TAMPEREEN AMMATTIKORKEAKOULU
Tietotekniikan koulutusohjelma
Ohjelmistotekniikka

Tutkintotyö

Miska Summala

# AUTOMATISOITU TESTAUS SYMBIAN TEXT SHELL -YMPÄRISTÖSSÄ

Työn valvoja Tony Torp, TAMK
Työn ohjaaja Tero Ahola, SYSOPENDIGIA Oyj
Tampere 2007

Tutkintotyö 42 sivua
Työn valvoja Lehtori Tony Torp, TAMK
Työn ohjaaja Ohjelmistosuunnittelija Tero Ahola, SYSOPENDIGIA Oyj
Huhtikuu 2007
Hakusanat Testaus, Testiautomaatio, Symbian, Mobiiliohjelmisto

# TIIVISTELMÄ

Tämän insinöörityön tarkoituksena on perehdyttää lukija siihen, mitä mobiiliohjelmistojen automatisoitu testaus on text shell –tasolla. Text shell –tasolla tarkoitetaan tässä yhteydessä matkapuhelimen ohjelmistokokonaisuutta, jossa on Symbian käyttöjärjestelmän lisäksi valmistajakohtaiset alemman tason komponentit mutta ei graafista käyttöliittymää. Työssä perehdytään myös yleisiin ohjelmistotuotannon periaatteisiin testausnäkökulmasta painoarvon ollessa kuitenkin testiautomaatiossa.

Työn tavoitteena on antaa mahdollisimman kattava selvitys matkapuhelimien text shell –tason testauksesta ja sen automatisoinnista. Yhtenä tarkoituksena tässä insinöörityössä on myös perehdyttää aloitteleva testausinsinööri työtehtäviinsä Symbian text shell –tason testauksessa. Työssä on myös pyritty laajentamaan kirjoittajan näkemystä ja osaamista mobiilitestiautomaation saralla.

Tutkielma sopii hyvin kaikille matkapuhelinten ohjelmistokehityksestä kiinnostuneille tahoille, ja varsinkin testausautomaatioon suuntautuneille ohjelmistoinsinööreille. Osa informaatiosta on varsin yleiskäyttöistä, ja se sopii sellaisenaan muunkinlaisen kuin mobiiliohjelmistotuotannon käyttöön.

Engineering Thesis 42 pages
Thesis supervisor Senior Lecturer Tony Torp, Tampere Polytechnic University
Thesis instructor Software Engineer Tero Ahola, SYSOPENDIGIA Plc
April 2007
Keywords Testing, Test Automation, Symbian, Mobile software

# ABSTRACT

The purpose of this engineering thesis is to explain what automated testing of mobile software is at text shell level. In this thesis text shell level means a mobile software entity with the Symbian operating system and manufacturer-specific low level components but no graphical user interface. Also the common software engineering guidelines are presented from a software testing point of view. The main point is to focus on automated software testing.

The objective of this thesis is to give an extensive description of automated mobile software testing at text shell level. One goal in this thesis is also to introduce testing engineer to his/her daily tasks related to Symbian text shell testing.

This thesis is suitable for everyone who is interested in mobile software development and especially for test automation -oriented software engineers. Some of the information is quite general, which is adequate for other software engineering purposes than only mobile software-related work.

# ALKUSANAT

Tämä insinöörityö on tehty SYSOPENDIGIA Oyj:n Telecommunications-yksikössä lokakuun 2006 ja maaliskuun 2007 välisenä aikana. Työn ohjaajana on toiminut ohjelmistosuunnittelija Tero Ahola SYSOPENDIGIA Oyj:stä ja valvojana lehtori Tony Torp Tampereen ammattikorkeakoulusta.

Kiitokset esimiehilleni ja työtovereilleni tuesta, kun olen tehnyt insinöörityötäni töiden ohella. Kiitokset myös SYSOPENDIGIAlle tuesta insinöörityön aiheen valintaprosessissa.

Kiitokset myös avovaimolleni Outille ja tyttärelleni Mintulle vankkumattomasta kannustuksesta insinöörityön kirjoittamisen aikana.

Tampereella 12.4.2007

_____
Miska Summala

# SISÄLLYSLUETTELO

**1 JOHDANTO ..... 7**
**2 OHJELMISTOTUOTANTO ..... 8**
**2.1 OHJELMISTON ELINKAARI ..... 9**
2.1.1 VAATIMUKSET JA ESITUTKIMUS ..... 9
2.1.2 VESIPUTOUSMALLIN VAIHEET ..... 10
**2.2 LAADUNVARMISTUS ..... 13**
2.2.1 MIKSI LAADULLA ON VÄLIÄ? ..... 13
2.2.2 MITEN LAATU VARMISTETAAN? ..... 13
**2.3 DOKUMENTOINTI ..... 15**
2.3.1 DOKUMENTOINNIN MERKITYS ..... 15
2.3.2 DOKUMENTTITYYPIT ..... 15
2.3.3 DOKUMENTOINTIMALLIT ..... 16
**3 OHJELMISTOTESTAUS ..... 17**
**3.1 TESTAUSTASOT ..... 18**
3.1.1 YKSIKKÖTESTAUS ..... 19
3.1.2 INTEGROINTITESTAUS ..... 20
3.1.3 JÄRJESTELMÄTESTAUS ..... 20
3.1.4 HYVÄKSYMISTESTAUS ..... 21
3.1.5 KÄYTETTÄVYYSTESTAUS ..... 22
**3.2 TESTITAPAUSTEN MÄÄRITTELY ..... 24**
3.2.1 RAKENTEELLINEN TESTAUS ..... 25
3.2.2 TOIMINNALLINEN TESTAUS ..... 26
**3.3 TESTAUKSEN RIITTÄVYYDEN ARVIONTI ..... 27**
**4 OHJELMISTOTESTAUKSEN AUTOMATISOINTI ..... 28**
**4.1 MITÄ KANNATTAA AUTOMATISOIDA? ..... 29**
4.1.1 REGRESSIOTESTAUS ..... 30
4.1.2 SAVUTESTAUS ..... 30
4.1.3 KUORMITUSTESTAUS ..... 30
4.1.4 SUORITUSKYKYTESTAUS ..... 30
**4.2 AUTOMATISOINNIN SUDENKUOPAT ..... 31**
4.2.1 RIITTÄMÄTÖN SUUNNITTELU ..... 31
4.2.2 LIIAN SUURET ODOTUKSET ..... 31
4.2.3 RIITTÄMÄTÖN PEREHDYTYS ..... 31
**5 TESTAUS SYMBIAN TEXT SHELL -TASOLLA ..... 32**
**5.1 TESTIKEHYS ..... 34**
5.1.1 EUNIT TESTIKEHYSJÄRJESTELMÄ ..... 35
**5.2 TESTIEN SUORITTAMINEN ..... 36**
5.2.1 TESTIYMPÄRISTÖN LUOMINEN ..... 36
5.2.2 MANUAALISET TESTIAJOT ..... 37
5.2.3 AUTOMATISOIDUT TESTIAJOT ..... 37
**5.3 TESTITULOSTEN RAPORTOINTI ..... 38**
5.3.1 QUALITY CENTER ..... 38
**5.4 KUN TESTAUKSEN AVULLA LÖYDETÄÄN VIRHE ..... 39**
5.4.1 VIRHEEN AIHEUTTAJAN KARTOITUS ..... 39
5.4.2 VIRHEEN PAIKANTAMINEN ..... 39
5.4.3 VIRHEEN RAPORTOINTI ..... 40
**6 YHTEENVETO ..... 41**

# KÄYTETYT LYHENTEET JA TERMIT

| | |
|---|---|
| **BT** | BlueTooth, lyhyen kantaman radiotekniikkaan perustuva langaton tiedonsiirtomedia. |
| **DLL** | Dynamic Link Library, ns. jaettu kirjasto. |
| **S60** | Nokian valmistama matkapuhelinkäyttöliittymä ja -alusta. Työn kirjoitushetkellä yleisin käytössä oleva alusta. |
| **S80** | Nokian valmistama matkapuhelinkäyttöliittymä ja -alusta. Ei enää tuotantokäytössä, kuitenkin SYSOPENDIGIAn ylläpitämä. |
| **TCP / IP** | Tietoliikenneprotokollaperhe, jossa kuljetuskerroksena on TCP ja verkkokerroksena IP. |
| **UI** | User Interface, käyttöliittymä |
| **UIQ** | Sony Ericssonin Symbian-käyttöjärjestelmän päällä oleva käyttöliittymätaso. |
| **USB** | Universal Serial Bus, sarjaväyläarkkitehtuuri usean laitteen välistä tiedonsiirtoa varten. |

# 1 JOHDANTO

Matkapuhelinteollisuus on nykyään kasvanut erittäin suureksi teollisuuden haaraksi koko maailmassa. Sen merkitys varsinkin Suomelle on Nokia Oyj:n ja sen alihankkijoiden kautta on merkittävä niin työpaikkojen kuin verotulojenkin osalta. Matkapuhelinten kehityksestä suuri osa on ohjelmistotuotantoa, josta suuri osa on matkapuhelinten ohjelmistojen testausta. Tätä prosessia pyritään jatkuvasti kehittämään ja tehostamaan testiautomaation avulla.

Ohjelmiston testausta tehdään periaatteessa koko sen kehityselinkaaren ajan. Tässä insinöörityössä on kattava selvitys testauksen V-mallista ja testauksen automatisoinnista yleensä. Korkealla tasolla matkapuhelinohjelmistojen testauskäytännöt eivät juurikaan poikkea yleisistä malleista, vaikka mobiililaitteissa testauksen painopiste onkin hieman toisenlainen kuin esimerkiksi x86-arkkitehtuuriin suunnitelluissa sovelluksissa. Mobiililaitteissa on ensiarvoisen tärkeää oikein toteutettu muistin- sekä resurssienhallinta. Tämä insinöörityö keskittyy Symbian text shell –tason testaukseen ja sen automatisointiin.

Tämän insinöörityön tilaajana on toiminut SYSOPENDIGIAn telekommunikaatio-divisioona, jonka alaisuudessa myös itse työskentelen. Työn aihe löytyikin oman työni kautta, joka liittyy text shell –tason testiautomaatioon. Projektin asiakas on määritellyt projektihenkilöstölle tiukat salassapitomääräykset, joka on hyvin yleistä tällä alalla. Tämä osaltaan vaikuttaa myös työn sisältöön, mutta onnistuneen rajauksen ansiosta tästä työstä voitiin tehdä julkinen.

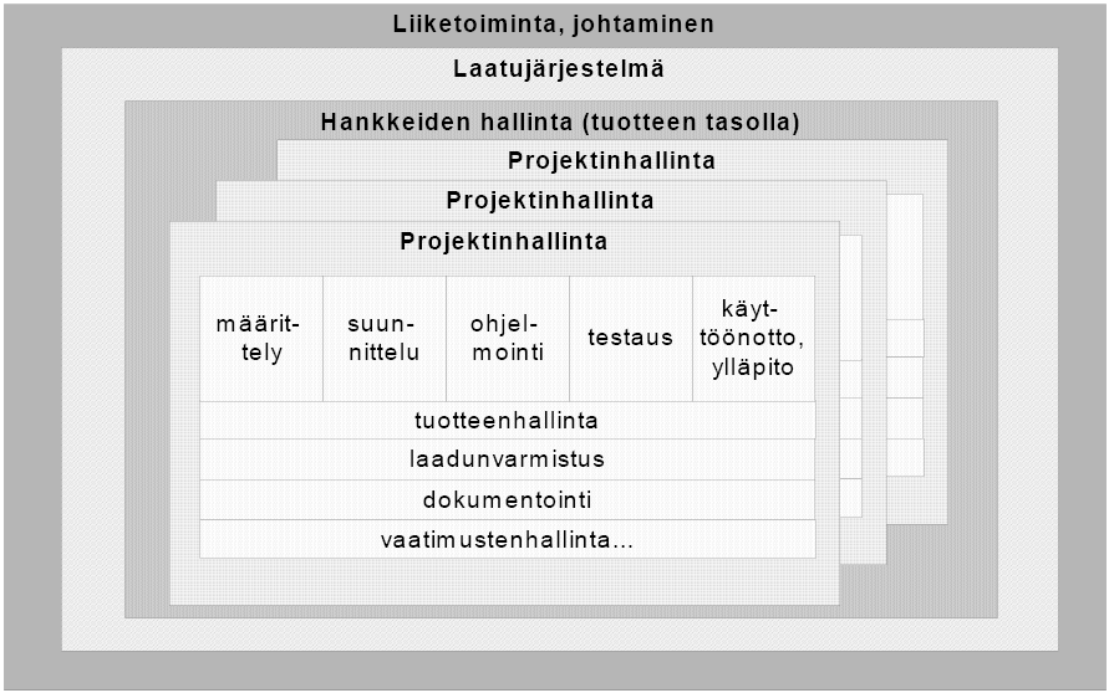
Työ etenee ohjelmistotuotannon olennaisimmista ja tämän työn kannalta tärkeimmistä asioista ohjelmistotestaukseen ja sen automatisointiin. Lopuksi luodaan käytännönläheinen katsaus siihen, minkälaisiin asioihin tulee kiinnittää huomiota, kun on tekemisissä text shell –tason testiautomaation kanssa. Tämän työn yksi tarkoitus onkin perehdyttää aloitteleva testausinsinööri työtehtäviinsä Symbian text shell –tason testauksessa.

# 2 OHJELMISTOTUOTANTO

Tässä luvussa kerrotaan lyhyesti ohjelmistotuotannon osa-alueista, ohjelmiston yleisestä elinkaaresta, laadunvarmistuksesta sekä dokumentoinnista.

Useimmissa ohjelmistoalan yrityksissä on käytössä jonkinlainen laatujärjestelmä, joka määrittelee tuotannon toimintatavat. Yritys voi myös halutessaan sertifioida itsensä omaan liiketoimintamalliin sopivalla standardilla.

Kuvan 2.1 mukaisesti voidaan yleisesti määritellä seuraavat kehitysprosessin vaiheet: määrittely, suunnittelu, ohjelmointi, testaus. Näiden vaiheiden jälkeen tuote otetaan käyttöön ja alkaa ylläpitovaihe. Ylläpidossa kiinnitetään erityisesti huomiota laadunvarmistukseen, jonka tärkeä tukipilari on huolellinen ja määritysten mukainen ohjelmiston sekä siihen mahdollisesti liittyvien muiden komponenttien dokumentointi. Myös muita tukitoimintoja käytetään, ja niissä jokaisessa on yleensä käytössä omat spesifioidut ratkaisumallit.



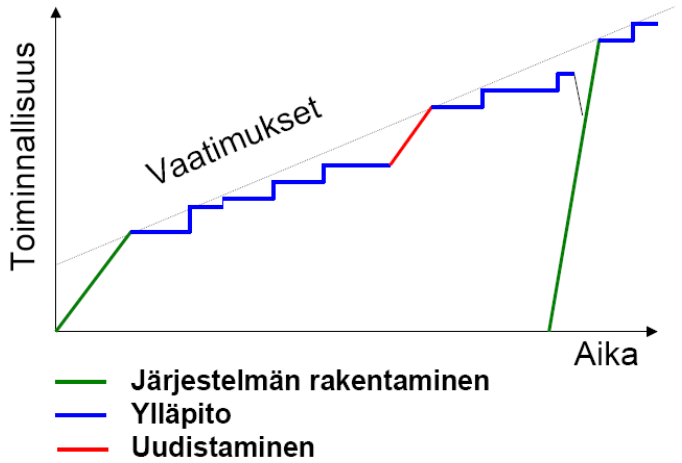

Kuva 2.1: Ohjelmistotuotannon osa-alueet (Haikala ja Märijärvi, 2004)

# 2.1 OHJELMISTON ELINKAARI

## 2.1.1 VAATIMUKSET JA ESITUTKIMUS

Ohjelmiston elinkaarella (life cycle) tarkoitetaan aikaa, joka kuluu ohjelmiston kehittämisen aloittamisesta sen poistamiseen käytöstä (Haikala ja Märijärvi, 2004). Tässä luvussa keskitytään lähinnä ns. vaihejakomalliin, jossa ohjelmiston koko elinkaari jaetaan erillisiin vaiheisiin. Perinteisessä vesiputousmallissa on myös heikkoutensa - kuten se, että jo elinkaaren alkuvaiheessa pitäisi ymmärtää täysin koko ohjelmistotuotannon vaatimukset ja sen tarvitsema aika. Tämän takia on kehitetty myös vaihtoehtoisia elinkaarimalleja kuten inkrementaali- ja evoluutiokehitysmallit, joihin ei tässä yhteydessä kuitenkaan paneuduta tämän tarkemmin.

Elinkaaren alkuvaiheen osioissa keskitytään määrittelytarkastuksiin sekä katselmointeihin. Todellista testausta päästään tekemään vasta toteutusvaiheesta alkaen. Tarkoitus on, että ohjelmistosta pystytään karsimaan virheet mahdollisimman aikaisessa vaiheessa ajan ja rahan säästämiseksi. Jos tarkastuksia ja testausta suoritetaan vasta juuri ennen käyttöönottovaihetta, on virheiden korjaus huomattavasti työläämpää kuin jos sitä olisi tehty koko elinkaaren ajan. Katselmointeja suoritetaan tarpeelliseksi katsottu määrä ohjelmiston koosta riippuen. Yleensä perusteellisempi katselmointitilaisuus järjestetään jokaisen suuremman välivaiheen päätteeksi.

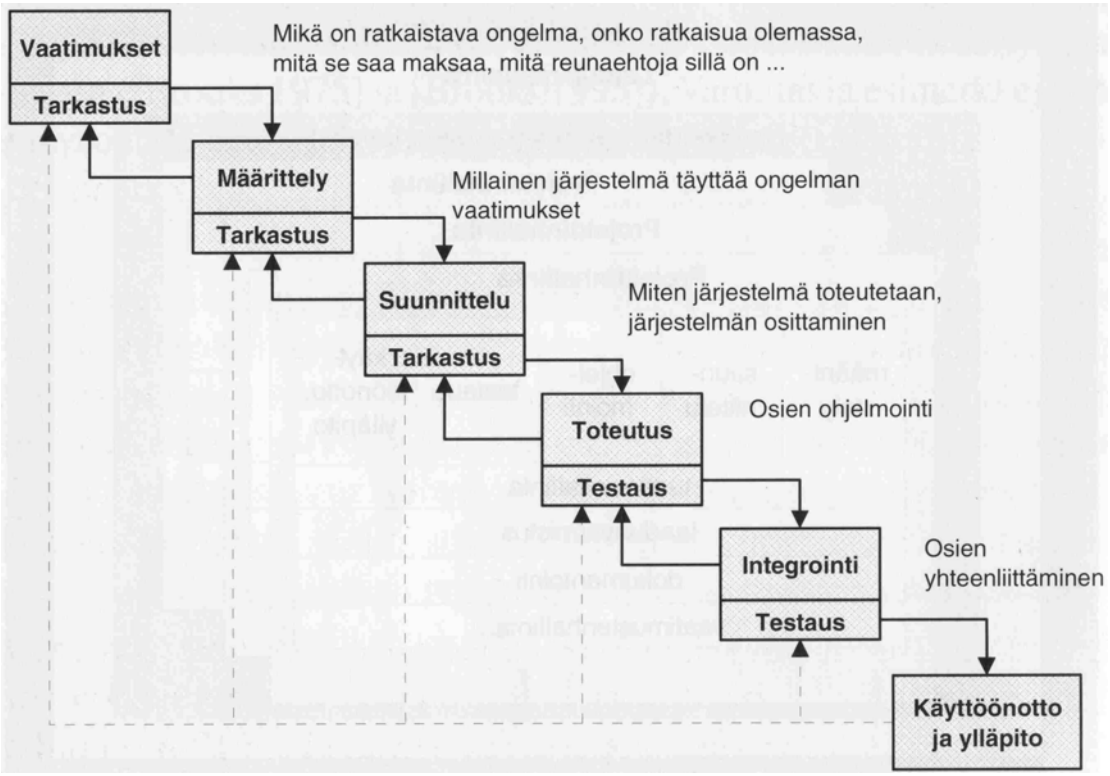



Kuva 2.2: Ohjelmien elinkaari (Harsu, 2006)

Ohjelmiston elinkaaren ehkä tärkein vaihe on esitutkimus. Siinä määritellään ohjelmiston yleiset järjestelmätason vaatimukset sekä asiakkaan todelliset tarpeet. Kartoitukseen kuuluu tarvittavien resurssien arviointi, asiakkaan vaatimusten selvittäminen sekä tulevien vaiheiden yleisempi suunnittelu. Asiakkaan tarpeiden ymmärtämiseen on kiinnitettävä erityistä huomioita, koska ilman sitä on erittäin vaikeaa tehdä sellaista tuotetta, jota asiakas todella haluaa. Toisin sanoen väärin ymmärretyistä asiakasvaatimuksista ei voi päätyä hyvään järjestelmään.

### 2.1.2 VESIPUTOUSMALLIN VAIHEET

Vesiputousmalli on jaettu kuvan 2.2 mukaisiin osiin. Kyseessä on yksi vesiputousmallin toteutus, muitakin versioita on olemassa. Seuraavissa luvuissa on tarkemmin kerrottu vesiputousmallin tärkeimmistä vaiheista, joita ovat määrittelyvaihe, toteutus- eli ohjelmointivaihe, testausvaihe sekä käyttöönotto- ja ylläpitovaihe. Näitä vaiheita voi soveltaa sekä koko vesiputousmalliin että yksittäisiin osa-alueisiin.



Kuva 2.2: Esimerkki vesiputousmallista (Haikala ja Märijärvi, 2004)

**MÄÄRITTELYVAIHE**

Määrittelyvaiheessa kartoitetaan asiakasvaatimukset ja niistä johdetaan ohjelmistovaatimukset, jotka taas määrittelevät toteutettavan järjestelmän. Termillä on monia synonyymeja, kuten järjestelmävaatimukset, toiminnalliset vaatimukset ja toiminnalliset ominaisuudet. Määrittelyn tuloksena syntynyttä dokumenttia sanotaan toiminnalliseksi määrittelyksi (Haikala ja Märijärvi, 2004). Tässä dokumentissa kuvataan ohjelmiston toiminnot, ei-toiminnalliset vaatimukset sekä rajoitukset. Ei-toiminnallisissa (eng. non-functional) määrityksissä keskitytään siihen hiekkalaatikkoon, jossa ohjelmisto tulee toimimaan. Ratkaisevia määreitä ovat mm. suoritusteho, muistimäärä sekä käytetty ohjelmointikieli. Määrittelyvaiheessa muunnetaan asiakasvaatimukset täsmällisiksi ohjelmistovaatimuksiksi.

**OHJELMOINTIVAIHE**

Ohjelmointivaiheessa tuotetaan ohjelman lähdekoodi siltä osin kuin se on määrittelyssä suunniteltu. Se voi käsittää joko yhden funktion tai koko ohjelmiston riippuen siitä, minkälainen vaihe on kyseessä. Tarkoitus on, että esimäärittelyt ja suunnittelu on suoritettu niin huolellisesti, että itse ohjelmointityö olisi mahdollisimman suoraviivaista. Käytännössä kuitenkaan harvoin päästään tähän tilanteeseen, mutta hyvä suunnittelu ja dokumentointi helpottavat ja nopeuttavat tätä vaihetta huomattavasti.

**TESTAUSVAIHE**

Testausvaiheessa etsitään ohjelmistosta virheitä. Testauksessa edetään yleensä ns. V-mallin mukaisesti, vaikka muitakin tapoja on. Testaus on hyvä suunnitella mahdollisimman varhaisessa vaiheessa, mieluummin jo ennen ohjelmointivaihetta. Eri testausvaiheita ovat mm. moduulitestaus, integrointitestaus sekä järjestelmätestaus. Luvussa 3 (Ohjelmistotestaus) paneudutaan ohjelmistotestaukseen syvällisemmin.

## YLLÄPITOVAIHE

Ylläpitovaihe alkaa siitä, kun ohjelmisto on otettu käyttöön suunnitellussa ympäristössä. Ylläpitovaiheessa korjataan ohjelmistossa ilmenneitä ongelmia, lisätään mahdollisia asiakkaan toivomia lisäominaisuuksia ja toimintoja sekä muokataan jo olemassa olevia ominaisuuksia.

Varsinkin pienemmissä ohjelmistoissa ylläpitovaihe painottuu harvemmin julkaistaviin uudempiin ohjelmiston versioihin eikä ns. päivityksiä juurikaan tehdä. Nykyään moniin suurempiin ohjelmistoihin tarjotaan ns. hätäpäivityksiä, joita asiakas voi ottaa käyttöön esimerkiksi internetin välityksellä.

Laajoissa ohjelmistokokonaisuuksissa ylläpitoon kuluu huomattavan paljon resursseja, koska muutos yhdessä moduulissa voi aiheuttaa mittaviakin muutostarpeita muihin moduuleihin. Tämän takia muutosten jälkeen pitäisi aina suorittaa regressiotestaus, jotta voidaan varmistua siitä, että ohjelmisto ei mene miltään osin rikki.

# 2.2 LAADUNVARMISTUS

## 2.2.1 MIKSI LAADULLA ON VÄLIÄ?

Laadunvarmistus on hyvin tärkeä osa jokaista elinkaarivaihetta, ja siihen onkin syytä kiinnittää erityistä huomiota. Laadunvarmistusta saatetaan pitää vain testaamisena, mutta tehokkaassa projektissa sillä tarkoitetaan testausta, teknisiä katselmuksia ja projektisuunnittelua. Kaikilla näillä aktiviteeteilla pyritään siihen, että virheet voitaisiin löytää ja korjata ajoissa ja taloudellisesti (McConnell, 1998).

Laadulla voidaan ymmärtää ohjelmistotuotannossa erilaisia asioita. Joku voi viitata sillä kaatumattomaan järjestelmään kun taas toinen ajattelee sen olevan määriteltyjen vaatimusten täyttämistä. Laadun määritelmää ei silti ole hyvä mennä määrittelemään liian vapaasti itse, vaikka tapauskohtaisiakin ratkaisuja varmasti on olemassa. Hyvin tehty ohjelmisto vastaa määrityksiä sekä toimii halutulla tavalla.

Virheiden hallinnalla on suuri merkitys ohjelmistotuotannossa. Se, miten hyvin ne pysyvät hallinnassa, vaikuttaa ohjelmistoprojektin kehitysnopeuteen, kehityskustannuksiin sekä tulevaan ylläpitoon. Jos virheet korjataan jo elinkaaren alkuvaiheessa, tulee se erään arvion mukaan jopa 50 - 200 kertaa halvemmaksi kuin jos ne korjattaisiin vasta projektin loppuvaiheessa.

## 2.2.2 MITEN LAATU VARMISTETAAN?

Ohjelmiston laadun takaamiseksi on tehtävä tarvittavat toimenpiteet jotta ohjelmistolla on varmasti haluttu laadun taso. Suositeltava tapa on tehdä huolellinen laadunvarmistussuunnitelma. Siinä tulisi kiinnittää huomiota ainakin seuraaviin asioihin:

- Ohjelmiston laadunvarmistusaktiviteetit on suunniteltava.
- Laadunvarmistussuunnitelman on oltava kirjallinen.
- Laadunvarmistusaktiviteetit on aloitettava ohjelmiston määrittelyvaiheessa tai aikaisemmin.
- On perustettava erillinen laadunvarmistusryhmä. Projektin koosta riippuen tuo ”ryhmä” voi olla myös yksi henkilö. Se voi jopa koostua kahdesta suunnittelijasta, jotka kumpikin työskentelevät erillisissä projekteissaan ja hoitavat toistensa töiden laadunvarmistuksen.
- Laadunvarmistusryhmän jäsenet on koulutettava laadunvarmistusaktiviteettien hoitamiseen.
- Ohjelmiston laadunvarmistustyölle on osoitettava riittävästi resursseja.

Laadun todentamisessa voidaan käyttää myös arvioinnin kohteesta riippumattoman tahon suorittamaa yrityksen tai projektin laatujärjestelmän arviointia. Arviointi perustuu yleensä laadunvarmistussuunnitelmaan tai standardiin. ISO 9001 –standardia käydään seuraavassa aliluvussa läpi hieman tarkemmin.

Katselmoinnit ovat olennainen osa laadunvarmistusta. Teknisen katselmoinnin tarkoituksena on löytää ja korjata virheitä elinkaaren alkuvaiheen osa-alueista. Yleistermillä ”tekninen katselmointi” (technical review) voidaan viitata mihin tahansa useista katselmointitekniikoista, kuten läpikäymiseen (walkthrough), tarkastusmenettelyyn (inspection) ja koodin lukemiseen (code reading).

# 2.3 DOKUMENTOINTI

## 2.3.1 DOKUMENTOINNIN MERKITYS

Ohjelmistotyö on suurelta osin dokumenttien tuottamista. On suotavaa, että kaikki ohjelmiston elinkaaren vaiheet dokumentoidaan huolellisesti. Tämä jääkin monesti liian vähälle huomiolle, kun tulee kiire aikataulussa pysymisen takia. Usein ohjelmiston yksittäiset komponentitkin ovat niin monimutkaisia, että kun niitä jälkikäteen ylläpidetään, on tutustuminen komponentin toimintaan ilman riittävää dokumentointia hyvinkin aikaa vievää työtä. Voidaankin todeta, että vaikka ohjelmointivaiheessa dokumentointi saattaa tuntua ajanhaaskaukselta, siinä loppujen lopuksi säästetään aikaa, kun tarkastellaan ohjelmiston koko elinkaarta.

## 2.3.2 DOKUMENTTITYYPIT

Dokumentit on hyvä kategorisoida yrityksen sisällä tarpeen mukaan. Yleishyödyllisiä suomenkielisiä dokumentointimalleja on saatavilla mm. HYTT-laatujärjestelmästä. Ohjelmiston kehityksessä tarvittavat dokumentit voidaan jakaa seuraaviin ryhmiin: laatukäsikirjaan liittyvät dokumentit, projektinhallintaan liittyvät dokumentit sekä tuotedokumentit.

Laatujärjestelmän dokumentteja:

- laatukäsikirja
- ohjeistukset
- dokumenttimallit
- auditointiraportit
- kokouspöytäkirjat

Projektidokumentteja:

- sopimukset
- projektisuunnitelma
- projektin seurantaraportit
- loppuraportti

Tuotedokumentteja:

- projektikohtaiset tuotedokumentit
- tuotekohtaiset tuotedokumentit

### 2.3.3 DOKUMENTOINTIMALLIT

Dokumentoinnissa kannattaa noudattaa yhdenmukaisuutta, jolloin luettavuus ja halutun tiedon etsiminen helpottuvat. Dokumenteissa onkin hyvä ottaa käyttöön yhteinen dokumentointimalli, jossa mm. sellaiset kohdat kuin ulkoasu ovat yhteneviä. Tietenkään sopimuspohja ei ole sisällöltään samankaltainen kuin esim. testausraportti, mutta dokumentaatio on hyvä tunnistaa jo ulkonäön perusteella oman yrityksen tuotokseksi.

Dokumentaatiossa on yleensä ainakin kansilehti, josta löytyvät tiedot dokumentin laatijasta sekä yrityksestä, joka dokumentin on tuottanut. Dokumentin nimi, versionumero, sivujen lukumäärä sekä dokumentin tila (kesken/valmis) täytyy olla merkittynä.

Dokumentaatiomallit soveltamisohjeineen liitetään osaksi laatujärjestelmän dokumentaatiota, jolloin niiden käyttöönotto on jouhevampaa.

Myös lähdekooditiedostot ovat dokumentointia. Lähdekoodin tyylin yhdenmukaistamiseksi on olemassa erilaisia tyylioppaita, joko yrityksen sisäisiä tai hieman virallisempia kuten Ellemtelin C++-tyyliopas (Ellemtel, 1992).

# 3 OHJELMISTOTESTAUS

Ohjelmistotestaus on hyvä mieltää suunnitelmalliseksi virheiden etsimiseksi. Moni edelleen ajattelee, että testaus on vain pakollinen ja ei niin välttämätön osa ohjelmiston elinkaarta. On kuitenkin osoitettu, että testaus on varmin tapa löytää ohjelmistosta virheitä ja että ylläpitokustannuksetkin pysyvät pienempinä, kun ohjelmisto on testattu. Testauksen tärkeyden huomaa vaikka siitä, että ohjelmistovirheiden vuosittaiset kustannukset USA:ssa ovat arviolta n. 0,6 % BKT:sta, eli n. $60 miljardia (The Economic Impacts of Inadequate Infrastructure for Software Testing, 2002).

Testauksen tarkoituksena ei siis missään nimessä ole osoittaa, että ohjelmisto toimii oikein, vaan että siitä löytyy virheitä (Myers, 2004). Laadukkaan testauksen suunnittelussa täytyykin ottaa huomioon nimenomaan sellaiset tilanteet, jotka ovat harvinaisia ja joita varten ohjelmaa ei välttämättä ole suunniteltu. Testitapausten suunnittelussa hyvä laatu korvaa tapausten määrän kirjaimellisesti. On siis pyrittävä testaamaan kaikki mahdolliset tilanteet, joihin ohjelmiston käytössä voidaan joutua. Osittain tästä johtuen testausinsinöörinä olisi hyvä toimia joku muu kuin ohjelmiston kehittäjä, jotta testaukseen saataisiin uutta näkökulmaa.

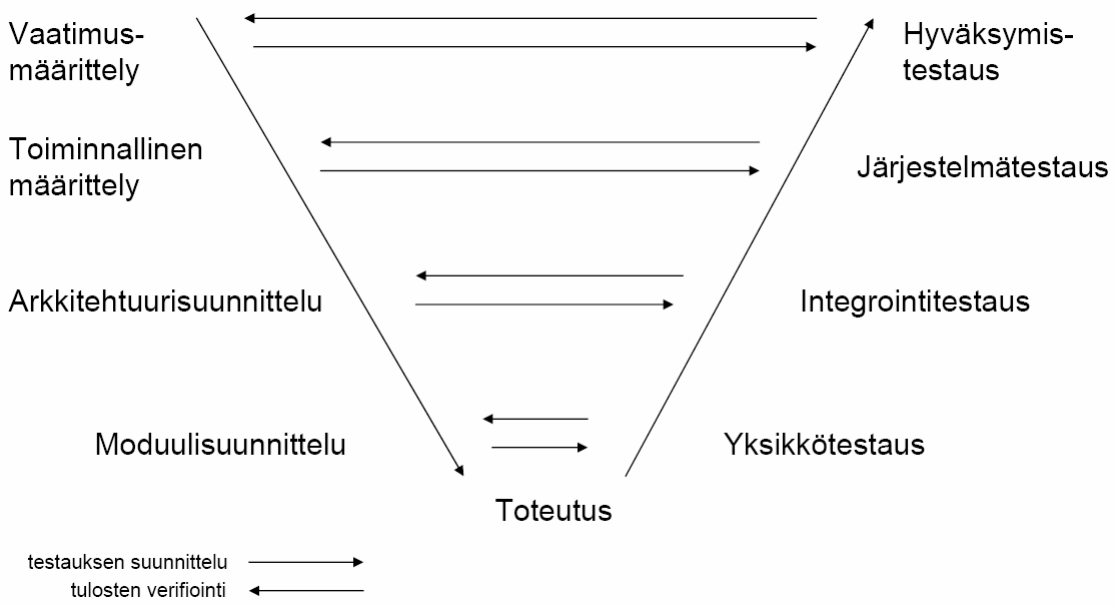
Tässä luvussa tullaan esittelemään testaustasot ns. V-mallin näkökulmasta. Muitakin testausmalleja on olemassa, mutta V-malli on niistä yleisin.

Testitapausten valinnasta kerrotaan alakohdassa 3.2, ”Testitapausten määrittely”. Luvussa syvennytään mm. mustalaatikko- sekä lasilaatikkotestauksen määritelmiin.

Lopuksi vielä käsitellään testitapausten riittävyyden arviointia. Aihetta lähestytään erilaisten kriteerien kautta ja keskitytään siihen, milloin ohjelmistoa on testattu riittävästi tai niin paljon kuin se on järkevää.

## 3.1 TESTAUSTASOT

Ohjelmistoa voidaan testata muun muassa ns. V-mallin mukaisesti, vaikka muitakin tapoja on olemassa. V-mallissa testaus voidaan aloittaa samaan aikaan kuin vastaava tuotantovaihekin, kuten kuvasta 3.1 voi päätellä. Hyväksymistestaus suunnitellaan vaatimusmäärittelyvaiheessa, järjestelmätestaus määrittelyvaiheessa, integrointitestaus arkkitehtuuri-suunnitteluvaiheessa ja yksikkötestaus moduulisuunnitteluvaiheessa.




Kuva 3.1: Testauksen V-malli (Katara, 2006).

V-mallin mukaisia testaustasoja ovat yksikkötestaus (module testing, unit testing), integrointitestaus (integration testing) ja järjestelmätestaus (system testing).

Yllä mainittujen testaustasojen lisäksi tutustutaan lyhyesti myös käytettävyystestaukseen.

Testaukseen on olemassa monia muitakin lähestymistapoja, mutta tässä yhteydessä niihin ei kuitenkaan syvemmin perehdytä. Tärkeimmät ovat ehkä alfa-testaus, jolla tarkoitetaan yleensä asiakkaan suorittamaa testausta toimittajan tiloissa sekä beta-testaus, jolla tarkoitetaan asiakkaan omissa tiloissaan itsenäisesti suorittamaa testausta.

### 3.1.1 YKSIKKÖTESTAUS

Yksikkötestauksessa (moduulitestauksessa) testataan yksittäistä moduulia tai komponenttia. Tämä tarkoittaa käytännössä mitä tahansa testattavissa olevaa ohjelman yksikköä: funktiota, algoritmia ja niin edelleen. Moduulitestauksen tuloksia verrataan moduulisuunnitteluun, arkkitehtuurisuunnittelun tuloksiin sekä tekniseen määrittelydokumenttiin. Testattava moduuli saattaa tarvita erityisen testiajurin (test driver) ja tynkämoduulin (test stub). Ajurin tehtävänä on välittää testitapaukset testattavalle moduulille sekä esittää testitulokset. Tynkämoduulit taas edustavat muita moduuleita, joita testattava moduuli kutsuisi (Myers, 2004). Yksikkötestauksessa testaajana voi toimia joko ohjelmoija itse tai esimerkiksi toinen ohjelmoija (paritestaus).

Moduulitestauksella on kiistattomat etunsa. Se helpottaa virheen korjausta ja paikantamista, koska testattava yksikkö on selkeästi rajattu pieneen kokonaisuuteen. Myös itse testaus saadaan tehokkaammaksi, koska sitä pystytään tekemään rinnakkain riippumatta toisista moduuleista.

Reaaliaikajärjestelmiä testattaessa pitäisi pyrkiä aina mahdollisuuksien mukaan suorittamaan testaus todellisessa ympäristössä, koska esim. erilaiset emulaattorit ja debug-ympäristöt aiheuttavat poikkeamia mm. ajastuksissa.

### 3.1.2 INTEGROINTITESTAUS

Integrointitestauksessa pyritään selvittämään, toimivatko ohjelmiston moduulit toistensa kanssa moitteettomasti, sujuuko tiedonsiirto virheettömästi ja yhdistävätkö rajapinnat moduulit oikein. Vaikka yksittäiset moduulit toimisivatkin oikein, se ei tarkoita sitä, että moduulit toimisivat yhdessä oikein. Alimmalla tasolla integrointitestausta tekee normaalisti ohjelmistokehitysryhmä, että voitaisiin taata moduulien toimivuus yhdessä. Korkeammalla tasolla integrointitestausta ja käännösten testausta tekevät testausryhmät (Craig ja Jaskiel, 2002). Integrointitestaus etenee usein rinnan moduulitestauksen kanssa, ja sitä onkin usein tarpeetonta tarkastella erillään moduulitestauksesta (Haikala ja Märijärvi, 2004).

Integrointi voidaan suorittaa joko ei-inkrementaalisesti (big bang), jolloin kaikki moduulit yhdistetään samanaikaisesti yhdeksi ohjelmaksi, tai lisäämällä moduuleja vähitellen yhdistettyyn kokonaisuuteen. Jälkimmäiseen tapaan on olemassa kaksi eri suoritustapaa: top-down -menetelmä ja
bottom-up -menetelmä. Kysymys testaustavan valinnassa on, tulisiko moduulit testata ensin itsenäisesti ja sitten yhdistää toimivaksi ohjelmistoksi, vai tulisiko testattava moduuli ennen testausta yhdistää aiemmin testatun moduulin kanssa (Myers, 2004).

### 3.1.3 JÄRJESTELMÄTESTAUS

Järjestelmätestauksella (system testing) pyritään takaamaan, että tuote kokonaisuudessaan täyttää sille asetetut tavoitteet. Tuloksia verrataan ohjelmiston toiminnalliseen määrittelyyn sekä asiakasdokumentaatioon. Tämä tarkoittaa sitä, että ilman asianmukaisia mitattavia tavoitteita järjestelmätestausta on mahdotonta suorittaa (Myers, 2004).

Järjestelmätestausvaiheessa on siis käytettävissä koko järjestelmä, jolloin voidaan testata myös määrittelydokumenttien ulkopuolella olevia asioita, kuten toimintavarmuutta kenttäolosuhteissa. Toipuminen virhetilanteista, skaalautuvuus ja resurssien käyttö ovat asioita, joita on myös hyvä testata, vaikkei niitä määrittelydokumentaatiossa olisikaan mainittu (Katara, 2006).

Mitä varhaisemmassa vaiheessa V-mallin mukaista elinkaarta virheet löydetään, sen halvemmaksi niiden korjaus tulee. Erittäin todennäköisesti yhden virheen korjaaminen aiheuttaa uusia virheitä tai ainakin yhteensopivuusongelmia jo onnistuneesti integroiduissa moduuleissa. Tämän vuoksi virheen löydyttyä järjestelmätestausvaiheessa pitääkin integrointitestaus suorittaa virheen korjauksen jälkeen uudelleen. Myös järjestelmätestaus täytyy suorittaa uudelleen. Tämänkaltaista uudelleentestausta (testaustasosta huolimatta) kutsutaan regressiotestaukseksi, ja sen suorittaminen voi tulla erittäin kalliiksi, ellei testausta saada automatisoitua (Haikala ja Märijärvi, 2004).

### 3.1.4 HYVÄKSYMISTESTAUS

Hyväksymistestaus vastaa V-mallin mukaista testausvaihetta vaatimusmäärittelylle. Hyväksymistestit suunnitellaan siis ensimmäisenä ja toteutetaan viimeisenä. Tässä vaiheessa todennetaan, onko lopullinen tuote sopimuksen mukainen ja että vaatimukset on täytetty. Kyseessä on kokonainen, valmis tuote ja testauksen tulisi olla laajoissakin ohjelmistoissa kestoltaan lyhyt, enimmillään viikkoja. Ensisijainen tarkoitus ei enää ole löytää virheitä. Jos niitä kuitenkin löytyy, korjaus tulee erittäin kalliiksi, koska ohjelmointi- ja testausvaiheita joudutaan läpikäymään mahdollisesti jopa elinkaaren alkuvaiheista asti.

Testaajina olisi hyvä käyttää tuotteen todellisia loppukäyttäjiä ja testausympäristönä mahdollisimman todellista käyttöympäristöä. Vaikka hyväksymistestaus onkin V-mallin mukaisesti vasta viimeinen testausvaihe, on tuotteesta hyvä saattaa jokin väliversio hyväksymistestaajille tutustumista varten. Mikäli käyttäjät pääsevät tutustumaan tuotteeseen vasta kun se on valmis, tullaan virheitä löytämään erittäin suurella todennäköisyydellä.

### 3.1.5 KÄYTETTÄVYYSTESTAUS

Käytettävyystestaus voidaan ajatella osaksi järjestelmätestausta, vaikka se onkin käytännössä oma osa-alueensa (Myers, 2004). Käytettävyystestauksessa selvitetään, kuinka loppukäyttäjä suoriutuu niistä toimenpiteistä, joita tuotteella on tarkoitus tehdä. Yleensä testaus suoritetaan vain tuotteen käyttöliittymän avulla, ja testausta tehdään usein jo määrittelyvaiheessa käyttöliittymäprototyypin avulla. Testaus voidaan järjestää erityisessä käytettävyystestilaboratoriossa, jossa käyttäjän toimenpiteet usein videoidaan. Käyttäjää kehotetaan puhumaan ajatuksensa tuotteen käytöstä ääneen, jotta tuloksia voidaan jälkikäteen helpommin analysoida (Haikala ja Märijärvi, 2004)

Siinä vaiheessa kun kaikki tekniset ongelmat on ratkaistu, keskitytään siihen miten loppukäyttäjä suoriutuu haluamastaan toimenpiteestä. Alla on lista asioista, jotka muun muassa täytyy ottaa huomioon testejä (ja yleensäkin käyttöliittymää) suunniteltaessa. Lista on kooste Glenford J. Myersin ohjelmistotestauksen perusteoksesta *”The Art of Software Testing”* .

1. Onko käyttöliittymä suunniteltu vastaamaan loppukäyttäjän älykkyyttä, koulutustaustaa sekä ympäristötekijöitä?

2. Ovatko ohjelmiston ulostulot tarkoituksenmukaisia ja selkokielisiä eivätkä syyllistä käyttäjää ”hölmöstä toimenpiteestä”?

3. Virheilmoitusten täytyy olla selkeitä. Jos ilmoitus on tyyliä ”IEK022S OPEN ERROR ON FILE ’SYSIN’ ABEND CODE=102”, harvempi käyttäjä ymmärtää ilmoituksesta yhtään mitään. Täytyykin siis kiinnittää huomiota selkeisiin ilmoituksiin, joita käyttäjä ymmärtää.

4. Tarvitaanko käyttäjän tunnistamisessa (esim. pankkipalvelut vs. kirjastolainojen tarkastelu) minkälaista tarkkuutta? Pitääkö käyttäjältä kysyä nimen lisäksi myös PIN-koodi, tilinumero, erillinen kirjautumistunnus jne.?

5. Onko käyttöliittymässä oikea määrä valikkokomentoja? Nykyisissä järjestelmissä suositaan monesti tapaa, joka käyttäjän toimenpiteiden perusteella pitää useimmin käytetyt valikkovaihtoehdot näkyvillä. Tietenkin kaikkien toimintojen käyttäminen täytyy aina olla mahdollista riittävän helposti.

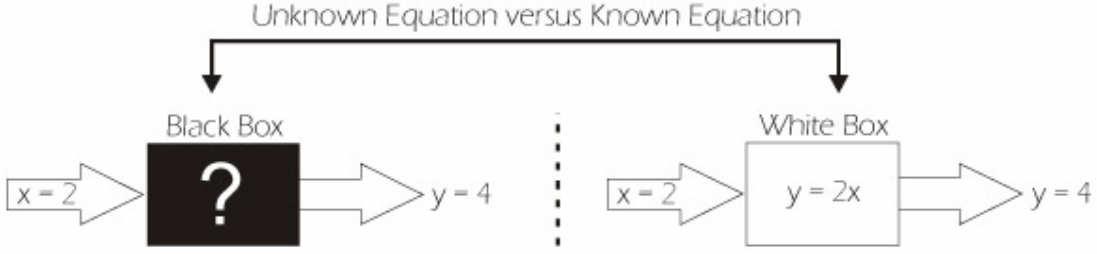
6. Onko ohjelmistoa ylipäätään helppo käyttää? Jos käyttäjä haluaa suorittaa jonkin hieman harvinaisemman toimenpiteen, navigoinnin haluttuun komentoon ei pitäisi tapahtua liian monen valikkolistan kautta. Myös paluun päävalikkoon täytyy olla mahdollisimman selkeää.

## 3.2 TESTITAPAUSTEN MÄÄRITTELY

Testitapausten määrittely- ja valintavaiheessa käytetään tavallisesti kahta lähestymistapaa: rakenteellista testausta (lasilaatikkotestaus, glass/white box testing, structural testing) ja toiminnallista testausta (mustalaatikkotestaus, black box testing, behavioral testing, functional testing). Rakenteellisessa testauksessa ohjelman sisältö on tiedossa ja testitapaukset suunnitellaan tämän tiedon perusteella. Toiminnallisessa testauksessa ohjelman sisältöä kooditasolla ei tunneta ja ainoastaan rajapinnat ovat käytettävissä testitapausten suunnittelun apuna. Testitapaukset määritellään syötteiden ja tulostusten perusteella.

Eri lähestymistavoilla on omat etunsa eri tuotantovaiheissa. V-mallin alimmalla tasolla testauksen luonne on rakenteellista, ja mitä ylemmäs mallissa edetään sitä enemmän luonne muuttuu toiminnallisen testauksen mukaiseksi. Esimerkiksi yksikkötestauksessa käytetään yleensä rakenteellista testausta, koska ohjelman sisältö on tunnettu, ja sen perusteella on helppo muodostaa yksikkötestauksen luonteen mukaisia testejä. Eri lähestymistavat eivät ole toisiaan poissulkevia. Testitapausten suunnittelussa tulisikin käyttää kumpaakin lähestymistapaa mahdollisimman laajan testikattavuuden saavuttamiseksi.

Toiminnalliset testitapaukset tulisi suunnitella ennen rakenteellisia testitapauksia, koska ne voidaan luoda vaatimusmäärittelyn avulla kauan ennen kuin itse koodia on toteutettu. Tällainen menettely edesauttaa hyvää koodisuunnittelua ja -toteutusta eikä kattavia rakenteellisia testejä voi edes toteuttaa ilman valmista testikoodia (Craig ja Jaskiel, 2002).




Kuva 3.2: Rakenteellisen ja toiminnallisen testauksen erot (Craig ja Jaskiel, 2002)

### 3.2.1 RAKENTEELLINEN TESTAUS

Rakenteellisessa testauksessa testitapaukset suunnitellaan niin, että testattava ohjelmakoodi on testitapauksen suunnittelijan tiedossa. Tarkoitus on siis testata mahdollisimman kattavasti halutun kokonaisuuden ohjelmakoodi. Tässä täytyy ottaa huomioon koodin kaikki mahdolliset ajonaikaiset etenemisvariaatiot, joita mm. ehtorakenteet aiheuttavat. Jos siis erilaisia ajonaikaisia koodinsuoritusmahdollisuuksia on 4 kappaletta, tarvitaan vähintään yhtä monta testitapausta, jotta riittävä kattavuus saavutetaan (Craig ja Jaskiel, 2002).

Rakenteellisen testauksen suunnittelussa käytetään myös erilaisia metodeja tarvittavan metriikan tuottamiseksi. Polkutestaus (path testing) ja kattavuustestaus (coverage testing) kuvaavat ohjelmakoodin suunnattuna graafina ja ne on johdettu matemaattisesta graafiteoriasta.

Erilaisia kattavuudenmäärittelytekniikoita käytetään myös laajalti testisuunnittelussa. Lausekattavuus (statement coverage) kertoo, kuinka suuri osa ohjelmakoodista käydään läpi testin aikana. Päätöskattavuus (decision or branch coverage) puolestaan käy läpi, kuinka paljon ehtorakenteita koodi sisältää. Polkukattavuus (path coverage) kertoo, kuinka monta erilaista ajonaikaista etenemisvaihtoehtoa testattava ohjelmakoodi sisältää (Jorgensen, 2002).

### 3.2.2 TOIMINNALLINEN TESTAUS

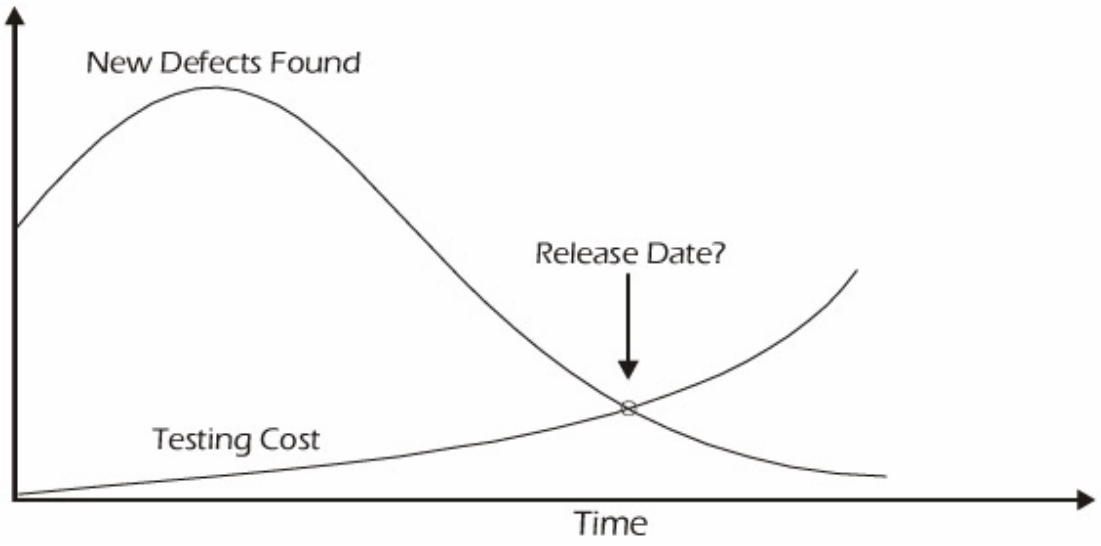
Toiminnallisessa testauksessa testattavaa ohjelmistoa käsitellään mustana laatikkona, jonka sisältöä ei tunneta. Ohjelmistolle annetaan erilaisia syötteitä ja tulosten analysointi tapahtuu ohjelman antamien tulosten perusteella. Myös toiminnallisella testauksella on etunsa ja haittansa. Koska toiminnallinen testaus perustuu määrittelydokumentaatioon, voi itse ohjelmakoodin toteutus muuttua sen vaikuttamatta testitapauksiin, kunhan ulkoisten rajapintojen käytös pysyy samana. Toisaalta määrittelemätön toiminnallisuus jää testaamatta, jos käytetään ainoastaan toiminnallista testausta, koska rajapintakaan ei ole silloin tiedossa (Jorgensen, 2002).

Ekvivalenssiositus on yleinen metodi testitapausten suunnittelussa. Siinä testitapausten määrittely perustuu syöteavaruuden jakamiseen ekvivalenssiluokkiin. Ekvivalenssiosituksessa voidaan olettaa, että yhdellä ekvivalenssiluokan edustajalla toimiva toteutus toimii myös muilla saman luokan edustajilla. Esimerkkinä voi käyttää ohjelmaa, joka laskee kahden kokonaisluvun summan onnistuneesti. Tällöin voidaan olettaa, että ohjelma suoriutuu myös muilla arvoilla onnistuneesti tehtävästään. Tällaisessa tapauksessa täytyy testata myös käytetyn kokonaislukualueen raja-arvot. Tämänkaltaista testausta kutsutaan raja-arvoanalyysiksi (Haikala ja Märijärvi, 2004).

## 3.3 TESTAUKSEN RIITTÄVYYDEN ARVIONTI

Testauksen ehkäpä haastavin vaihe on sen riittävyyden arviointi. Miten määritellä, koska ohjelmistoa on testattu tarpeeksi? Hyvin yleinen kriteeri testauksen lopettamiselle on testaussuunnitelmassa määritellyt hyväksymiskriteerit. Tätä suunnitelmaa voi tosin olla hankala noudattaa, varsinkin jos ohjelmiston luovutusajankohta on jo sovittu asiakkaan kanssa. Tarvittavaa testauksen määrää voi arvioida ns. mutkikkuusmitoilla ja testauksen riittävyyttä puolestaan ns. kattavuusmitoilla sekä virheitä kylvämällä. Mutkikkuus- eli kompleksisuusmittojen (complexity measure) avulla analysoidaan koodin monimutkaisuutta ja sen kautta paljon testausta vaativat osat. Kattavuusmitoilla arvioidaan sitä, onko ohjelmistoa testattu riittävän kattavasti (Haikala ja Märijärvi, 2004). Erilaisia kattavuusmittareita käsiteltiin rakenteellisen testauksen yhteydessä (alakohta 3.3.2).

Kuten alla olevasta kuvasta voidaan päätellä, virheiden löytymisaste laskee selvästi mitä enemmän ohjelmistoa on testattu. Toisaalta löydettyjen virheiden yksikköhinta nousee selvästi.

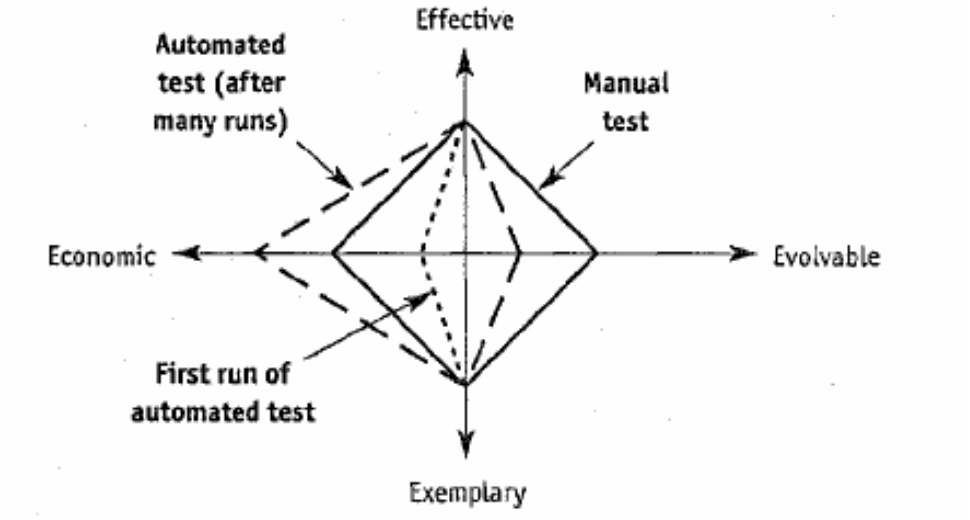



Kuva 3.3: Virheiden löytymisaste (Craig ja Jaskiel, 2002)

# 4 OHJELMISTOTESTAUKSEN AUTOMATISOINTI

Ohjelmistotestauksen automatisointi tulee ennemmin tai myöhemmin ajankohtaiseksi hiemankaan laajemman ohjelmistoprojektin kuluessa. Yksittäisenkin testitapauksen voi automatisoida ja joissain tapauksissa se voi olla jopa järkevää. Useimmiten testausta automatisoidaan silloin, kun kyse on suuremmista kokonaisuuksista tai paljon aikaa ja resursseja vievästä testauksesta kuten regressiotestauksesta. (Haikala ja Märijärvi, 2004).

Kuva 4.1 havainnollistaa testitapauksen neljää ominaisuutta Keviat-kaaviossa. Manuaalisesti ajetut testitapaukset esitetään paksulla mustalla viivalla ja automaattiset katkoviivoin. Kun sama testi ajetaan automatisoidusti ensimmäistä kertaa, se on taloudellisesti huomattavasti kalliimpaa (koska automatisointi vie resursseja). Kun automatisoitua testiä ajetaan toistuvasti, se muuttuu myös taloudellisemmaksi kuin jos sama määrä testitapauksia ajettaisiin manuaalisesti (Fewster ja Graham, 1999).



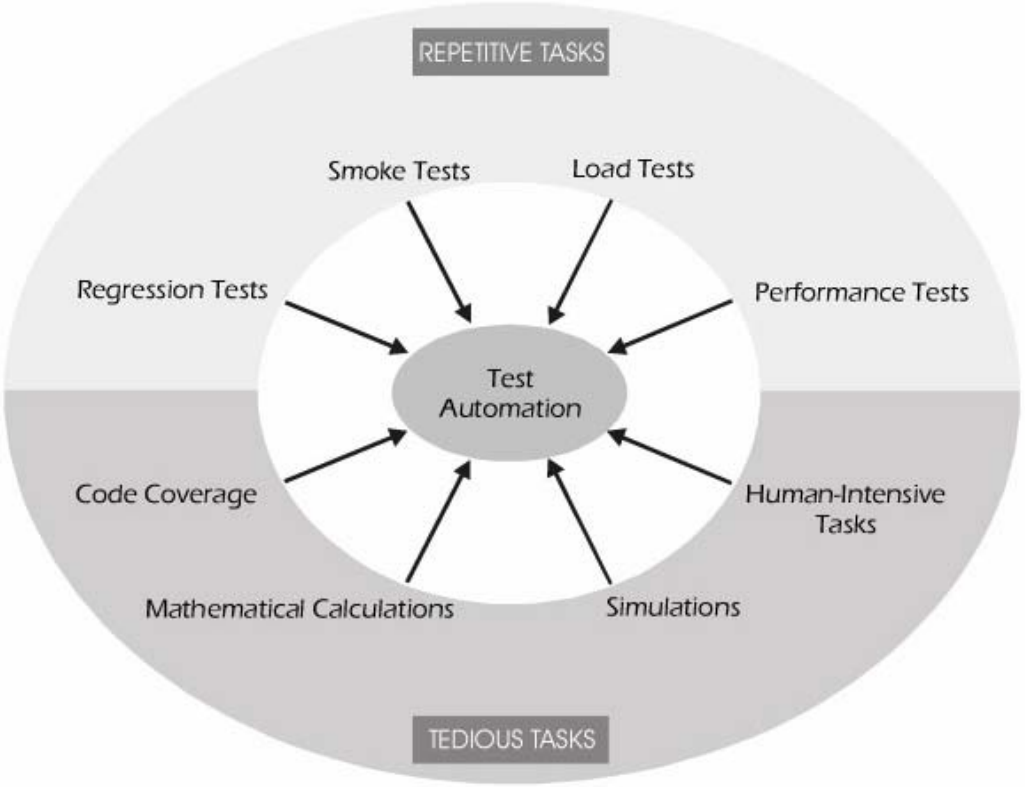

Kuva 4.1: Keviat-kaavio, joka havainnollistaa testitapauksen ominaisuuksia (Fewster ja Graham, 1999).

Tässä luvussa keskitytään vielä siihen, minkälaisissa tapauksissa testejä kannattaa automatisoida sekä tarkastellaan lyhyesti eri testausmalleja joihin automatisointi sopii. Lopuksi paneudutaan muutamaan asiaan, joiden kanssa kannattaa automatisoinnin suunnittelussa olla tarkkana.

## 4.1 MITÄ KANNATTAA AUTOMATISOIDA?

Kaikkia ohjelmistotestejä ei kannata automatisoida. Testien automatisointi vie usein huomattavasti enemmän aikaa ja osaamista kuin manuaalisten testien kehittäminen. Eräs tapa on luoda ensin kaikki testit manuaalisesti ajettaviksi ja sen jälkeen automatisoida ne testit, joita suoritetaan hyvin monta kertaa peräkkäin. Joissakin organisaatioissa automatisoinnin hoitavat kokonaan eri tahot kuin alkuperäisten testien kehittäjät. Jos automatisoitujen testien kehittäminen vie enemmän aikaa kuin manuaalisten testien kehittäminen, täytyy selvittää, kuinka paljon aikaa säästyy automatisoiduissa ajoissa. Sen jälkeen arvioidaan, kuinka monta kertaa automatisoituja testejä täytyy ajaa, jotta se on ajankäytön kannalta hyödyllistä. Tämän säännön pohjalta voi päättää, mitkä testit kannattaa automatisoida (Craig ja Jaskiel, 2002).

Joitain testejä ei edes pystytä automatisoimaan. Esimerkiksi tietoliikennekaapelin irrottaminen testilaitteesta kesken tiedonsiirron voi tietyissä tapauksissa ohjelmallisesti toteutettuna antaa täysin eri testitulokset kuin että se irrotetaan ihmisen toimesta fyysisesti.




Kuva 4.2: Toistuvat (repetitive) ja pitkästyttävät (tedious) testit ovat pääehdokkaita testien automatisoinnissa (Craig ja Jaskiel, 2002).

### 4.1.1 REGRESSIOTESTAUS

Regressiotestauksen (regression testing) tulisi olla hyvin pitkälle automatisoitua. Regressiotestauksen avulla on tarkoitus selvittää, että muunneltu ohjelma lisäyksen tai muutoksen jälkeen vastaa yhä määrityksiään ja ettei uusia virheitä ole päässyt ohjelmaan. Vaikka regressiotestausta käytetään yleisimmin ohjelmiston ylläpidon aikana, eivät muunneltavat ohjelmat ole ainoa regressiotestauksen kohde. Regressiotestausta voidaan käyttää myös kehitystyössä varmistamassa, etteivät uusien alijärjestelmien integroimiset ole aiheuttaneet tahattomia sivuvaikutuksia (Pressman, 2000).

### 4.1.2 SAVUTESTAUS

Savutestauksessa (smoke testing) pyritään todentamaan ohjelmiston vakautta. Tarkoitus ei ole niinkään löytää virheitä, vaan osoittaa testausryhmälle ohjelmiston sen hetkinen tila. Savutestaus on luonteeltaan hyvin samankaltaista kuin regressiotestaus ja soveltuu erittäin hyvin automatisoitavaksi. (Craig ja Jaskiel, 2002).

### 4.1.3 KUORMITUSTESTAUS

Kuormitustestauksen (load testing) tarkoituksena on kuormittaa ohjelmistoa maksimikapasiteetilla riittävän kauan tarpeellisen toimintavarmuuden todentamiseksi. Kuormitustestausta on usein vaikeaa suorittaa täysin samalla tavalla montaa kertaa peräkkäin manuaalisesti, joten myös tämäntyyppinen testaus soveltuu hyvin automatisoitavaksi.

### 4.1.4 SUORITUSKYKYTESTAUS

Suorituskykytestaus (performance testing) on luonteeltaan samantyyppistä kuin kuormitustestaus. Se on pitkäkestoista ja ohjelmistoa pyritään kuormittamaan paljon. Suorituskykytestejä ei yleensä ajeta kovin usein, mutta luonteensa takia ne eivät sovi manuaalisesti ajettaviksi.

## 4.2 AUTOMATISOINNIN SUDENKUOPAT

Ohjelmistotestauksen automatisointi ei itsessään ole kaikenkorjaava ratkaisu, jolla virheet saadaan eliminoitua ja jolla manuaalinen testaus tehtäisiin tarpeettomaksi. Automatisoinnista on hyötyä vain, jos sillä saavutetaan etua manuaaliseen testaukseen verrattuna. Automatisoinnilla pyritään siis testauksen ja ohjelmiston laadun parantamiseen ja resurssien tarpeen vähentämiseen. Seuraavissa kohdissa käydään läpi joitain asioita, joita täytyy huomioida automatisoinnin suunnittelussa.

### 4.2.1 RIITTÄMÄTÖN SUUNNITTELU

Automatisoinnin tarve täytyy kartoittaa erittäin huolellisesti ennen kuin sitä ryhdytään toteuttamaan. Automatisointia olisi hyvä suunnitella jo ohjelmiston määrittelyvaiheessa, jolloin todelliset tarpeet ja niitä vastaavat ratkaisut olisi helppo löytää.

### 4.2.2 LIIAN SUURET ODOTUKSET

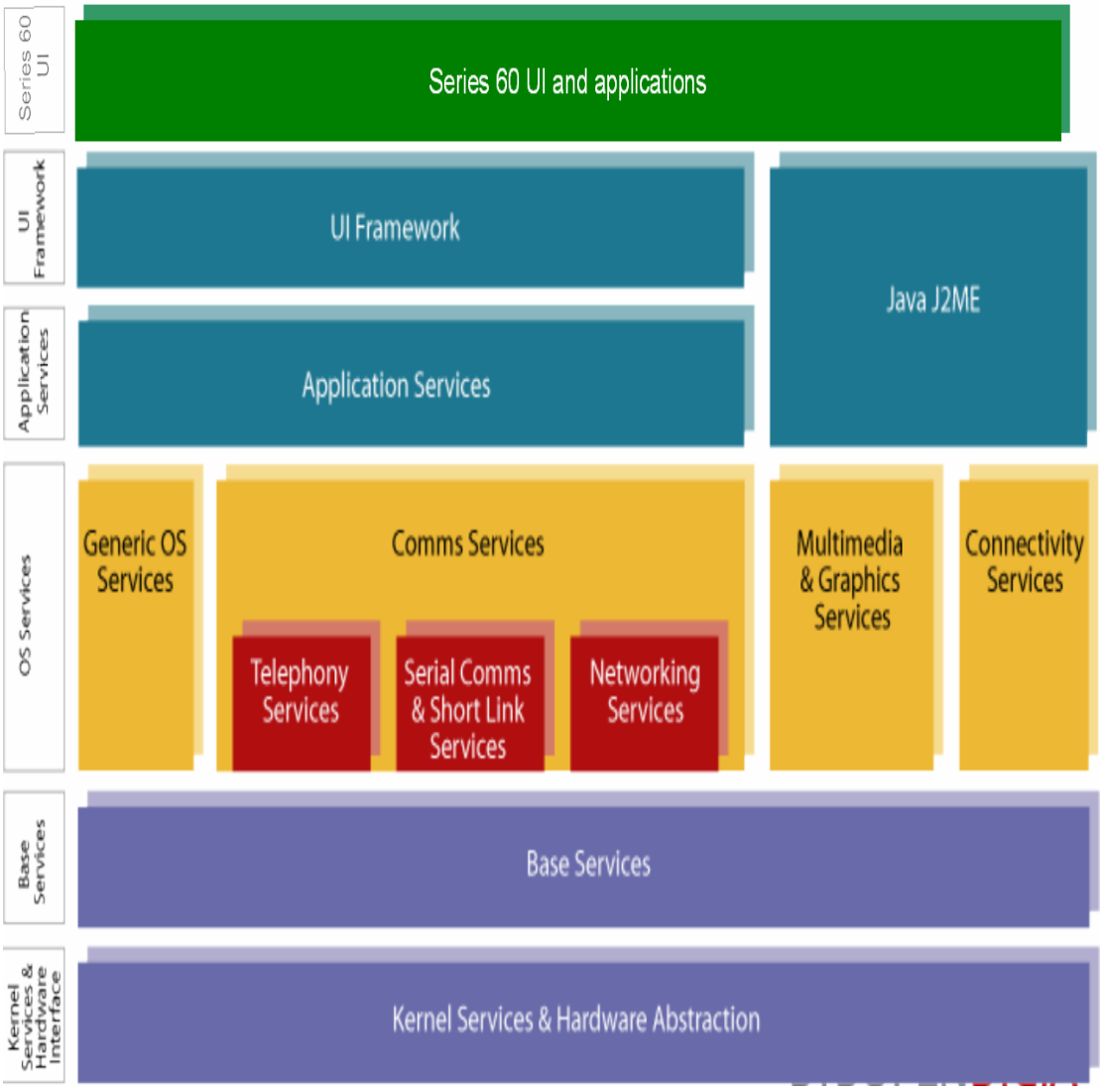
Ohjelmiston tuotannosta vastaavat tahot - erityisesti ylemmät sellaiset – odottavat, että testauksen automatisointi tekee testauksesta välittömästi parempaa, nopeampaa ja halvempaa. Joissain projekteissa näin saattaa käydäkin, mutta yleispätevä sääntö se ei ole. Halutut tulokset täytyy siis määritellä, jotta voidaan sanoa, ovatko automatisoinnilla saavutetut hyödyt sellaiset kuin haluttiin (Craig ja Jaskiel, 2002).

### 4.2.3 RIITTÄMÄTÖN PEREHDYTYS

Useimmiten testauksesta vastaavat tahot ymmärtävät perehdytyksen tärkeyden, mutta aina näin ei ole. Automatisoitu testaus ei tapahdu samalla tavalla kuin manuaalinen, ja testaajan täytyy osata käyttää kaikkia työkalun ominaisuuksia. Testaajan pitää myös tietää, mitä testataan, miksi ja miten saatuja tuloksia analysoidaan. Hyväkin testausautomaatiotyökalu menettää tehonsa, jos sitä ei käytetä oikein.

# 5 TESTAUS SYMBIAN TEXT SHELL -TASOLLA

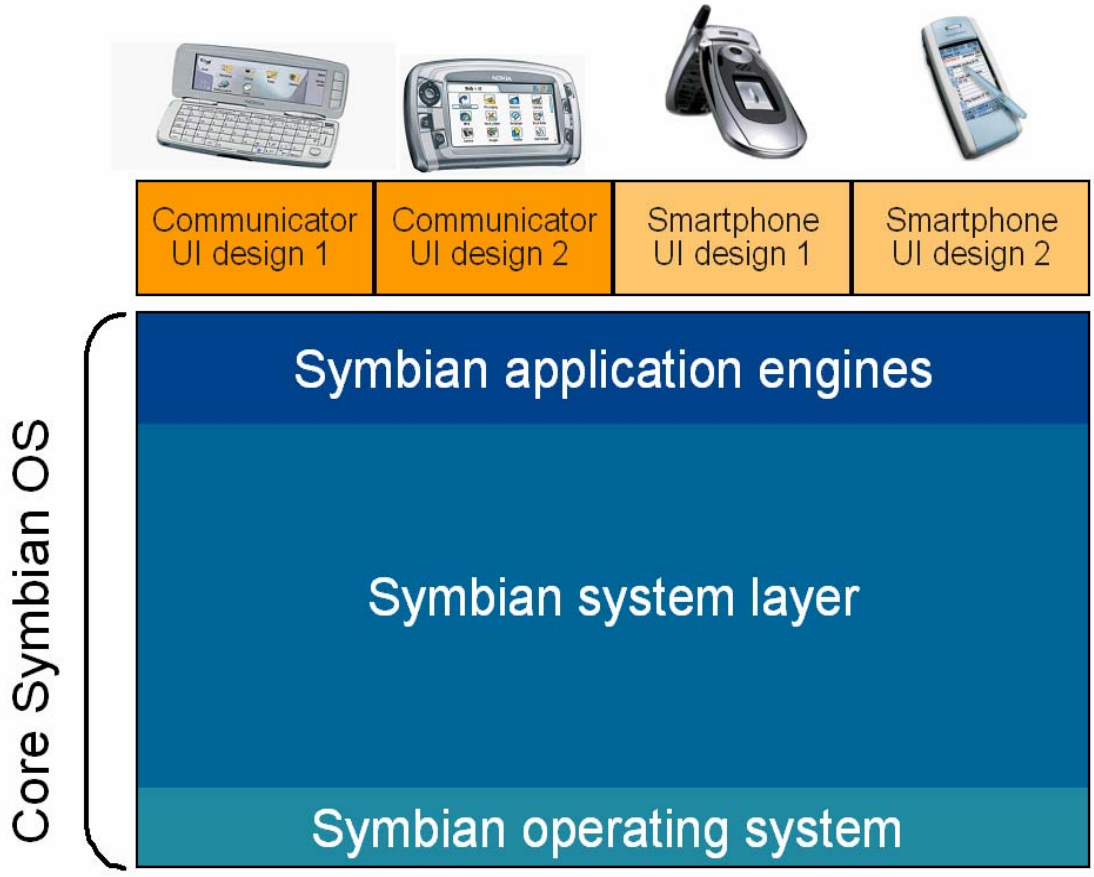
Tässä luvussa kerrotaan mitä on testaus Symbian text shell –tasolla. Tässä yhteydessä text shell –tason määritelmä on ”Symbian OS ilman UI-komponentteja”. Kuvassa 5.1 kolme alinta horisontaalista tasoa kuuluvat text shell –tason piiriin, sitä ylemmät S60-arkkitehtuuriin ja sen liitännäisiin. Käytännössä testattaviin osa-alueisiin kuuluvat mm. tietoliikennetoiminnallisuus ja tiedostopalvelut, ylipäätään alueet, joita voidaan testata ilman S60-alustaa tai vastaavaa.




Kuva 5.1: Text shell –tason testaus tapahtuu OS Services-tasolla ja sen alapuolella. (SYSOPENDIGIA Oyj (1), 2007)

Aluksi selvitetään, mikä on testikehys ja mihin sitä tarvitaan. Sen jälkeen tutustutaan ruohonjuuritason testaukseen, miten testiympäristö luodaan ja kuinka testejä ajetaan sekä manuaalisesti että automatisoidusti. Testaukseen liittyvään dokumentointiin, lähinnä tulosten raportointiin ja siinä käytettäviin työkaluihin, luodaan myös lyhyt katsaus. Lopuksi tarkastellaan tilannetta, jossa testaus onnistuu, eli löytää virheen. Tässä osuudessa paneudutaan myös siihen, kuinka mahdollisia virheen aiheuttajia pyritään etsimään, ja miten paikannetaan virheen sijainti. Myös virheiden raportoinnista kerrotaan lyhyesti.

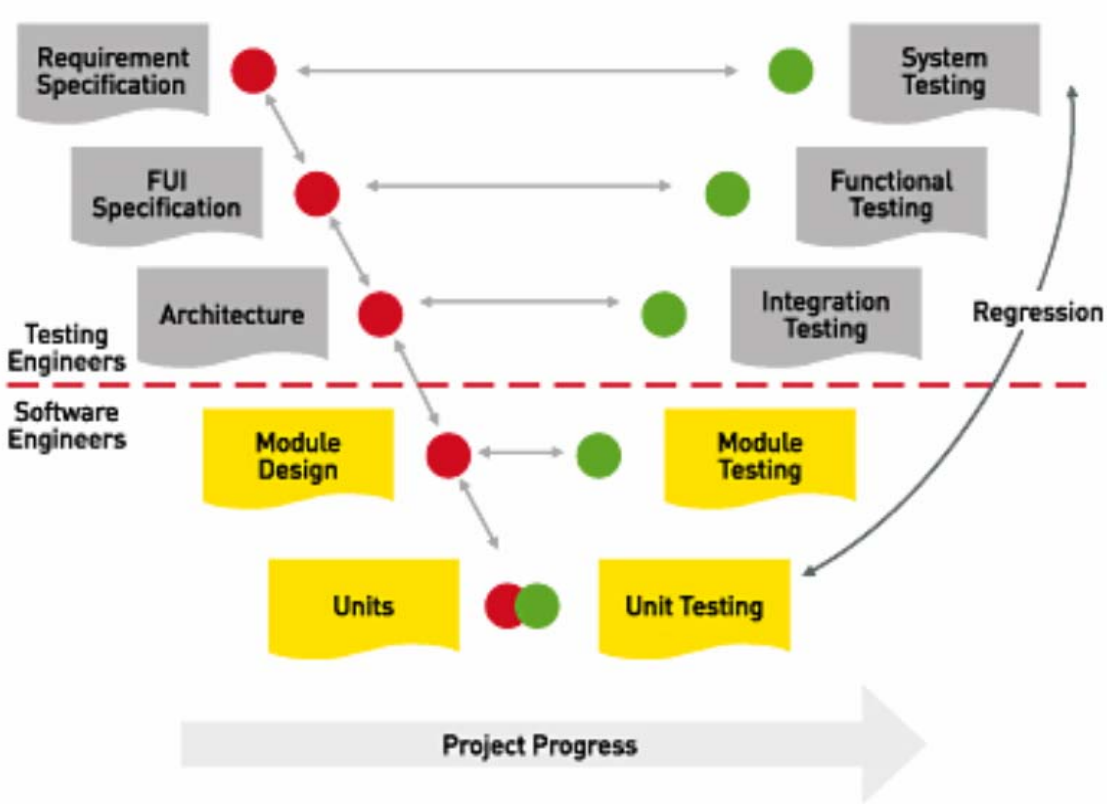
Asiat, joita tässä luvussa käydään läpi, eivät ole millään tavalla riippuvaisia yhdestä tietystä Symbian platformista, ympäristöstä tai jostain työkalusta. Text shell –tasolla toimittaessa on periaatteessa täysin sama, tuleeko sen päälle esimerkiksi S60-, S80- tai UIQ-ympäristö. Kuva 5.2 havainnollistaa tätä periaatetta. Teollisuudessa on käytössä myös lukuisia sekä julkisia että salassapitosopimusten alaisia yritysten sisäisiä työkaluja, joita käytetään tämäntyyppisissä tehtävissä. Tämän insinöörityön esimerkit ovat julkisista järjestelmistä sekä työkaluista.




Kuva 5.2: Symbian-käyttöjärjestelmän rakenne. (SYSOPENDIGIA Oyj (2), 2007)

## 5.1 TESTIKEHYS

Tämäntyyppisessä sulautetussa järjestelmässä tapahtuva automatisoitu testaus suoritetaan käytännössä aina jonkinlaisessa testikehyksessä (test framework). Testikehysjärjestelmiä on käytössä pelkästään matkapuhelimien testauksessa lukuisia, ja tässä yhteydessä käytetään esimerkkinä SYSOPENDIGIAn kehittämää julkista EUnit-testikehysjärjestelmää. Testikehyksen käytöllä on kiistattomat etunsa. Hyvin suunnitellussa testimoduulissa voidaan luoda uusia testitapauksia ainoastaan skriptejä muuttamalla. Komponentteja ei tarvitse kääntää uudelleen silloin, kun niihin ei ole tullut muutoksia. Myös kattava virhetilanteista toipuminen sekä monesti kuvaavammat virheilmoitukset puhuvat testikehyksen käytön puolesta. EUnit-järjestelmä on suunniteltu yksikkö- ja moduulitestaukseen. Kuvassa 5.3 on esitetty testauksen V-malli myös testaus- ja ohjelmistoinsinöörin näkökulmasta.

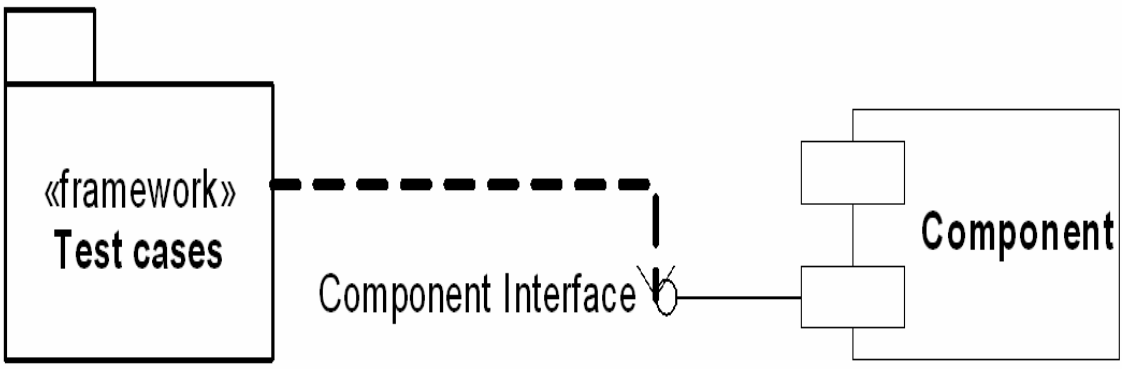



Kuva 5.3: Yksikkö- ja moduulitestaus ohjelmistoprosessin aikajanassa. (SYSOPENDIGIA Oyj (3), 2007)

### 5.1.1 EUNIT TESTIKEHYSJÄRJESTELMÄ

EUnit-testikehysjärjestelmä on SYSOPENDIGIAn kehittämä testaustyökalu matkapuhelinten testaukseen. EUnitilla voidaan suorittaa sekä yksikkö- että moduulitestausta, ja tässä yhteydessä keskitytään moduulitestaustapaan.

Tämäntyyppisessä kehyksessä itse testikoodi on vain pieni osa koko testiohjelmiston koodia. Suurimman osan muodostavat kehyksen runko (skeleton) ja muu rakenne. Testaava koodi on usein yksi Symbian C++ -luokka, josta ulospäin näkyvät vain rajapintafunktiot, joita testikehys tarvittaessa kutsuu. Testaava koodi täytyy kääntää kehyksen ymmärtämäksi testi-dll -tiedostoksi alustakohtaisesti. Myös mahdolliset testaavan koodin käyttämät komponentit täytyy kääntää uudelleen. Nämä toimenpiteet täytyy suorittaa binaariyhteensopivuuden (binary compability) säilyttämiseksi.




Kuva 5.4: EUnit-testikehyksen moduulitestausrakenne. (SYSOPENDIGIA Oyj (3), 2007)

EUnit-järjestelmässä on myös kattavat wizard-toiminnot, jotka nopeuttavat kehittäjän työtä testausrungon luomisessa. Testikoodi on rakenteeltaan sellaista, että sitä on helppo käyttää uudelleen ja useat palvelut on suunniteltu nopeuttamaan kehittäjän sekä testaajan työtä.

## 5.2 TESTIEN SUORITTAMINEN

Yksinkertaisimmillaan testin suoritus on hyvin suoraviivaista ja kokeneelle testausinsinöörille jopa rutiininomaista. Useimmiten asia ei kuitenkaan ole näin varsinkaan alemman tason testauksessa, vaan asiaan liittyy monia vaiheita, joita ei ilman vahvaa ammattitaitoa ole helppoa ratkaista.

Itse testien suorittaminen tapahtuu karkeasti tarkasteltuna kahdella tavalla: joko manuaalisesti tai automatisoidusti. Kummassakin tapauksessa voidaan säädellä suoritettavien testien määrää ja ympäristöstä riippuen myös järjestystä. Testiympäristöllä on suuri merkitys testien ajotavassa, joka voi vaihdella hyvinkin paljon.

### 5.2.1 TESTIYMPÄRISTÖN LUOMINEN

Testiympäristön luomiseen voi tilanteesta riippuen kuulua montakin vaihetta. Monesti olennainen osuus on oikean käyttöjärjestelmäversion asennus puhelimeen. Myös testitapaukseen liittyvät komponentit voivat olla kohdennettuja, esimerkiksi tietoliikenneominaisuuksia testattaessa ei tarvita välttämättä kameraominaisuuksia ja niin edelleen. Text shell -tasolla puhelimeen asennettavat ohjelmistoversiot ovat yleensä hyvin pelkistettyjä, ainoastaan kyseisen testikategorian tarvitsemat komponentit ovat asennuksessa mukana.

Laajamittaisessa matkapuhelintestauksessa pyritään kirjaimellisesti testaamaan aivan kaikkea. Jokainen voi omien käyttökokemuksiensa perusteella päätellä, minkälaisia toimintoja puhelimessa on ja minkälaista toiminnallisuutta kyseisten toimintojen alla mahdollisesti piilee. Kaikkia näitä ominaisuuksia testataan. Muutamia esimerkkejä mainitakseni erilaisia testiympäristöjä ovat mm. kahden puhelimen väliset tietoliikennetestit, muistikortteihin liittyvät testit, puhelimen ja PC:n väliset tietoliikennetestit USB-yhteyden kautta ja niin edelleen.

### 5.2.2 MANUAALISET TESTIAJOT

Käytännössä testiajot pyritään suorittamaan automatisoidusti aina, kun se on mahdollista. On kuitenkin olemassa joitain erikoistapauksia, jolloin yksittäisten testitapausten välillä tarvitaan testausinsinöörin toimenpiteitä esim. laitteiston mukauttamiseen. Tarkastellaan fiktiivistä tilannetta, jossa haluttaisiin testata eri valmistajien PC:n ja puhelimen välisiä USB-kaapeleita peräkkäisinä testitapauksina. Olisi lähestulkoon mahdotonta automatisoida tällaisia testitapauksia, joissa kahden laitteen välinen kaapeli on vaihdettava joka testitapauksen välissä. Tietenkin jokaiselle tapaukselle voisi luoda oman testiympäristön, mutta tällainen vaihtoehto olisi huomattavasti kalliimpi ja hitaampi toteuttaa kuin manuaalinen testiajo.

Kun testejä kehitetään tai selvitetään virheiden aiheuttajan syytä, on tietenkin voitava ajaa yksittäisiä testejä ja tämä tapahtuu joustavimmin manuaalisesti. Koko testisetin ajaminen yhden testin vuoksi on turhaa ja jopa haitallista. Yleensä ammattitasoinen testiympäristö mahdollistaa joustavan testien ajotapojen muokkauksen.

### 5.2.3 AUTOMATISOIDUT TESTIAJOT

Matkapuhelinten automatisoidussa testauksessa käytetään aina jonkinlaista testausjärjestelmää, joka koostuu useista eri tasoista. Tällaisia tasoja ovat ainakin testikehys ja sen päällä oleva automaattinen testausjärjestelmä. Järjestelmä hallinnoi siihen liitettyjä puhelimia sekä erilaisia testisarjoja. Sen avulla pystytään myös asentamaan puhelimeen tarvittavat ohjelmistot ja komponentit. Testitapausten kulku ja tulokset tallentuvat järjestelmään, josta niitä voidaan myöhemmin (ja ajon aikana) tarkastella ja halutessa verrata aiempiin tuloksiin.

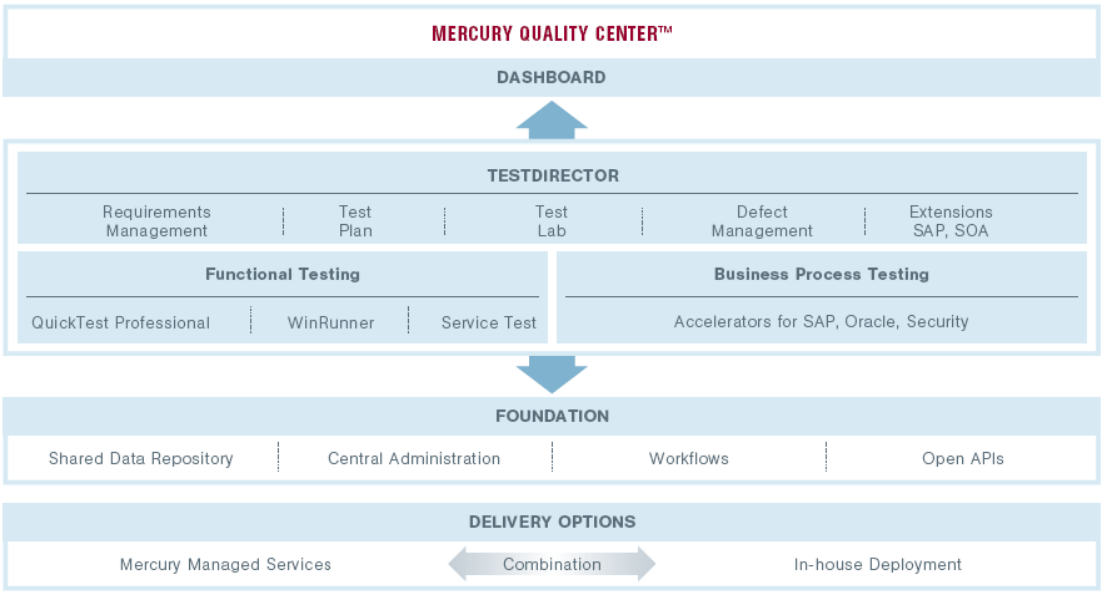
Järjestelmää voidaan käyttää joko paikallisesti omalta työasemalta tai TCP/IP-protokollan kautta muualla sijaitsevassa ympäristössä, jolloin järjestelmän käyttöliittymä on siis omalla työasemalla. Järjestelmän ohjaamaa testiajoa ohjataan yleensä skriptitiedoston avulla, johon järjestelmäkohtaiset komennot tehdään.

## 5.3 TESTITULOSTEN RAPORTOINTI

Testitulosten raportointi on tärkeä ja ulospäin näkyvin osa testausta. Raportoinnin perusteella testauksesta vastaavat tahot seuraavat testauksen edistymistä sekä raportoivat tilannetta eteenpäin. Tähän liittyy olennaisesti myös testauksen riittävyyden arviointi, josta kerrottiin tarkemmin alakohdassa 3.3. Yleensä testitulosten raportointiin käytetään jotain keskitettyä ratkaisua. Tässä tapauksessa tarkastellaan Quality Center -ohjelmistoa, joka on hyvin yleisessä käytössä oleva ratkaisu ohjelmistotestauksessa.

### 5.3.1 QUALITY CENTER

Quality Center -ohjelmisto (ja muut vastaavat ohjelmistot) on tärkeä väline ohjelmistotestauksessa. Jo siinä vaiheessa, kun yksittäiset testit on suunniteltu ja toteutettu, ne voidaan kirjata ohjelmiston ”Test Plan”-osioon. Sieltä käsin testauksesta vastaavat tahot valitsevat testausvaatimusten mukaiset testitapaukset, jotka lisätään mukaan ajettavien testien joukkoon. Testausinsinöörit näkevät ”Test Lab”-osiosta testit, jotka täytyy ajaa, ja voivat myös merkitä testien tulokset kyseiseen osioon. Quality Centeriin testikehittäjät lisäävät myös testien kuvaukset ja ”ajo-ohjeet”, joiden avulla testausinsinööri saa riittävän tiedon testiajon eri vaiheista.




Kuva 5.5: Quality Center –ohjelmisto kattaa tärkeimmät testiraportointivälineet. (Mercury, 2007)

## 5.4 KUN TESTAUKSEN AVULLA LÖYDETÄÄN VIRHE

Testauksen tarkoitus on löytää ohjelmistosta virheitä ja näin myös lähes poikkeuksetta tapahtuu. Kun testi ei mene läpi, testausinsinööri toimii tietyn kaavan mukaisesti, jotta prosessi etenee suunnitelman mukaisesti. Ensimmäiseksi pyritään kartoittamaan virheen aiheuttaja, jos mahdollista. Sen jälkeen virhe täytyy paikantaa niin tarkasti kuin testausympäristössä on mahdollista. Kun kaikki vaaditut esitutkimustoimenpiteet on suoritettu, testausinsinööri tekee virheraportin, jonka perusteella mm. testikehittäjät alkavat ratkaista ongelmaa.

### 5.4.1 VIRHEEN AIHEUTTAJAN KARTOITUS

Kun testiajo joko keskeytyy tai päättyy virheilmoitukseen, testi ei ole mennyt läpi hyväksytyllä tavalla. Virheilmoituksia voi olla tilanteesta riippuen monenlaisia, mutta alemman tason testauksessa virheilmoituksen mukana on Symbian Error Code, esim. ”KErrNoMemory”. Siitä voidaan päätellä virheen johtuvan mahdollisesti muistivuodosta tai muistin vähyydestä. Jos laite on kaatunut, ei virheilmoituksen antamiseen asti tietenkään päästä. Joskus saattaa myös käydä niin, että testiympäristössä on jotain vikaa, eli virhe ei ole todellinen. Tällaisten tapausten takia testiympäristö kannattaa aina tarkistaa virheen ilmaantuessa, ellei virheen luonne ole hyvin selvä.

### 5.4.2 VIRHEEN PAIKANTAMINEN

Virheen paikantaminen alkaa testausinsinöörin testiajon aikana tekemistä havainnoista. Silmämääräiset havainnot eivät tietenkään riitä, vaan tarvitaan tarkempaa tietoa siitä, missä vaiheessa testiajoa virhe on syntynyt. Tällöin testi ajetaan uudelleen niin, että puhelimeen liitetään ns. trace-laitteisto, jonka avulla puhelimen pieninkin tapahtuma rautatasolta asti saadaan selvitettyä. Näitä jäljityksiä, ”traceja”, lukemalla saadaan virheen aiheuttaja tai ainakin testiajon vaihe selville. Näiden tietojen perusteella testikehittäjien on helppo paikantaa kooditasolla vaihe, jossa virhe syntyy tai mitä toiminnallisuutta kyseisessä koodiosiossa käytetään.

### 5.4.3 VIRHEEN RAPORTOINTI

Kun kaikki tarvittavat selvitykset virheen syystä ja sijainnista on tehty, tehdään virheraportti. Virheraportin perusteella siitä vastaavat tahot ohjaavat selvitystä eteenpäin, jotta virhe saadaan mahdollisimman pian korjattua. Testausinsinöörin täytyy varmistaa, että virhe on toistettavissa eli virhe on todellinen. Tämä on välttämätöntä, jotta raportin perusteella asiaa selvittävät tahot voivat ylipäätään aloittaa työnsä.

Raporttiin tulee mukaan sanallinen selvitys virheen laadusta. Tämä osio on hyvin tärkeä, koska sen avulla virhettä korjaava taho saa nopeasti kuvan siitä, mikä virheen aiheuttaja oikeasti on. Tietenkin tämänkaltainen ennakoiva arviointi vaatii testausinsinööriltä kokemusta ja ammattitaitoa, jotta näköala ylettyy myös piiri- ja kooditasolle asti.

Raporttiin kirjataan myös käytetyt ohjelmistoversiot, käytössä olleen puhelimen prototyypin versio sekä testitapausta koskevat tiedot. Raportti täytyy lähettää heti oikeille tahoille, jotta turhaa viestiketjua ei turhaan syntyisi.

Viimeisenä, muttei vähäisimpänä, raportin liitteeksi lisätään testin lokitiedostot, joita voi olla useita testitapauksesta riippuen. Testausinsinöörin kannattaa lisätä sanalliseen selvitykseensä lokien olennaisimmat kohdat, jotta selvityksen jatkajan työ lähtisi mahdollisimman nopeasti käyntiin.

Useimmissa testiraporttityökaluissa on ominaisuus, joka asettaa virheelle tilan, esim. ”virhe löydetty” tai ”selvityksen alla”. Tämän tilan muuttuessa oikeat tahot saavat viestin selvityksen tilasta.

# 6 YHTEENVETO

Tämän insinöörityön tarkoituksena oli antaa mahdollisimman kattava selvitys matkapuhelimien text shell –tason testauksesta ja sen automatisoinnista. Tietenkin työ täytyi rajata hyvin tarkkaan, jotta työn pituus ja aihealue saatiin järkeviin mittasuhteisiin. Tavoitteena oli myös antaa lukijalle vähintään yleiskuvaus mobiilitestauksesta, sen periaatteista ja teoriasta.

Joihinkin asioihin paneuduttiin hieman tarkemmin, koska ne olivat tärkeitä kokonaisuuden ymmärtämisen kannalta. Yleisen ohjelmistotuotannon periaatteiden omaksumista ei voi mielestäni missään nimessä väheksyä, koska niiden pohjalle vasta voi rakentaa yksityiskohtaisempaa tietämystä erikoisosaamisalueista. Työssä onkin annettu painoarvoa ohjelmistotuotannosta, testauksesta ja testausautomaatiosta kertoville luvuille. Tietenkin itse text shell -tason testauksesta kertovalla luvulla on hyvin tärkeä merkitys. Nämä luvut yhdessä muodostavatkin tasapainoisen kokonaisuuden, jonka luettuaan lukija ymmärtää aihealuetta paremmin.

Mielestäni työ onnistui tavoitteessaan hyvin. Tämän työn luettuaan testausinsinöörillä on huomattavasti paremmat valmiudet päästä kiinni työhönsä perusteet tuntien ja tehokkaalla otteella. Monia tehtäviä on mahdollista tehdä ilman asian syvempää tuntemusta, mutta mielestäni se ei ole pitkällä aikavälillä järkevää. Työn tekijän on aina hyvä tietää, miksi hän tekee jotain, ei ainoastaan miten.

Työn toinen tarkoitus oli syventää tietämystäni ohjelmistotestauksen teoriasta ja periaatteista. Aiheesta löytyy kirjallisuutta hyvin paljon, ja jouduinkin suorittamaan karsintaa lähdekirjallisuutta valitessani. Myös tässä tavoitteessa onnistuin kuten pitikin. Prosessin aikana tuli esille monia asioita, joita ei aiemmissa opinnoissa tai työtehtävissä ollut tullut vastaan. Näistä asioista on ja tulee olemaan paljon hyötyä työtehtävissäni matkapuhelimiin liittyvän testiautomaation parissa.

# LÄHDELUETTELO

**Painetut lähteet**

Craig, R ja Jaskiel, S. *Systematic Software Testing* (2002)

Glenford J. Myers, *The Art of Software Testing* (2004)

Ilkka Haikala ja Jukka Märijärvi, *Ohjelmistotuotanto* (2004)

Jorgensen, *Software Testing: A Craftsman's Approach*. (2002)

Maarit Harsu, *Ohjelmien ylläpito ja uudistaminen*, TTY:n kurssi (2006)

Mark Fewster ja Dorothy Graham, *Software Test Automation* (1999)

Mika Katara, *Ohjelmistojen testaus*, TTY:n kurssi (2006)

Pressman Roger S., *Software Engineering: A Practitioner's Approach* (2000)

Steve McConnell, Ohjelmistoprojektit: *Selviytymisopas* (1998)

SYSOPENDIGIA Oyj (1), *Symbian Architectural View*, koulutusmateriaali (2007)

SYSOPENDIGIA Oyj (2), *Symbian Overview for Developers*, koulutusmateriaali (2007)

SYSOPENDIGIA Oyj (3), *EUnit Basics*, koulutusmateriaali (2007)

**Sähköiset lähteet**

Ellemtel, *C++-tyyliopas*, http://www.doc.ic.ac.uk/lab/cplus/c++.rules/

Mercury, *Quality Center -ohjelmisto*, http://www.mercury.com/us/products/quality-center/