK	PK5

PKx = Palauttaa kortin, TJx = Toiminta jatkuu, SK = Syö kortin

	Sarake 1		Sarake 2				
Kortti voimassa	-	-	F	T	T	T	T
Kortin tunnus ok	-	T	T	F	T	T	T
Syötetty tunnus ok	-	-	T	T	F	T	T
Pankiin tunnus ok	-	-	T	T	T	F	T
Tili avoin	-	-	F	F	F	F	T
Vastaava alkup. sarake	6	2	6	1	5	4	3

7. Oliopohjainen testaus (P&Y:15)

- Oliopohjainen ohjelmisto on rakenteeltaan selkeästi erilainen kuin proseduraalinen ohjelmisto.
  - Esimerkiksi metodit ovat yleensä lyhyitä ja yksinkertaisia verrattuna proseduureihin,
  - mutta yhteisten attribuuttien johdosta metodien välillä on enemmän sidontaa kuin proseduurien välillä.
- Koska olioparadigma sisältää proseduraalisen paradigman, kaikki tähän asti esitellyt tekniikat sopivat myös olio-ohjelmistoille.
- Olioparadigma on kuitenkin proseduraalista laajempi, joten oliopohjaisessa testauksessa on omat erityispiirteet, jotka täytyy ottaa huomioon.

Olio-ohjelmistojen erityispiirteet testauksen kannalta

- Olio-ohjelmistoja testattaessa tulee huomata seuraavat oliopiirteet:
  - Tilariippuva käytös: Olion tila täytyy huomioida testauksessa.
  - Tiedon piilotus: Testauksessa täytyy päästä käsiksi myös olion yksityisiin tietoihin.
  - Perintä: Metodin toiminta voi riippua koko perintähierarkiasta.
  - Polymorfismi ja dynaaminen sidonta: Metodin kutsu voi vaihtua suorituskertojen välillä.
  - Abstraktit luokat ja rajapinnat: Suora testaus ei onnistu, vaikka luokat/rajapinnat sisältävät suorituksen kannalta tärkeää tietoa.
  - Poikkeuskäsittely: Metodin suoritus saattaa keskeytyä tai siirtyä uuteen paikkaan ilman että siirtymä näkyy suoraan koodissa.
  - Rinnakkaisuus: Usea toisistaan riippuva metodi voi olla suorituksessa samaan aikaan. (Sama pätee proseduraalisiin ohjelmointikieliin, mutta oliomaailmassa tämä on hyvin yleistä.)

Tilariippuva käytös

- Jokaisella oliolla on tila, joka määräytyy olion attribuuttien mukaan.
- Jos metodin toiminta riippuu syöteparametrien lisäksi sen omistaman olion tilasta, pelkkä metodin testaus ei riitä. Olion tila täytyy huomioida.
- Olion tilat ja siirtymät tilojen välillä muodostavat äärellisen automaatin, joten tilariippuvan olion metodien testaukseen sopii äärellisen automaatin testaus.

Tiedon piilotus (kapselointi)

- Jotta jokainen metodi voidaan testata, sitä vastaava olio täytyy rakentaa sellaiseksi, että testauksessa
  - päästään käsiksi myös yksityisiin metodeihin,
  - päästään käsiksi myös yksityisiin attribuutteihin,
  - päästään käsiksi myös yksityisiin olion tiloihin ja
  - päästään käsiksi myös nimettömiin aliolioihin.
- Oliopohjaiseen ohjelmaan täytyy käytännössä toteuttaa testausrajapinta, jonka kautta koko olion rakenne aliolioineen on näkyvillä.

Perintä

- Metodin suoritus ei riipu vain luokasta, missä se on määritelty, vaan koko perintähierarkiasta.
  - Toimiva yläluokka ei takaa, että aliluokan olio toimii oikein - *ei edes niiltä metodeilta, joita ei ole muokattu perinnässä.*
  - Laajojen perintähierarkioiden kaikkien (sivu)vaikutusten ymmärtäminen on erittäin vaikeaa, joten ne ovat potentiaalisia virhelähteitä.
  - Perintään liittyy metodin nimen kuormitus, jolloin metodin toiminta riippuu perintähierarkiasta.
    - C++:ssa jopa operaattoreita voidaan kuormittaa. Tällä tavalla on mahdollista tehdä ohjelmisto, johon ei voida soveltaa juuri mitään tavallisista rakenteisen testauksen menetelmistä!
- Perintä vaikeuttaa luokan testausta. Luokalla voi olla vain muutama oma metodi, mutta perinnän kautta satoja perittyjä ominaisuuksia.

Polymorfismi ja dynaaminen sidonta

- Olio-ohjelmassa metodikutsu voidaan sitoa dynaamisesti metodiin suoritusaikana. Tästä tulee uusia haasteita testaukselle:
  - Kaikki mahdolliset sidonnat on testattava
  - Jos sidonnat riippuvat toisistaan, tarvitaan sidontojen kombinaatioiden testauksia.
  - Jos sidonta riippuu olion tilasta, tarvitaan sidontojen tilariippuvaa testausta.
- Polymorfismi ja dynaaminen sidonta ovat ehkä (operaattoreiden kuormituksen lisäksi) vaikeimmin testattava oliopiirre, sillä ennen testien suunnittelua ohjelman rakenteen lisäksi täytyy vähintään analysoida mahdolliset sidontakombinaatiot.

Abstraktit luokat ja rajapinnat

- Koska abstraktia luokkaa ja rajapintaa ei voi testata suoraan, ne testataan aliluokkien avulla.
- Aliluokkien avulla testattaessa tulee huolehtia siitä, että kaikki rajapinnan tai abstraktin luokan ominaisuudet tulee testattua.
- Tarvittaessa voidaan generoida keinotekoinen aliluokka, jonka ainoa tarkoitus on testata abstrakti luokka/rajapinta.

Poikkeuskäsittely

- Poikkeuskäsittely on teoriassa hankalaa, mutta käytännössä hallittavaa.
  - Poikkeuskäsittely on teoriassa hankalaa, koska teoriassa mikä tahansa poikkeus voi tapahtua missä tahansa kohdassa ohjelmakoodia.
  - Käytännössä tiedetään aika hyvin, missä päin koodia poikkeukset tapahtuvat ja mitä poikkeuksista seuraa.
  - Testauksessa on tärkeää testata kaikki poikkeukset ja poikkeuskäsittelijät.
  - Eniten ongelmia tulee silloin, kun metodi palauttaa poikkeuksen, jonka kutsunut metodi käsittelee. Tällöin poikkeus ja poikkeuskäsittelijä ovat eri metodeissa, mutta ne tulee testata yhdessä.

Vielä poikkeuskäsittelystä

- Poikkeuskäsittelyä ei kannata kuvata OVK:n avulla. Esimerkiksi Javan try-catch-lohkot kannattaa OVK:ssa kuvata tavallisina peruslohkoina.
- Ohjelmavirheiden poikkeuksia ei tarvitse testata. Tärkeämpää on verifioida, että ohjelmavirheiden poikkeuksia ei tule.
  - Jos ohjelmavirheelle on poikkeuskäsittelijä, se pitää testata. Tällaista käsittelijää tarvitaan esimerkiksi tilanteissa, jossa palvelupyynnön esittäjään ei luoteta.
- Poikkeustilanteet, jotka eivät ole ohjelmavirheitä, täytyy testata. Tällaisia ovat esimerkiksi levytilan tai muistin loppuminen, tiedoston avauksen epäonnistuminen ja tiedoston loppumerkin luku.
- Jos metodi palauttaa poikkeuksen, eli siinä ei ole poikkeuskäsittelijää, metodi ei saa aiheuttaa muita sivuvaikutuksia. Testaus vaikeutuu huomattavasti, jos tämä ehto ei toteudu.
- Poikkeusketjuja (poikkeuskäsittelijä generoi poikkeuksen) ei tarvitse testata, paitsi jos sellaisia on erityisesti lueteltu ohjelman määrittelykuvauksessa.

Rinnakkaisuus

- Koska olio-ohjelmat eivät tarvitse ohjaavaa pääohjelmaa, ne voidaan helposti hajauttaa usealle rinnakkaiselle prosessille.
- Rinnakkaisuudesta tulee jälleen haasteita:
  - Kaikki mahdolliset metodien skeduloinnit on testattava.
  - Kaikki mahdolliset äärellisten automaattien skeduloinnit on testattava.
  - Kaikki mahdolliset resurssien käytön kilpailutilanteet on testattava.
- Onneksi rinnakkaisuuden hallinta on tunnettu vuosikymmeniä prosessien ja tietokantojen hallinnassa. Ongelma on ratkaistavissa, mutta se vaatii kurinalaista olio-ohjelmointia.

Luokka ja ryväs oliotestauksessa

- Oliopohjaisessa testauksessa pienin testattava kokonaisuus on luokka.
  - Yksittäisten metodien testaaminen erikseen ei yleensä ole järkevää, koska metodit vaikuttavat toisiinsa luokan attribuuttien kautta.
- Luokkien testaus yksinään ei yleensä riitä, sillä luokkien välillä on sidoksia.
- Toisiinsa sitoutuneita luokkia kutsutaan *ryppääksi* (cluster). Ne testataan sitten, kun niiden yksittäiset luokat on testattu erikseen.

Luokkatestauksen vaiheet

- Oppikirja luettelee kuusi vaihetta luokkatestaukselle:
  1. Jos testattava luokka on abstrakti (tai rajapinta), sitä vastaavat luokan ilmentymät generoidaan. Myös sovelluksessa valmiiksi määriteltyjä ilmentymiä voidaan käyttää.
  2. Perityille ja kuormitetuille metodeille mukaan lukien konstruktorit suunnitellaan testitapaukset, joilla varmennetaan näiden oikea käyttö.
  3. Jos luokka on tilakone, sille suunnitellaan tilasiirtymäkaavioon (eli äärelliseen automaattiin) perustuvat testitapaukset.
  4. Tilakoneen testitapauksia täydennetään halutuilla rakenteisen testauksen kattavuuksilla.
  5. Poikkeuskäsittelijöille suunnitellaan alustavat testitapaukset, joilla varmennetaan systemaattisesti testattavan luokan metodien aiheuttamat poikkeukset, käsiteltävät poikkeukset ja poikkeuskäsittelijät.
  6. Polymorfisille metodikutsuille suunnitellaan alustavat testitapaukset, jotka testaavat luokan sisäistä dynaamista sidontaa.

Luokkien välisen testauksen vaiheet

- Luokkatestauksen lisäksi ryppäät tarvitsevat luokkien välistä testausta.
  1. Ryppäät tunnistetaan.
  2. Jokaiselle ryppäälle suunnitellaan toiminnalliset testitapaukset.
  3. Suunnitellaan testitapaukset, joilla testataan tietojen kulkua ryppään sisällä.
  4. Täydennetään luokkien poikkeuskäsittelijöiden testitapauksia sellaisilla testitapauksilla, jotka käsittelevät luokkien välisiä poikkeuksia.
  5. Täydennetään polymorfisten metodikutsujen testitapauksia sellaisilla testitapauksilla, jotka käsittelevät luokkien välisiä polymorfisia testitapauksia.

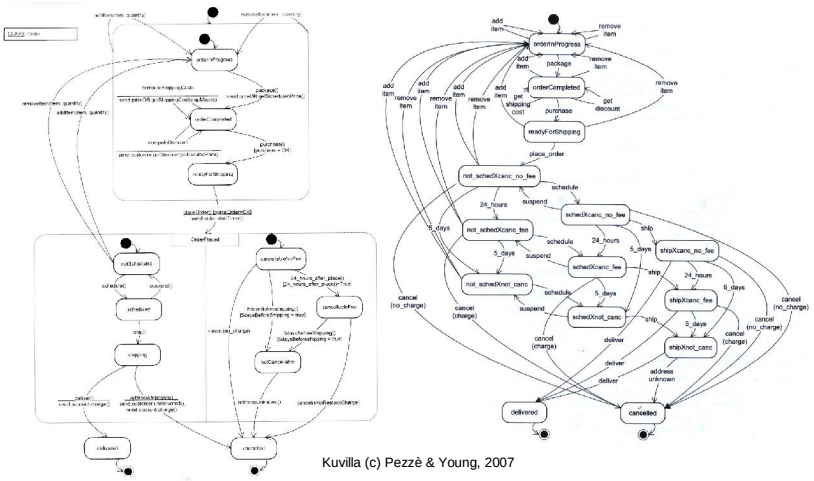
Tilakoneen testaus

- Tilakone on luokka, jossa on äärellinen määrä selvästi toisistaan eroavia tiloja. Luokan oliot siirtyvät tilasta toiseen metodien kautta.
- Tilakoneluokka on äärellinen automaatti, joten se testataan äärellisen automaatin keinoin:
  - Jokainen tila vastaa äärellisen automaatin tilaa.
  - Jokainen metodi vastaa äärellisen automaatin siirtymää.

Tilakoneen tilaräjähdys

- Jos tilakone on suunniteltu huolimattomasti tai se kuvaa monimutkaista ja monitasoista toimintaa, tuloksena saattaa olla tilaräjähdys: tiloja ja varsinkin siirtymiä tilojen välillä tulee hallitsematon määrä.
  - Tilaräjähdys syntyy tilanteissa, joissa on rakenteisia tiloja. Rakenteinen tila tarkoittaa tilaa, joka sisältää tiloja ja siirtymiä. Yleinen siirtymä rakenteiseen tilaan tarkoittaa siirtymää jokaiseen rakenteisen tilan sisäiseen tilaan.
- Tilaräjähdyksen tapauksessa tavalliset äärellisen automaatin testausmenetelmät tuottavat liikaa testitapauksia. Tällöin voidaan käyttää *yksinkertaista siirtymäkattavuutta* (simple transition coverage).
- Yksinkertaisessa siirtymäkattavuudessa vaaditaan, että jokainen siirtymä testataan ainakin kerran, mutta ei vaadita, että kaikkien rakenteisten tilojen yleisten siirtymien kombinaatiot testataan.

Tilakone-esimerkki ja vastaava äärellinen automaatti



Kuvilla (c) Pezzè & Young, 2007

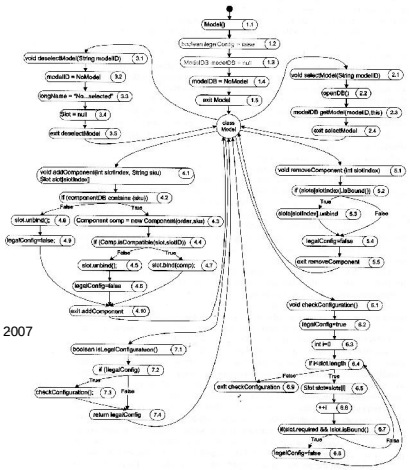
Rakenteinen oliopohjainen testaus

- Toistaiseksi listatut oliopohjaisen testauksen menetelmät ja erikoistapaukset ovat liittyneet toiminnalliseen testaukseen. Meillä on ollut (dokumentista johdettu) malli, jonka perusteella olemme tehneet testitapaukset.
- Oliopohjaiseen testaukseen tarvitaan toiminnallisen testauksen lisäksi rakenteista testausta. Näin ollen tarvitaan sopiva OVK, josta johdetaan testitapausmäärittelyjä.

Luokkatason ohjausvuokaavio

- Jos testattava luokka ei ole tilakone, OVK:n määrittely palautuu tavalliseen OVK:n määrittelyyn. Kustakin metodista tehdään oma OVK.
- Jos testattava luokka on tilakone, erilliset OVK:t eivät ole tehokas työkalu. Metodit liittyvät tilakoneen kautta toisiinsa, joten tarvitaan tilakoneen huomioiva yhteinen laaja OVK.
- Luokkatason OVK rakennetaan tekemällä kustakin metodista OVK ja liittämällä nämä OVK:t keskussolmun avulla yhteen.
  - Keskussolmu kuvaa tilanteen, missä metodien kutsujärjestyksellä on vaikutusta.
- Luokkatason OVK:n avulla voidaan kuvata kaikki tilakoneen tilat ja siirtymät. Kaaviosta tulee tosin helposti hallitsemattoman suuri.

Luokkatason ohjausvuokaavioesimerkki



Kuvalla (c) Pezzè & Young, 2007

Dynaamisen sidonnan testaus

- Koska dynaaminen sidonta päätetään suoritusaikana, sidontaan tarvitaan päätös. Tämä muistuttaa rakenteisen testauksen ehtotestausta.
- Periaatteessa dynaamisen sidonnan vaatimat kombinaatiot saadaan listaamalla vaihtoehtojen kombinaatiot ja ottamalla niistä karteesinen tulo. Tämä muistuttaa ehtotestauksen lisäksi kombinaatiotestausta.
- Kombinaatiotestauksen tapaan dynaamisen sidonnan testauksen ongelmaksi tulee vaihtoehtojen määrä.

Dynaamisen sidonnan esimerkki

- Olkoon meillä esimerkiksi seuraava määritelmä:

```
abstract class Credit {
…
  abstract boolean validateCredit(Account a, int amt,
  CreditCard c);
…
}
```

- Määritelmässä on kolme sidontaa:
  - metodi validateCredit sidotaan toteuttavassa aliluokassa,
  - parametri Account a sidotaan suoritusaikana ja
  - parametri CreditCard c sidotaan suoritusaikana.

Dynaaminen sidonta ja kombinaatiotestaus

- Kun tiedetään, mitkä kaikki luokat toteuttavat abstraktin metodin ja mitä vaihtoehtoja parametrit Account a ja CreditCard c voivat saada, voidaan laskea kaikki kombinaatiot.
  - Esimerkiksi jos aliluokkia on kolme, eri tilejä viisi ja erilaisia luottokortteja kolme, niin saadaan 3*5*3=45 eri vaihtoehtoa eli testitapausta.
- Kaikkien kombinaatioiden luettelointi vastaa luokittelutestausta. Vastaavasti voimme käyttää parittaista kombinaatiotestausta.
  - Esimerkin kohdalla parittaisessa kombinaatiotestauksessa tarvitaan 15 testitapausta.

Geneerisen luokan testaus

- Moderniin olio-ohjelmointiin kuuluu *geneerisyys* (genericity). Geneerinen luokka on sellainen, joka voidaan instantioida eri tyyppisillä todellisilla parametreilla.
- Abstraktin luokan tapaan geneerinen luokka voidaan testata vain instantiointiensa kautta.
- Geneerinen luokka on yleensä suunniteltu sellaiseksi, että sen toiminta on yhtenevää kaikilla sallituilla tyypeillä. Näin geneeristen luokkien testaus jakautuu kahtia:
  - testataan, että tietty instantiaatio toimii oikein ja
  - testataan, että kaikki mahdolliset instatiaatiot toimivat samalla tavalla.

Instantiaatioiden testaus

- Tietyn instantiaation testaus palautuu tavalliseen luokkatestaukseen. Riittää, että sekä instantioidun luokan että geneerisen luokan lähdekoodi on saatavilla. Geneeriset luokkamäärittelyt korvataan yksinkertaisesti todellisilla määrittelyillä.
- Kaikkien mahdollisten instantiaatioiden samanlaisen toiminnan varmistus vaatii
  - kaikkien mahdollisten instantiaatioiden lähdekoodin,
  - geneerisen luokan toiminnan määrittelyt ja
  - testitapaukset, jolla kustakin instantiaatiosta varmennetaan, että se toimii määrittelyn mukaan.
- Käytännössä kaikkien instantiaatioiden listaaminen on mahdottomuus saati sitten niiden toiminnan yhtenäisyyden varmistus. Geneerisessä ohjelmoinnissa on oltava tarkkana, sillä muuten tulokseksi saadaan erityisen virhealtista koodia.

8. Testien suoritus (P&Y:17)

- Siinä missä testien suunnittelu vaatii tekniikoidenkin kanssa suunnittelutyötä, testien suorituksen pitää olla automaattista.
- Suunnitellut ja toteutetut testit pitää voida suorittaa ja niiden tulokset analysoida automaattisesti.
  - Testien suunnittelu ja suoritus on analogista ohjelman suunnittelun ja käännöksen kanssa. Suunnittelu vaatii ihmistä, käännöksen lähdekoodista konekielelle tekee ohjelma.

Testitapausmäärittelyistä testitapauksiksi

- Parhaimmillaan testitapausmäärittely on niin yksityiskohtainen, että siitä johdetun testitapauksen generointi on automaattista.
- Testitapausmäärittely voi kuitenkin olla niin yleinen, että se voi vastata isoa joukkoa eri testitapauksi. Tällöin testitapauksen generointi määrittelystä on suunnittelupäätös, eli valittu testitapaus riippuu testaajasta.
- Hyvin tehty testitapausmäärittely antaa joko yksikäsitteisen testitapauksen tai niin selkeän testitapausjoukon, että testitapauksen valinta on suoraviivaista ja jopa automaattista.
  - Kurssilla esitellyt tekniikat johtavat joko suoraviivaiseen tai jopa automaattiseen testitapausten valintaan.

Testiympäristö

- Ohjelmiston kehitystyön aikana suurin osa koodista on kirjoittamatta. Silti tehty koodi pitää saada testattua, vaikka se ei olisi yksinään suoritettava osa.
- Edellisen johdosta tarvitaan *testiympäristö* (scaffolding), jonka avulla voidaan testata yksinään toimimattomia ohjelman osia.
  - Scaffolding tarkoittaa tarkkaan ottaen rakennustelineitä, mutta Suomessa puhutaan kyllä testiympäristöstä.
- Arviolta puolet projektin aikana kirjoitettavasta koodista liittyy testiympäristöön eli on poissa toiminnot tarjoavasta ohjelmakoodista.

Testiympäristön elementtejä

- Testiympäristöön kuuluvat
  - *testiajurit* (test drivers),
    - testiajuri huolehtii testitapauksen käynnistämisestä ja parametrien välittämisestä
  - *testauskehys* (test harness),
    - testauskehys tarjoaa testaukselle välttämättömän suoritus- ja analyysiympäristön
  - *tyngät* (stubs) ja
    - tyngät tarjoavat minimitoteutukset sellaisille toiminnoille, joita ei voida tai haluta testata menossa olevan testauksen yhteydessä
  - *oraakkelit* (oracles).
    - oraakkeli kertoo, menikö testi läpi

Automaattinen ja automatisoitu testaus

- Kun ohjelmistot huolehtivat koko tesitympäristöstä, kerran kirjoitettujen testien suorittaminen uudestaan on automaattista. Tällöin puhutaan *automatisoidusta testauksesta* (automated testing).
- Automatisoitu testaus on eri asia kuin *automaattinen testaus* (automatic testing). Automaattisessa testauksessa testitapaukset generoidaan automaattisesti sopivasta formaalista määrittelystä.
  - Automaattinen testaus ei onnistu yleisessä tapauksessa.
  - Automatisoitu testaus onnistuu aina mutta vaatii ohjelman ja testiympäristön toteuttajilta huolellisuutta.
  - Jos testejä ei automatisoida, niitä ei voi/kannata suorittaa uudestaan. Joskus tämä on ok, mutta yleensä ei.

Oraakkeli

- Testien automaattinen suoritus ei auta mitään, jos saadut tulokset pitää analysoida käsityönä.
- Näin myös saatujen tulosten vertaaminen haluttuihin tuloksiin on osa testiympäristöä ja siten automatisoitava.
- Sellaista ohjelmaa, joka kertoo, oliko saatu tulos oikein vain väärin, kutsutaan *oraakkeliksi* (oracle).

Lisää oraakkeleista

- Ideaalioraakkeli tunnistaa jokaisen mahdollisen testin kaikki oikeat tulokset ja siten havaitsee kaikki virheet.
  - Tällaista oraakkelia ei ole, sillä siinä tapauksessa testattavan ohjelman sijaan kannattaisi täydentää oraakkelista haluttu ohjelma.
- Käytännössä kaikki oraakkelit ovat vajaita. Toisaalta varsinkin luokkien testauksen oraakkelit ovat yleensä yksinkertaisia ja varmatoimisia, jos luokkien toiminnallisuus on yksinkertaista.
- Oraakkeleissa on eroja:
  - Ennalta määritelty oikea tulos on oraakkeli, jos määrittelyä voidaan käyttää hyväksi testiympäristössä. Tällöin varsinainen oraakkeli on itse asiassa testaaja, koska oikean tulos tulee häneltä.
  - Ohjelman aiempi versio tai samoista määrittelyistä tehty toinen ohjelma ovat oraakkeleita, jos niiden toteutus eroaa riittävästi testattavasta ohjelmasta.
  - Joskus on helpointa tehdä oraakkeli, joka antaa syötteen, kun tiedetään tulos. Tällöin generoitu syöte vastaa testitapausta, jonka tulee palauttaa tunnettu tulos.

Nauhuri

- Toisinaan testin odotettu tulos on sellainen, että sen kuvaaminen täsmällisesti koneen ymmärtämällä kielellä ei onnistu. Tällöin testaajan täytyy toimia oraakkelina.
  - Tavanomaisin tilanne tulee vastaan käyttöliittymien testauksessa. Sekä käyttöliittymätestin teko että varsinkin tuloksen analyysi vaatii arvostelukykyä.
- Vaikka testaaja toimii ensimmäisellä kerralla oraakkelina, testi voidaan silti automatisoida. Tähän tarvitaan *nauhuri* (capture and replay).
  - Nauhuri nauhoittaa testaajan tekemät toiminnot.
  - Testaaja toimii oraakkelina ja kertoo, menikö testi läpi.
  - Testiympäristö kirjaa kunkin testin nauhoituksen ja testin läpäisyn.
  - Seuraavilla testin suorituskerroilla nauhoitus on testitapaus ja kirjattu testin läpäisytieto on oraakkeli.
    - Tämä toimii tietenkin vain testeillä, jotka ovat menneet läpi. Vaikka uusi testi antaisi epäonnistuneelle testille eri tuloksen, se ei takaa, että uusi tulos on oikea.

9. Laatuprosessi (P&Y:20)

- Laadun varmennus – siis verifiointi ja validointi - on prosessi, johon pätevät samat säännöt kuin laajempiin ohjelmistoprosesseihin. Prosessin täytyy olla suunniteltu, systemaattinen ja seurattu.
- Laadunvarmennusprosessin, tai lyhyemmin *laatuprosessin* (quality process) täytyy erityisesti olla yhteensopiva kehitystyössä käytettävän prosessimallin kanssa.
  - Esimerkiksi ketterät prosessit perustuvat lyhyisiin sykleihin, kevyeen projektinhallintaan ja minimaaliseen dokumentaatioon. Tällaiseen prosessiin on vaikea liittää yksityiskohtaista formaalia tarkastuskäytäntöä.

Laatuprosessi ja ohjelmistoprosessi

- Hyvä laatuprosessi on rakenteeltaan yhtenevä koko käytettävän ohjelmistoprosessin kanssa.
  - Vesiputousmallin tapaisissa prosesseissa käytetään V-mallia (kalvo 10) tai sen varianttia.
  - XP:ssa yksikkötestit yhdistetään jokaisella syklillä osajärjestelmien ja järjestelmän testaukseen.
  - Spiraalimallin tapaisissa syklisissä prosessissa on sekä laajempaa monen syklin yli ulottuvaa testausta että tiivistä syklikohtaista testausta.
- Olivatpa prosessin yksityiskohdat mitä tahansa, laadun varmistuksessa tulee huolehtia, että virheet havaitaan ja korjataan mahdollisimman pian.
  - Mitä myöhemmin virhe havaitaan, sitä kalliimmaksi sen korjaaminen tulee. Tuotantokäyttöön päässeen virheen kustannukset voivat olla heti löytymiseen verrattuna tuhatkertaiset.

XP:n laatuprosessi

- Laatuprosessin kannalta asiakkaan läsnäolo XP-projektissa on oleellista.
  - Asiakas osallistuu vaatimusmäärittelyyn tekemällä käyttäjätarinoita, jotka toimivat sekä suunnittelun että testauksen pohjana.
  - Asiakas on vastuussa jokaisen syklin lopussa tehtävästä hyväksymistestauksesta.
  - Koska XP:n syklit ovat lyhyitä, asiakas on läsnä koko projektin ajan.
- Tiivistäen voi sanoa, että mitä paremmin asiakas on läsnä XP-projektissa, sitä luultavammin projekti onnistuu.
  - Tämä on osoittautunut ongelmaksi, sillä yleensä asiakkaalla ei ole aikaa olla mukana projektissa.
  - Jos asiakas ei ole kunnolla läsnä, prosessi käytännössä degeneroituu jonkinlaiseksi vesiputousmallin ja iteratiivisen mallin yhdistelmäksi.

XP:n laatuprosessi 2

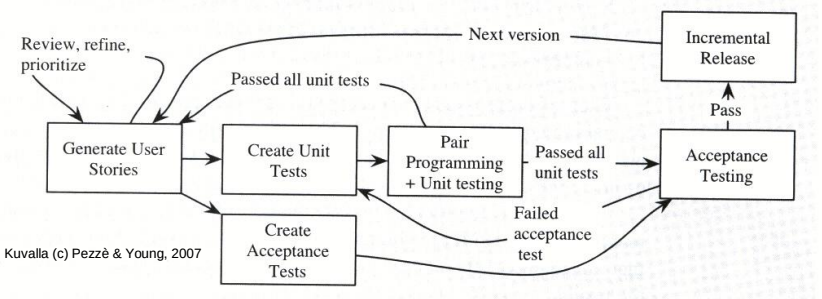
- XP:n testitapaukset perustuvat käyttäjätarinoihin. Ne korvaavat osin vaatimusmäärittelydokumentin.
  - Toisin sanoen XP:n testitapaukset ovat osa tehtävän ohjelmiston määrittelyä. Puhtaimmillaan käyttäjätarinat ja testitapaukset korvaavat vaatimusmäärittelydokumentin.
- Automatisoitavissa olevat yksikkötestit lasketaan ohjelmakoodiksi. Itse testattava koodi kirjoitetaan vasta sen jälkeen, kun kaikki sitä verifioivat testitapaukset on kirjoitettu.
  - Tätä sanotaan *testauslähtöiseksi kehitystyöksi* (test-driven development, TDD).
- Kaikkia testejä ei kirjoiteta ennen koodia. Esimerkiksi käytettävyystestit ja osin järjestelmätestit kirjoitetaan vasta sen jälkeen, kun testattava ohjelma on valmis.

XP:n laatuprosessi 3

- XP:n kehitystyö tehdään *pariohjelmointina* (pair programming). Pariohjelmointi on *katselmointi-tekniikka* (review technique), jossa toinen ohjelmoi ja toinen tarkkailee kirjoitettavan koodin laatua.
  - Katselmointitekniikat ovat staattisen analyysin tekniikoita, missä kaksi tai useampi henkilö etsii dokumentista puutteita ja virheitä.
  - Katselmointitekniikat vaihtelevat prosessin formaaliuden mukaan. Pariohjelmointi on eräs vähiten formaaleista tekniikoista. *Tarkastukset* (inspection) ovat formaalein katselmointitekniikka.
- Aiemmin tehtyä testipakettia täydennetään tämän syklin uusilla yksikkötesteillä ja kaikki testit suoritetaan syklin päätteeksi.
- Epäonnistunut hyväksymistesti tarkoittaa, että yksikkötesteissä on puutteita.

XP:n laatuprosessi 4

- Seuraavassa on XP:n laatuprosessi kaaviokuvana.
  - XP:n laatu perustuu automatisoituihin yksikkötesteihin.
  - Syklin päättymisen vaatimuksena on, että kaikki hyväksymistestit ovat menneet läpi.



Laatustrategia

- Hyvän laatuprosessin ehdoton vaatimus on onnistunut yrityksen sisäinen *laatustrategia* (quality strategy). Se määrittelee, miten laatu varmennetaan yrityksessä.
- Strategia määrittelee ainakin:
  - mitä projektien laatusuunnitelmat sisältävät,
  - mitkä ovat yrityksen sisäiset laatustandardit,
  - miten projektien laadun varmennukseen kuuluu ja
  - miten käytettyjä prosesseja valvotaan ja parannetaan.
- Lisäksi strategia voi määritellä:
  - mitä kaikille tuotteille yhteisiä laatuvaatimuksia on,
  - miten projektien etenemistä mitataan,
  - mitä tekniikota ja niitä tukevia työkaluja käytetään,
  - mitä dokumentteja tuotetaan ja
  - mitä työnkuvia (roolit ja vastuut) projekteihin kuuluu.
- Strategia voi olla kuvattuna yrityksen *laatukäsikirjana* (quality manual). Sitä seurataan ja parannetaan säännöllisillä *laatuauditoinneilla* (quality audit).

10. Integrointitestaus (P&Y: 21)

- Perinteinen testauksen V-malli (kalvot 9-10) jakaa testauksen neljään työvaiheeseen:
  - *yksikkötestaukseen* (module testing, unit testing), missä testataan pienimpiä jakamattomia yksiköitä (yleensä olioita tai proseduureja),
  - *integrointitestaukseen* (integration testing), missä testataan yksiköiden välinen yhteensopivuus,
  - *järjestelmätestaukseen* (system testing), missä testataan koko järjestelmä ja
  - *hyväksymistestaukseen* (acceptance testing), missä järjestelmä validoidaan.
- Työvaiheista integrointitestaus on vähiten arvostettu, mutta se on yhtä tärkeä kuin muut työvaiheet.

- Tehokas integrointitestaus perustuu hyvin tehtyyn yksikkötestaukseen ja katselmointeihin.
  - Yksikötestaus varmistaa, että integroitavat yksiköt toteuttavat kaikki niille asetetut vaatimukset.
  - Katselmoineilla löydetään yksikköjen rajapintojen välisiä korkean tason puutteita ja virheitä.
- Integrointivaiheessa integroitavat yksiköt on testattu niin hyvin, että yksiköiden sisäisiä ongelmia ei ole. Kaikki ongelmat johtuvat yksiköiden keskinäisestä yhteensopimattomuudesta.
- Yhteensopimattomuus voi johtua
  - vajaasti määritellystä tai toteutetusta rajapinnasta,
  - virheellisestä resurssien käytöstä tai
  - puuttuvista tai väärin toteutetuista ominaisuuksista.

Tyypillisiä integrointivirheitä

- Yhteensopimattomuudesta seuraa integrointivirheitä. Seuraavassa on listattu muutama tyypillinen vaihtoehto:
  - Epäyhtenäinen parametrien tulkinta
    - Jokaisen yksikön tulkinta voi olla järkevä, mutta tulkinnat eivät ole yhteensopivia.
  - Arvoalue- ja kapasiteettivirheet
    - Parametrin arvoalueesta tai kapasiteetista tehdään vääriä oletuksia.
  - Parametrien tai resurssien sivuvaikutukset
    - Sivuvaikutukset voivat aiheuttaa implisiittisen testaamattoman rajapinnan kahden yksikön välille, joilla ei muuten olisi mitään yhteistä.
  - Puuttuva tai väärin tulkittu toiminnallisuus
    - Oletetusta eroava toiminta voi johtaa odottamattomaan lopputulokseen.
  - Ei-toiminnalliset ongelmat
    - Vaikka yksiköt täyttävät ei-toiminnalliset vaatimukset, integrointi ei välttämättä täytä niitä. Esimerkiksi vasteaika on ei-toiminnallinen vaatimus, joka voi kärsiä syvästä rajapintahierarkiasta.
  - Ajonaikaiset yhteensopimattomuudet
    - Väärä dynaaminen sidonta voi johtaa yhteensopimattomaan rajapintaan.

Integrointitestausstrategiat

- Integrointitestauksessa yksiköitä kootaan lisäävästi pienistä kokonaisuuksista suuremmiksi, kunnes koko ohjelmisto on koottu yhteen.
  - Tavoitteena on, että yhdessä integrointitestauksen vaiheessa testataan vain yhtä rajapintaa. Toisin sanoen kerralla integroidaan vain kaksi yksikköä.
- Hyvä integrointitestaus tapahtuu yhtä aikaa kehitystyön kanssa. Integrointia tehdään heti, kun on integroitavaa.
  - Tietenkin integroitavat yksiköt on ensin testattu kunnolla. Hyvä yksikkötestaus on integrointitestauksen ehdoton vaatimus.

Perinteiset integrointitestauksen menetelmät

- Perinteiset integrointitestauksen menetelmät ovat *top-down* ja *bottom-up.*
- Menetelmissä testattavat yksiköt järjestetään käytön/sisältyvyyden mukaan.
  - Matalimmalla tasolla ovat yksiköt, jotka eivät tarvitse muilta palveluita.
  - Korkeimmalla tasolla ovat yksiköt, jotka eivät tarjoa muille palveluita.
- Top-down-testauksessa integrointitestaus aloitetaan korkeimmalta tasolta.
- Bottom-up-testauksessa integrointitestaus aloitetaan alimmalta tasolta.
- Yleensä ei tehdä puhdasta top-downia tai bottom-upia, vaan integrointitestausta tehdään samaan aikaan ylhäältä alas ja alhaalta ylös. Tämä on *voileipätestausta* (sandwich testing).

Modernit integrointitestauksen menetelmät

- Perinteiset menetelmät riittävät pienten järjestelmien integrointiin, mutta isompiin järjestelmiin tarvitaan järeämpiä keinoja.
- *Säietestauksessa* (thread integration testing) yksiköt integroidaan järjestelmän ominaisuuksien mukaan.
  - Jokainen säie kuvaa jonkun järjestelmän ominaisuuden vaatimat yksiköt.
  - Integrointitestaus tehdään säikeittäin ylhäältä alas tai alhaalta ylös.
- *Kriittisen yksikön testauksessa* (critical module integration testing) resurssit keskitetään niihin yksiköihin, jotka ovat projektin onnistumisen kannalta suurin riski.
  - Yksiköt lajitellaan riskin mukaan.
  - Integrointi aloitetaan riskialtteimmista yksiköistä.
  - Myös ulkoiset riskit, projektin riskit ja liiketoiminnan riskit voidaan huomioida.

11. Järjestelmä-, hyväksymis- ja regressiotestaus

- *Järjestelmätestaus* (system testing) on tavallaan integrointitestauksen viimeinen vaihe. Siinä testataan koko järjestelmää, mutta paino on rajapintojen sijaan järjestelmätason ominaisuuksissa.
- *Hyväksymistestaus* (acceptance testing) on validointia. Siinä testataan, miten hyvin valmis järjestelmä täyttää asiakkaan tarpeet.
- *Regressiotestaus* (regression testing) auttaa löytämään muutoksen tai uuden ominaisuuden aiheuttamia virheitä jo aiemmin testatussa koodissa.

Järjestelmätestaus

- Järjestelmätestaus perustuu vaatimusmäärittelydokumenttiin. Se on riippumatonta suunnittelun ja toteutuksen yksityiskohdista.
- Järjestelmätestauksesta voi olla vastuussa erillinen testaustiimi, mutta varsinkin ketterissä prosesseissa järjestelmätestauksen tekee usein kehitystiimi.
  - Erillinen testaustiimi on paras keino varmistaa, että suunnittelu- ja toteutusratkaisut eivät vaikuta järjestelmätestaukseen.
  - Jos järjestelmätestauksesta vastaa kehitystiimi, paras keino varmistaa järjestelmätestauksen objektiivisuus on suunnitella järjestelmätestit ennen yksikkötestejä. Tämä onnistuu myös ketterissä prosesseissa suunnittelemalla sykliin liittyvät järjestelmätestit syklin alussa.

Ohjelmistojen testaus / Taina 151

Ei-toiminnallisten vaatimusten järjestelmätestaus

- Järjestelmätestaukseen liittyy ei-toiminnallisten vaatimusten testaus.
  - Ei-toiminnalliset vaatimukset liittyvät laatuattribuutteihin. Yleensä testataan ainakin suorituskyky *rasitustestauksella* (stress testing).
    - Rasitustestauksessa järjestelmän kuormitusta lisätään tasaisesti, kunnes se ei enää selviä kuormasta. Näin selvitetään, täyttääkö järjestelmä sille asetetut suorituskykyvaatimukset.
- Jotkut ei-toiminnalliset vaatimukset ovat vaikeita tai jopa mahdottomia testata.
  - Esimerkiksi tietoturvan testaaminen on äärimmäisen vaikeaa. Toki voidaan verifioida, että kaikki tunnetut turva-aukot on tukittu, mutta tuntemattomien turva-aukkojen tukkiminen ja sisäisten väärinkäytösten estäminen on vaikeampaa.

Ohjelmistojen testaus / Taina 152

Yksikkö- integrointi- ja järjestelmätestauksen erot

	Yksikkötestaus	Integrointitestaus	Järjestelmätestaus
Testitapaukset johdetaan	Yksiköiden (luokkien) määrittelyistä	Arkkitehtuurista ja suunnittelun yksityiskohdista	Vaatimus-määrittely-dokumentista
Näkyvyystaso	Kaikki yksityiskohdat	Rajapinnat, osa koodia	Vain ulkoiset rajapinnat
Testausympäristö	Monimutkainen. Tarvitaan ajurit, tyngät ja oraakkelit	Riippuu arkkitehtuurista ja integrointi-tekniikasta. Tarvitaan ajurit ja oraakkelit, tynkiä ei välttämättä	Tarvitaan oraakkelit ja joskus käyttöympäristön simulaatio (jos järjestelmällä on paljon ympäirstöön sidottuja vaatimuksia)
Testaus keskittyy	Yksiköihin (luokkiin)	Yksiköiden yhteistyöhön	Järjestelmän toiminnallisuuteen

Hyväksymistestaus

- Hyväksymistestauksessa päätetään, onko kehitettävä järjestelmä sellaisessa kunnossa, että sen voi julkaista.
- Hyväksymistestit voidaan tehdä formaalisti:
  - hyväksymistesteillä validoidaan, että määritellyt ulkoiset ja sisäiset laatuattribuutit toteutuvat tuotteessa.
- tai ne voidaan tehdä epäformaalisti:
  - loppukäyttäjät tekevät hyväksymistestit *alfatestauksella* (alpha testing) ja *betatestauksella* (beta testing).

Alfa- ja betatestaus

- Alfa- ja betatestauksessa lopullisista käyttäjistä valitaan sopiva otos, jotka testaavat järjestelmää.
- Alfatestauksessa käyttäjät käyttävät järjestelmää laboratorio-olosuhteissa. Käyttäjien toimintaa seurataan tarkasti.
- Betatestauksessa käyttäjät käyttävät järjestelmää todellisessa ympäristössä ilman seurantaa.
- Alfa- ja betatestaus pitäisi tehdä lopulliselle tuotteelle. Usein tästä oiotaan, ja ensimmäiset alfa- ja betatestit tehdään keskeneräiselle tuotteelle tai prototyypille.
  - Tarkkaan ottaen tämä ei ole hyväksymistestausta, sillä prototyyppi tai keskeneräinen tuote ei tule tuotantokäyttöön.

Regressiotestaus

- Aina kun koodia muutetaan tai integroidaan uusia yksiköitä on riski, että aiemmin kirjoitettu koodi ei ole yhteensopivaa uuden koodin kanssa.
- Regressiotestauksella verifioidaan, että muutokset eivät ole rikkoneet jo testattua koodia.
- Yksinkertaisin regressiotestauksen tekniikka on suorittaa kaikki aiemmin tehdyt testit uudestaan. Aina tämä ei ole järkevää:
  - Testejä voi olla niin paljon, että kaikkien testien suorittaminen ei onnistu järkevässä ajassa.
  - Muutokset voivat olla sellaisia, että vanhat testitapaukset eivät ole enää yhteensopivia muokatun järjestelmän kanssa.
  - Testit voivat olla muutoksen jälkeen tarpeettomia.
- Näiden kohtien johdosta tarvitaan *testitapausten ylläpitoa* (test case maintenance). Siinä huolehditaan, että regressiotestauksen testipaketti on ajan tasalla.