# Wreckamovie.com-verkkopalvelun ohjelmistotestauskäytännöt

Viestinnän koulutusohjelma
Verkkoviestintä
Opinnäytetyö
27.4.2009

Vesa Nieminen

Metropolia

TIIVISTELMÄSIVU

| Koulutusohjelma | Suuntautumisvaihtoehto |
|---|---|
| Viestinnän koulutusohjelma | Verkkoviestintä |

Tekijä
Vesa Nieminen

Työn nimi
Wreckamovie.com-verkkopalvelun ohjelmistotestauskäytännöt

Työn ohjaaja/ohjaajat
Juhana Kokkonen

| Työn laji | Aika | Numeroidut sivut + liitteiden sivut |
|---|---|---|
| Opinnäytetyö | 27.4.2009 | 35 + 7 |

TIIVISTELMÄ
Työ käsittelee Wreckamovie.com-verkkopalvelun ohjelmistotestauskäytäntöjä. Ohjelmistotestauskäytäntöjä analysoidaan kirjallisuuden ja kirjoittajan kokemuksen pohjalta. Tavoitteena oli löytää ohjelmistotestauskäytäntöjen heikkoudet ja vahvuudet ja niiden pohjalta tarjota ehdotuksia palvelun ohjelmistotestauksen parantamiseksi. Lisäksi tavoitteena oli tarjota lukijalle esimerkki ohjelmistotestauksen toteuttamisesta todellisessa verkkopalveluprojektissa.

Työ on hankkeistettu Star Wreck Studios Oy:lle. Kirjoittaja on työskennellyt palvelun kehittäjänä kehitysprojektin alusta lähtien ja vastannut ohjelmistotestauksen suunnittelusta ja toteuttamisesta.

Työ esittelee Wreckamovie.com-verkkopalvelun ohjelmistotestauksen tarkastelun kannalta olennaisia ohjelmistotestausprosesseja, -strategioita ja -työkaluja. Suurin osa huomiosta kiinnittyy automatisoidun kehittäjätestauksen käsittelyyn, jolla tarkoitetaan testausta, jossa ohjelmoija kirjoittaa automatisoidut testit testaamaan omaa työtään. Myös perinteisempi ohjelmistotestaus esitellään.

Wreckamovie.com-verkkopalvelu on toteutettu käyttäen Ruby on Rails -verkkosovelluskehystä, joka käyttää Model-View-Controller-arkkitehtuurimallia. Testaus on toteutettu niin, että jokaista arkkitehtuuritasoa testataan erillisinä toisistaan. Testauskäytännöt on analysoitu tarkastellen jokaisen tason testejä erikseen.

Työn tuloksena on parannusehdotuksia, joita toteuttamalla Wreckamovie.com-verkkopalvelun ohjelmistotestausta voitaisiin kehittää. Testien yleisenä ongelmana on, että samantyyppisten asioiden testaaminen on toteutettu hieman eri tavoilla. Parannusehdotuksina esitetään yleisten toiminnallisuuksien testaamiseen vaadittavien testien paketoimista ja testausmakrojen käyttöä, mitkä yhtenäistäisivät testejä ja tekisivät jatkotestaamisesta selkeää ja tehokasta. Näkymätasolla parannuksena ehdotetaan JavaScript-koodin eli käyttäjän selaimessa tapahtuvien toiminnallisuuksien testaamista.

Teos/Esitys/Produktio
-

Säilytyspaikka
Aralis-kirjastokeskus

Avainsanat
Ohjelmistotestaus, ohjelmistokehitys, Ruby, Ruby on Rails, WWW

Metropolia

Culture

| Degree Programme in | Specialisation |
|---|---|
| Media | New Media Design |

Author
Vesa Nieminen

Title
Software Testing Practices in Wreckamovie.com Web Application

Tutor(s)
Juhana Kokkonen

| Type of Work | Date | Number of pages + appendices |
|---|---|---|
| Bachelor´s Thesis | 27.4.2009 | 35 + 7 |


The topic of this thesis is software testing practices in Wreckamovie.com web application. Software testing practices are analyzed on the basis of research literature and the author's own experience. The goal is to find weaknesses and strengths of the software testing practices and based on those give suggestions on how the service could develop its software testing. Also, a further goal is to provide an example of software testing in a real web service project.

The thesis has been done in collaboration with Star Wreck Studios Ltd. The author has been working as a developer of the service from the beginning and has been responsible for planning and executing the software testing.

The thesis introduces software testing processes, strategies, and tools which are relevant for examining the software testing of Wreckamovie.com. Most attention is on automated developer testing, which means testing in which the programmer writes automated tests for the work he has done. Moreover, traditional software testing is also discussed.

Wreckamovie.com web application is implemented by using Ruby on Rails web application framework, which uses Model-View-Controller-architecture model. The testing is executed separately for each level of architecture. Testing practices are analyzed by examining tests for each level separately.

The end result consists of recommendations on how software testing in Wreckamovie.com could be improved. The overall problem with the tests is that the same kind of things are tested in different ways. This could be improved by packaging tests needed to test common functionality and by using testing macros. It would unify the tests and would make future testing clear and efficient. Furthermore, a further improvement could occur when introducing the testing of JavaScript code.


Work / Performance / Project
-

Place of Storage
Aralis Library and Information Centre


Keywords
Software testing, software develoment, Ruby, Ruby on Rails, WWW

SISÄLLYS

# 1 JOHDANTO

Opinnäytetyöni aiheena on Wreckamovie.com-verkkopalvelun ohjelmistotestaus-käytäntöjen analyysi. Siihen liittyen esittelen asiaankuuluvia ohjelmistotestauksen menetelmiä. Olen erikoistunut opinnoissani ohjelmointiin ja muuhun tekniseen toteuttamiseen. Lähes poikkeuksetta opinnoissa tehdyissä projekteissa olen hakeutunut tehtävään, jossa vaaditaan teknisen toteuttamisen taitoja ja kehittänyt itseäni opintojen puitteissa ja omalla ajallani tähän osa-alueeseen.

Verkkopalveluiden toteuttamiseen käytetyissä tekniikoissa olen perehtynyt erityisesti Ruby on Rails -verkkosovelluskehykseen. Olen seurannut Ruby on Rails -yhteisöä jo vuosia internetin kautta lukemalla blogeja, uutissivustoja, haastatteluja, postituslistoja ja katsomalla konferensseista saatavilla olleita videoesityksiä. Ruby on Rails -yhteisö arvostaa laajalti ohjelmistotestaamista. Aihe on usein esillä yhteisön keskusteluissa ja esimerkiksi alan tapahtumissa, kuten RailsConf-tapahtumassa, jossa on lähes poikkeuksetta muutama esitys, jotka jollain tavalla liittyvät ohjelmistotestaamiseen ja siinä käytettäviin työkaluihin. Yhteisön kiinnostus on ollut yksi suurimmista tekijöistä, jotka ovat ajaneet minut alun perin tutustumaan ohjelmistotestaukseen.

Koulutusohjelmani ohjelmointikoulutus keskittyy perusteisiin eikä käsittele ohjelmistotestaamista lainkaan. Koen kuitenkin, että moni linjaltamme valmistunut toimii työssä, jossa ohjelmointiosaamista tarvitaan jatkuvasti. Käytettyäni ohjelmistotestaamista säännöllisesti muutamia kuukausia koin, että ohjelmistotestaamiseen tutustuminen ja sen käyttäminen olisi hyödyllistä monille. Päätin, että haluan esitellä opinnäytetyössäni ohjelmistotestaamista.

Wreckamovie.com-verkkopalvelu, jota kehitän osa-aikatyössäni opiskelujeni ohella, vaikutti hyvältä analysoinnin kohteelta. Analyysi antaa lukijalle esimerkin ohjelmistotestauksen toteuttamisesta ohjelmistoprojektissa.

Opinnäytetyöni tarkoituksena on antaa tietoa ohjelmistotestauksesta laadun parannukseen käytettävänä menetelmänä. Tieto on suunnattu koodin kanssa työskenteleville, jotka haluavat parantaa oman työnsä laatua. Lisäksi siitä saatua tietoa käytetään hyväksi kehitettäessä Wreckamovie.com-palvelun testauskäytäntöjä.

Wreckamovie.com on verkkopalvelu, joka mahdollistaa eri tyyppisten elokuvien tekemisen yhteisöllisesti suuren ihmismäärän yhteisellä työpanoksella. Käyttäjät voivat perustaa palveluun projekteja ja määrittää niihin tehtäviä, joiden lopputuloksena on valmis elokuva. Pohjimmaisena ideana on tarjota käyttäjille mahdollisuus toteuttaa itseään ja antaa ihmisille mahdollisuus luoda ammattilaistason elokuvia käyttämällä hyväksi internetin ihmisiä yhdistävää voimaa.

Wreckamovie.com-palvelun tekninen kehitys on aloitettu syksyllä 2007 ja olen ollut siinä mukana alusta asti. Palvelu on toteutettu pienellä tiimillä, joka koostuu lähes kokonaisuudessaan osa-aikaisista työntekijöistä. Tämän vuoksi projektin eri jäsenten työnkuvat ovat olleet laajoja. Itse olen muun muassa suunnitellut, piirtänyt grafiikkaa, tehnyt HTML:ää ja CSS:ää, ohjelmoinut ja ylläpitänyt palvelimia. Ohjelmointia on lisäkseni tehnyt kaksi henkilöä, mutta heinäkuusta 2008 lähtien olen ollut ainoa ohjelmoija ja nykyisestä koodista suurin osa on minun kirjoittamaani. Testit ovat kaikki minun tekemiäni.

Ohjelmistotestaus on prosessi, jonka tarkoitus on pohjimmiltaan parantaa ohjelmiston laatua. Käsittelen tässä työssä sekä perinteisempää ohjelmistotestausta, jonka päämääränä on löytää ohjelmistosta mahdollisimman paljon virheitä, että uudempia ohjelmistotestauksen suuntia, joiden tavoitteena on parantaa ohjelmistojen laatua muillakin tavoilla kuin vain virheitä vähentämällä.

Työni tutkii Wreckamovie.com-verkkopalvelun testauskäytäntöjä ja esittelee niiden heikkouksia, vahvuuksia ja parannusehdotuksia kirjallisuuden ja oman kokemukseni pohjalta. Rajaan tarkastelun kohteeksi sovellusten automatisoidun kehittäjätestauksen, jolla tarkoitan sovelluksen kehittäjien itse harjoittamaa testausta. En siis käsittele syvällisesti laadunvarmistukseen erikoistuneiden asiantuntijoiden käyttämiä

hyväksymis- ja muita testausmenetelmiä, eikä tavoitteena ole myöskään tarjota täydellistä selontekoa eri testausvaihtoehdoista ja työkaluista.

Wreckamovie.com-projektin testauskäytäntöjen muotoutumiseen on vaikuttanut vahvasti edellä mainitut tiimin koostumus ja koko. Testaukseen ja laadunvalvontaan ei ole ollut resursseja palkata erillistä henkilöä, vaan testauksen on suorittanut henkilö, joka on kirjoittanut testattavan koodin. Opinnäytetyöni ja siinä käsitellyt menetelmät ovatkin suunnattu enemmän ohjelmoijille kuin täysipäiväisille ohjelmistotestaajille. Käsiteltävä projekti luo painotuksen myös ohjelmistotestaamisen osa-alueisiin, jotka ovat tärkeimpiä verkkosovelluksia testattaessa.

Työ koostuu kahdesta osiosta. Ohjelmistotestaus-osiossa käsitellään työn näkökulmasta relevantteja ohjelmistotestausprosesseja, -strategioita ja -työkaluja. Wreckamovie.com-osiossa käsitellään verkkopalvelua ja sen testauskäytäntöjä. Ensimmäisen osion aluksi käsittelen perinteistä ohjelmistotestausta ja sen hallitsevia testausstrategioita musta- ja lasilaatikkotestausta. Sen jälkeen selvitän, mitä tarkoitetaan automatisoidulla ohjelmistotestauksella ja mitä hyötyjä sillä saavutetaan. Sitten käsittelen kehittäjätestausta, joka on ohjelmistotestauksen alalaji, ja sen alalajeja testivetoista (test-driven development) ja käyttäytymisvetoista kehitystä (behaviour-driven development). Sen jälkeen käyn läpi ohjelmistotestauksen kautta saavutettavia hyötyjä, sekä automatisoidussa ohjelmistotestauksessa käytettäviä työkaluja. Lopuksi selvitän vielä verkkosovellusten automatisoidun ohjelmistotestauksen erityispiirteitä.

Wreckamovie.com-osiossa selvitän, minkälainen palvelu Wreckamovie.com on ja mitä tekniikoita ja työkaluja käyttäen se on rakennettu. Wreckamovie.com on toteutettu käyttämällä Ruby on Rails -verkkosovelluskehystä, mikä määrittää jonkin verran sitä, miten testaus on toteutettu. Projektin testaustavan ja testien rakenteen läpikäyntiä seuraa Wreckamovie.com-projektin ohjelmistotestauskäytäntöjen analyysi.

Lopuksi pohdin vielä muiden laadunvarmistusmenetelmien suhdetta ohjelmistotestaamiseen.

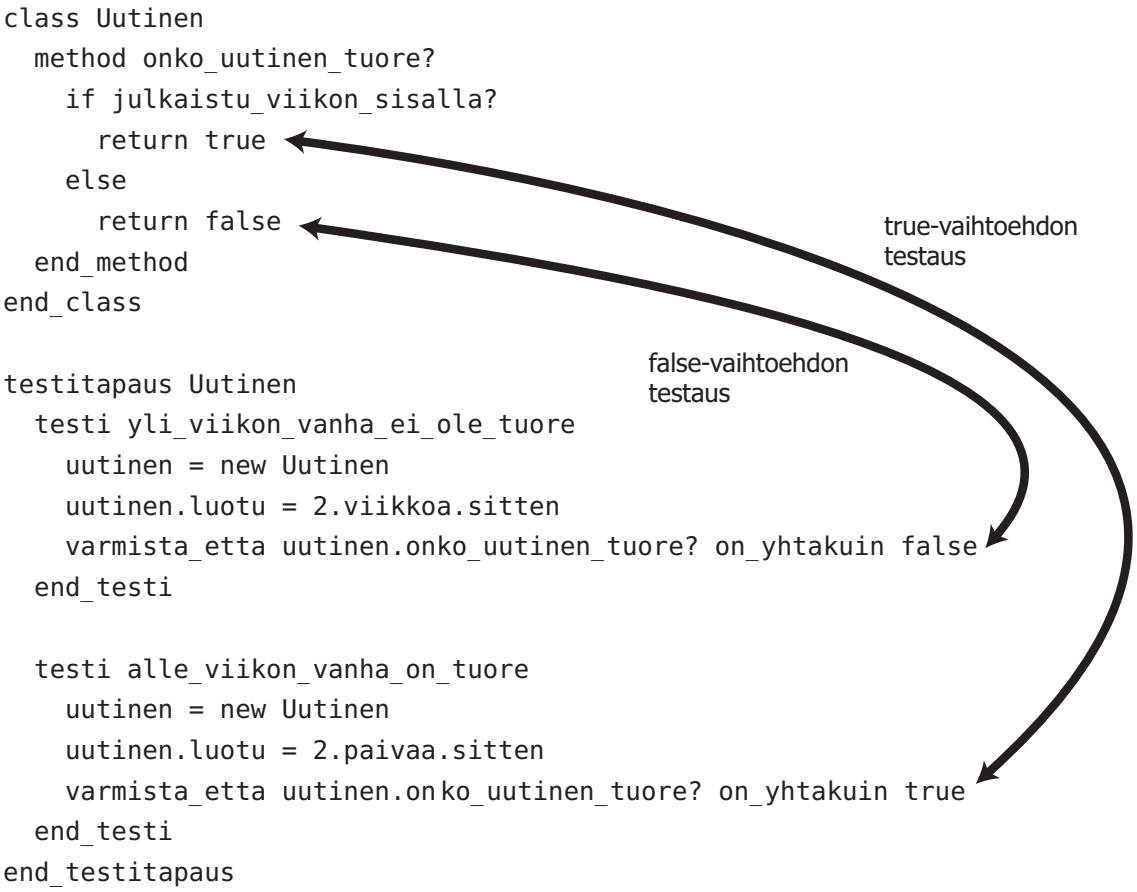
# 2 OHJELMISTOTESTAUS

Alunperin vuonna 1979 julkaistussa ohjelmistotestaamisen perusteoksessa Glenford J. Myers määrittelee ohjelmistotestauksen prosessiksi tai sarjaksi prosesseja, joiden tarkoituksena on varmistaa, että ohjelmakoodi tekee sen, mitä sen on suunniteltu tekevän, ja että se ei tee mitään, mitä sen ei ole tarkoitettu tekevän (Myers, Badgett, Thomas & Sandler 2004, 1–2). Pohjimmiltaan tämä tarkoittaa sitä, että testauksen tarkoitus on löytää sovelluksesta virheitä. Määritelmä saattaa vaikuttaa nurinkuriselta, koska siinä keskitytään virheisiin virheettömyyden sijaan, mutta siihen on syynsä. Ihmiset ovat tavoiteorientoituneita ja tavoitteella on siksi tärkeä psykologinen vaikutus. Jos tavoitteena on todistaa, että ohjelmassa ei ole virheitä, ajaudumme alitajuisesti kohti tätä tavoitetta. Emme siis yritä parhaamme mukaan etsiä ohjelmasta virheitä, vaan keskitymme tekemään testeistämme sellaisia, että ne eivät aiheuttaisi virheitä ohjelmassa. Näin ollen ohjelman virheet jäävät löytymättä, kun keskitymme välttelemään niitä, mikä on ohjelman laadun kannalta luonnollisesti huono asia. (Mts. 6.)

Perinteinen malli on hyvin sidoksissa sovelluskehityksen prosessimalliin, jossa ensin suunnitellaan ja sitten ohjelmoidaan, jonka jälkeen koodi annetaan eteenpäin jollekin toiselle henkilölle testattavaksi. Ohjelmoijan ja testaajan roolin eriyttäminen toisistaan nähdään hyvänä asiana, koska testaaja pystyy lähestymään testattavaa koodia ilman omasta työstä löytyvien virheiden aiheuttamaa tunnelatausta ja pystyy arvioimaan ohjelman toimivuutta tuorein silmin, kun ei ole itse ollut mukana suunnittelemassa ja toteuttamassa kyseistä ohjelmaa.

Perinteisen mallin kaksi hallitsevaa ohjelmistotestausstrategiaa ovat mustalaatikko- ja lasilaatikkotestaus. Mustalaatikkotestauksessa testien tarkoituksena on yrittää löytää olosuhteita, joissa sovellus ei toimi niin kuin on määritelty. Mustalaatikkotestauksessa ei oteta huomioon sovelluksen sisäistä rakennetta ja käyttäytymistä, vaan ainoastaan testataan ja tarkkaillaan sovelluksen ulkopuolelle näkyvää toimintaa. Tämä tarkoittaa käytännössä sitä, että sovellukseen syötetään suuret määrät erilaisia syötteitä ja varmistetaan, että lopputulos on määrittelyn mukainen. Sovelluksen täydellinen mustalaatikkotestaus on mahdotonta, koska mahdollisten syötteiden määrä lähestyy ääretöntä. (Myers ym. 2004, 9–10.)

Lasilaatikkotestaus on strategia, joka antaa mahdollisuuden tarkastella sovelluksen sisäistä rakennetta. Tässä strategiassa testidata päätetään sovelluksen logiikan perusteella niin, että kaikki eri suorituspolut tulevat testatuiksi. Tämä tarkoittaa sitä, että kaikki eri lopputulosvaihtoehdot ja niihin johtavat ohjelmiston sisäiset polut käydään läpi testitapauksissa. Tekemällä oletuksia ohjelmiston toiminnasta säästetään tarvittavan testauksen määrässä.




Kuvio 1. Ohjelman eri suorituspolkujen testaus.

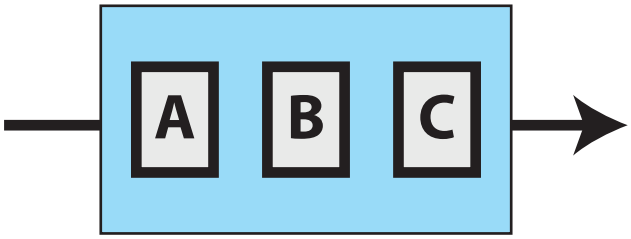
Kuviossa 1 on esimerkkitapaus lasilaatikkotestauksessa käytettävästä koodista. Siinä esitetään pseudokoodina Uutinen-luokka, jossa on metodi `onko_uutinen_tuore?` ja sitä testaava testitapaus. `Onko_uutinen_tuore?` -metodi palauttaa joko arvon tosi tai epätosi riippuen `julkaistu_viikon_sisalla?` -metodin palauttamasta arvosta. Jo metodin nimestä voidaan päätellä, että viikon sisällä julkaistut uutiset palauttavat arvon tosi ja vanhemmat arvon epätosi. Tämän havainnon myötä ei tarvita kuin kaksi testiä testaamaan nämä kaksi erilaista suorituspolkua. Näihin testeihin sisältyy oletus, että ehto toimii oikein ja että mikään muu ohjelman osa ei vaikuta tulokseen. Lisäksi

todellisessa testauksessa lisätään usein testi myös raja-arvolle, eli tässä tapauksessa määritetään mikä on oikea tulos, kun uutinen on julkaistu tasan viikko sitten.

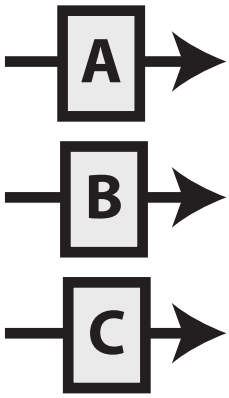
Vaikka ohjelmistotestaus onkin tekninen tehtävä, siihen liittyy myös taloudellisia näkökulmia. Koska täydellisen virheettömyyden todistava ohjelmistotestaus on mahdotonta, sen tavoitteena tulisi olla mahdollisimman monen virheen löytäminen rajallisella määrällä testitapauksia. Tämän tavoitteen saavuttamista helpottaakin lasilaatikkotestauksen suoma mahdollisuus tutustua sovelluksen sisuksiin ja tehdä sen pohjalta kohtuullisia oletuksia siitä mitä eri polkuja ohjelman suorituksessa voi olla.

Kuvio 2 hahmottaa lasi- ja mustalaatikkotestauksen eroa. Ylemmässä kuvassa on esimerkki mustalaatikkotestistä ja alemmassa lasilaatikkotestistä. Mustalaatikkotestissä esitetään ohjelma, jossa on kolme komponettia. Ohjelman sisään ei päästä katsomaan ja ohjelma täytyy testata kokonaisena. Lasilaatikkotestissä voidaan tarkastella ohjelman sisuksia ja testata jokainen komponenteista yksitellen. Jos jokaisella komponentilla on kolme erilaista lopputulosta, jotka tulee testata, mustalaatikkotestejä tarvitaan 3 * 3 * 3 = 27 kattamaan kaikki vaihtoehdot. Lasilaatikkotestauksessa riittää 3 + 3 + 3 = 9 testiä kattamaan samat toiminnot. Kun ohjelman koko kasvaa triviaalista esimerkistä kunnolliseksi satoja toimintoja sisältäväksi ohjelmaksi, vaihtoehtoisten polkujen määrä ja sen myötä mustalaatikkotestaukseen tarvittavien testien määrä nousee niin suureksi, että se ei useimmiten ole enää kannattavaa.

Mustalaatikkotestaus




Lasilaatikkotestaus




Kuvio 2. Musta- ja lasilaatikkotestauksen vertailu.

Perinteisiin testausmenetelmiin voidaan nähdä kuuluvan myös menetelmät, joissa ihmiset käyvät läpi koodia. Myers kutsuu näitä menetelmiä ihmistestausmenetelmiksi. Niitä tulisi käyttää sovelluksen koodaamisen ja sen tietokonepohjaisen testaamisen aloittamisen välissä. Vaikka ihmistestausmenetelmät ovatkin melko epämuodollisia tietokonepohjaiseen testaukseen verrattuna, ne eivät estä onnistunutta testausta, vaan täydentävät ja ovat osallisena testauksen luotettavuudessa ja tuottavuudessa. (Myers ym. 2004, 21–22.)

Ohjelmistokehityksen parhaita käytäntöjä kouluttava Steve McConnell kutsuu näitä samoja tekniikoita yhteistoiminnallisen rakentamisen (Collaborative construction) tekniikoiksi. Tähän piiriin kuuluvat pariohjelmointi, viralliset katsaukset (formal inspections), epävirallisemmat tekniset läpikäynnit, sekä lisäksi muita tekniikoita joissa ohjelmistokehittäjät jakavat vastuun koodin luomisesta. (McConnell 2004, 480.) Riippumatta siitä katsotaanko näiden tekniikoiden kuuluvan ohjelmistotestauksen alle vai, niiden tavoite on sama eli löytää koodista virheitä korjattavaksi, että ohjelman laatua saataisiin parannettua. Niitä kannattaakin käyttää muiden testausmenetelmien kanssa.

Virallisissa katsauksissa ja epävirallisemmissa teknisissä läpikäynneissä joukko ihmisiä käy läpi ohjelman koodia tavoitteenaan löytää virheitä. Ratkaisuja näiden virheiden korjaamiseen ei etsitä itse kokoontumisen aikana. Näitä tekniikoita käyttämällä voidaan löytää 30–70 % logiikka- ja koodausvirheistä, jotka löydetään tyypillisimpien sovelluksien testauksen aikana. (Myers ym. 2004, 22–23.) Yksi syy tähän on ajatus, johon kaikki yhteistoiminnallisen rakentamisen tekniikat pohjautuvat. Kehittäjät ovat sokeita joillekin oman työnsä ongelmakohdille ja jos joku muu käy läpi toisen tekemää työtä, hän voi tarkastella sitä ilman näitä sokeita kohtia ja löytää piileviä virheitä (McConnell 2004, 485).

Katsaukset ja läpikäynti muistuttavat toisiaan melko paljon. Molemmissa materiaalit (suunnitteludokumentit ja ohjelmakoodi) jaetaan osallistuville ihmisille etukäteen, että he voivat tutustua niihin ja itse tapahtuma on ikään kuin aivoriihi, jossa koostetaan osallistujien huomaamista korjattavista virheistä lista koodin kirjoittaneelle ohjelmoijalle. Erona on virallisuuden aste. Katsauksiin kuuluu huolellinen valmistelu, tarkasti määritellyt roolit ja tarkistuslistojen käyttö. Läpikäynniksi taas voidaan kutsua lähes mitä tahansa kehittäjien yhdessä tekemää koodin tutkiskelua. (McConnell 2004, 485–497.)

## 2.1 Automatisoitu ohjelmistotestaus

Kaikki koodi testataan sen kehitysvaiheessa, vaikka sovelluksella ei olisi minkäänlaista testaussuunnitelmaa. Ohjelmoija muuttaa koodia, käännettävissä kielissä kääntää ja sen jälkeen ajaa sen. Seuraavaksi hän testaa, että muutettu ominaisuus toimii ja mahdollisesti, että siihen liitoksissa olevat ominaisuudet toimivat. Näissä testeissä löytyy usein virheitä erityisesti ensimmäisellä kerralla. Ohjelmoija korjaa huomaamansa virheet ja yrittää uudelleen. Tämä sykli jatkuu uudestaan ja uudestaan. Sen sijaan, että kaikki tämä työ tehdään manuaalisesti, se voidaan automatisoida. Kun testit on kerran kirjoitettu, niitä voidaan ajaa pienellä vaivalla. Näin säästyy aikaa ja voidaan varmistaa, että sovellus toimii vähintään ajettujen testien vaikutusalueella niin kuin on haluttu. (Massol 2004, 4.)

Automatisoitu testi on suuremmalla todennäköisyydellä oikein suoritettu verrattuna manuaaliseen testiin. Manuaalisessa testaamisessa tapahtuu yhtä paljon virheitä kuin testattavan ohjelmakoodin luomisessa eli testit suoritetaan epätäydellisesti, minkä

vuoksi osa virheistä saattaa jäädä huomaamatta. Kun testit on automatisoitu, testejä voidaan ajaa useammin, mikä lyhentää virheen syntymisen ja sen löytymisen välistä aikaa, mikä taas johtaa helpompaan korjaamiseen ja vähentyviin korjauskustannuksiin. (McConnell 2004, 528–529.)

Nopeampi virheiden löytyminen auttaa myös pitämään ohjelman toimintavarmana. Kun automatisoitu testi on saatu kirjoitettua, se on ilman ylimääräistä vaivaa kaikkien projektin jäsenten ajettavissa. Näin ohjelmoijat voivat varmistaa omien muutostensa toimivuuden ja eri osasten yhtenäinen integrointi pysyy vaivattomana, kun pienet virheet eivät pääse kumuloitumaan vaikeasti navigoitavaksi ongelmaverkostoksi.

Testauksen automatisoinnin avulla saavutettuihin hyötyihin kuuluu myös mahdollisuus regressiotestaukseen. Regressiolla tarkoitetaan sovelluksiin liittyen taantumista eli sitä, että jokin jo kerran korjattu virhe uusiutuu. Regressiotestaus-termiä käytetään kahteen tarkoitukseen:

1) Testi löytää virheen koodissa. Kun virhe on korjattu, regressiotestit ajetaan uudestaan varmistaen, että kyseinen virhe on todella korjattu.
2) Regressiotestaus on integraatiotestauksen vastakappale. Kun uutta koodia lisätään vanhaan koodiin, regressiotestaus varmistaa, että vanha olemassa oleva koodi toimii lisäyksen jälkeen. Integraatiotestit taas varmistavat, että uusi koodi toimii lisäyksen jälkeen niin kuin pitääkin.

Regressiotestaus siis varmistaa, että olemassa oleva koodi ei pääse rikkoutumaan. Käytännössä tämä tarkoittaa myös sitä, että kaikesta testauskoodista, joka sovellukselle on tehty, tulee osa sen regressiotestausta. (Sisson 2000.) Regressiotestausta voidaan suorittaa manuaalisesti, mutta käytännössä vain automatisoidut testit antavat mahdollisuuden käytännölliseen ja taloudelliseen regressiotestaukseen.

Regressiotestit luovat ohjelmoijille turvaverkon, jonka turvin he voivat luottavaisemmin kehittää ohjelmaa edelleen. Ne luovat myös jatkuvuutta projektille. Uusi tiimin jäsen tai jopa kokonainen tiimi voi regressiotestien avulla tutustua ohjelmaan ja aloittaa pienten muutosten ja niitä seuraavien regressiotestien ajon avulla ohjelman edelleen kehittämisen.

## 2.2 Kehittäjätestaus

Kehittäjätestaus on ohjelmistotestauksen alalaji, jossa ohjelmakoodin kirjoittaja myös testaa sen. Se linkittyy sovelluskehityksessä viimeisen vuosikymmenen aikana levinneeseen ketterän kehityksen (agile development) liikkeeseen ja siinä erityisesti Extreme Programming -menetelmän edustamiin käytäntöihin (Myers ym. 2004, 177–179).

Kehittäjätestaus itsessään ei suoraan paranna ohjelman laatua. Testitulokset ovat indikaattori laadusta, mutta itsessään testit eivät sitä paranna. Jos ohjelman laatua halutaan parantaa, pelkästään testauksen määrän lisääminen ei auta vaan esimerkiksi ohjelmoinnin on parannuttava, jotta ohjelman laatu paranisi. (McConnell 2004, 500–502.)

Epäsuorasti testaus parantaa testattavan ohjelman laatua huomattavasti. Yksi suurimpia kehittäjätestauksen kautta saavutettavissa olevia arkkitehtuuritason parannuksia on parempi vaikutusalueiden eriyttäminen. Tämä on olio-ohjelmoinnin tärkeimpiä ohjenuoria ja tarkoittaa, että eri luokkien ja objektien tulisi vastata vain omasta vastuualueestaan. (Hunt 2007, 143.) Samalla kun koodista tehdään yksinkertaisesti testattavaa, ohjelman arkkitehtuuri muuttuu niin, että sitä on helpompi laajentaa ja ylläpitää. Jos jokin ohjelman osa-alue vaikuttaa hankalalta testattavalta, saattaa olla, että sen vaikutusalueet eivät ole tarpeeksi eriytetyt ja sen arkkitehtuuria tulisi vielä kehittää. Lisäksi testaus parantaa usein sovelluksen sisäistä ohjelmointirajapintaa, koska ohjelmoija joutuu käyttämään omaa koodiaan testeissä ja saattaa sitä kautta saada oivalluksia, miten rajapinta saataisiin helppokäyttöisemmäksi.

McConnell esittää mielenkiintoisen huomion siitä, miten testeissä löytyviä yleisimpiä virheitä voidaan käyttää hyväksi suunnitellessa tulevia tiimin koulutuksia, koodin läpikäyntejä ja ohjelman testitapauksia. Ajatuksena tässä on, että yleensä virheet kasaantuvat johonkin tiettyyn ohjelman monimutkaiseen osaan tai alueelle, jonka toiminnasta tiimillä ei ole täydellistä käsitystä. Virheet voivat olla merkki puutteellisesta osaamisesta ja nämä signaalit kannattaa ottaa huomioon toiminnan kehittämisessä. (McConnell 2004, 502.)

Kehittäjätestauksella on myös rajoitteensa. Kehittäjätestauksen tuottamat testit painottavat yleensä puhtaita testejä. Puhtaat testit testaavat toimiiko koodi, kun taas likaiset testit testaavat eri tapoja, joilla koodi voisi hajota. Tällaisia ovat erilaiset virhetilanteet ja tilanteet, jotka ovat harvinaisia tai yllätyksellisiä. Huolellisesti testatussa sovelluksessa tulisi olla likaisia testejä huomattavasti puhtaita enemmän, koska suunnitellusta poikkeavat tapahtumat ovat yleisempiä kuin kapea suunniteltu alue. (McConnell 2004, 504.) Esimerkiksi lomake voidaan täyttää oikein vain muutamilla tavoilla, mutta vääriä täyttötapoja on runsaasti, ja jotta testaus voisi saada kiinni mahdollisimman paljon virheitä, myös näitä vääriä käyttötapoja tulisi testata.

### 2.2.1 Yksikkötestaus

Kehittäjätestauksessa suuri osa huomiosta kiinnittyy yksikkötestaamiseen. Ne tarjoavat rungon koko sovellukselle ja sen muulle testaamiselle. Yksikkötestit tarkkailevat tietyn yksikön tai yksikköryhmän käyttäytymistä. Useimmiten tämä yksikkö on jonkin olion metodi. Yksikkö voi kuitenkin olla mikä tahansa toiminto, joka ei ole suoraan riippuvainen minkään muun toiminnon suorittamisesta. Yksikkötestien vastakohtana ovat integraatio- ja hyväksymistestit, jotka testaavat eri yksiköiden yhteistoimintaa. (Massol 2004, 6.) Käytännössä yksikkötesti on kappale koodia, joka testaa pientä, määriteltyä osaa toiminnallisuutta testattavassa sovelluksessa. Yleensä tämä tarkoittaa jonkun tietyn metodin testaamista jossain tietyssä kontekstissa. Testi rakentaa tarvittavan tilanteen testille, suorittaa metodin ja varmistaa, että metodi teki mitä sen oli tarkoituskin tehdä. Yksikkötestin ideana on todistaa, että pätkä koodia tekee sitä mitä ohjelmoija kuvittelee sen tekevän. Tämän pohjan päälle voidaan rakentaa muita testejä käyttäen muita testauslajeja. (Hunt 2007, 4.)

Yksikkötestaus vähentää virheiden etsimisen tarvetta ja siihen käytettyä aikaa. Isommassa sovelluksessa eri osa-alueet ovat riippuvaisia monesta muusta osa-alueesta ja virhe alemmalla tasolla saa kaiken sitä koodia käyttävän toiminnallisuuden hajoamaan. Kun tarvittava varmistus koodin toimivuudesta puuttuu, ohjelman kehittämisen edetessä sen kehittämisestä tulee usein todella hidasta, koska muutokset koodissa aiheuttavat virheitä, joiden korjaaminen aiheuttaa uusia virheitä ja niin edelleen. Loppujen lopuksi ongelmat ovat jo niin monimutkaisia, että sovelluksesta on tullut käytännössä mahdoton jatkokehittää. (Hunt 2007, 4–5.)

### 2.2.2 Hyväksymistestaus

Kehittäjätestauksessa toinen tärkeä testausmuoto on hyväksymistestaus. Extreme Programming -menetelmää noudatettaessa sovelluksen suunnitteluvaiheessa keskitytään asiakkaan tarpeiden määrittämiseen ja niitä vastaavien käyttötapausdokumenttien luomiseen. Itse sovellusta ei suunnitella vielä tässä vaiheessa vaan keskitytään vain kokoamaan tehtäviä, joita sovelluksen olisi tarkoitus suorittaa (Myers ym. 2004, 179). Näitä ylös kirjoitettuja käyttötapauksia voidaan käyttää hyväksymistestauksessa. Usein yhtä käyttötapausta kohden tarvitaan useita hyväksymistestejä. Hyväksymistestauksen tarkoitus on määrittää, vastaako sovellus sille määriteltyjä vaatimuksia, joihin kuuluvat esimerkiksi ominaisuudet ja käytettävyys. Hyväksymistestit ovat yhtä arvokkaita kuin sovelluksen sisäistä toimintaa testaavat yksikkötestit, eikä niitä pidä väheksyä. (Myers ym. 2004, 185.)

Hyväksymistestien avulla asiakas tarkistaa, että sovellus on sitä mitä on tilattu. Poikkeavuus odotetuista tuloksista on virhe, joka tulee korjata. Tähän liittyen on tärkeää huomata, että sovellus voi läpäistä kaikki yksikkötestinsä, mutta aiheuttaa virheitä hyväksymistesteissä. Tämä johtuu siitä, että yksikkötestit varmistavat vastaavuutta määrittelyihin, eivät tiettyjen toimintojen olemassaoloa tai esteettisiä asioita. Kaupalliselle sovellukselle "look and feel" -tekijät ovat yksi tärkeä kokonaisuus eikä niitä voida jättää testaamisen ulkopuolelle. (Myers ym. 2004, 185.)

Hyväksymistestit voivat olla manuaalisia tai automaattisia. Manuaalista testausta tarvitaan esimerkiksi silloin, kun asiakas varmistaa, että käyttöliittymä vastaa määrittelyä värien kannalta. Luonnollisesti on kuitenkin viisainta automatisoida kaikki testit, jotka ovat automatisoitavissa. Automatisoitu hyväksymistesti voi näyttää esimerkiksi tältä:

```
Ominaisuus: Uutisen luominen
  Jotta sivustolla olisi uutisia
  reportterien tulisi olla mahdollista lisätä niitä

  Tapaus: Sisäänkirjautunut reportteri luomassa uutista
    Oletetaan että olen sisään kirjautunut reportteri
    Ja että olen uutistenlisäys-sivulla
    Kun täytän "Jymy-uutinen" kohtaan "Otsikko"
```

```
        Ja täytän "Tämä on uutiseni sisältö." kohtaan "Sisältö"
        Ja painan "Tallenna"
        Niin näen tekstin "Uutinen luotu onnistuneesti"

      Tapaus: Sisäänkirjautunut reportteri luomassa virheellistä
uutista
        Oletetaan että olen sisään kirjautunut reportteri
        Ja että olen uutistenlisäys-sivulla
        Kun täytän kohdan "Otsikko" tekstillä ""
        Ja täytän kohdan "Sisältö" tekstillä ""
        Ja painan "Tallenna"
        Niin näen tekstin "Uutista ei voitu luoda puuttuvien tietojen
vuoksi"
```

Testi on kirjoitettu tiettyyn muotoon, jota vasten kirjoitetaan itse testaava koodi. Tekstimuotoisten tarinoiden lisäksi hyväksymistesteinä käytetään usein testejä, joissa annetaan halutut syötteet ja odotetut palautusarvot taulukkomuodossa. Taulukkomuotoista dataa käyttäviä testaustyökaluja on esimerkiksi Fit ja tekstimuotoa esimerkiksi Cucumber. Edellä esitetyn tekstimuotoisen hyväksymistestin suorittava testikoodi esitellään luvussa 3.5.

### 2.2.3 Testivetoinen kehitys

Testivetoinen kehitys (test-driven development) on ohjelmistokehitystekniikka, joka helpottaa ohjelmoijaa tuottamaan kauttaaltaan testattua koodia, jonka rakenne on paras mahdollinen. Sen tavoitteena on koodin hyvä rakenne ja korkea testausaste. Testivetoinen kehitys auttaa ohjelmoijia kehittämään ohjelmia, jotka toimivat kehityksen alusta asti, lisäämään toiminnallisuutta pienissä osissa ja varmistamaan, että ohjelmat toimivat mahdollisimman virheettömästi. (Koskela 2008, 8.) Lisäksi testivetoisessa kehityksessä syntyy jatkuvasti pieniä automatisoituja testejä, joista vähitellen muodostuu tehokas valvontajärjestelmä suojelemaan koodia regressioilta. Testivetoisen kehityksen lyhyt kehityssykli kannustaa laadukkaan koodin tuottamiseen alusta lähtien.

Perinteisesti ohjelmointiprosessi on mennyt niin, että ensin ohjelma on suunniteltu, sitten toteutettu kyseinen suunnitelma ja sitten testattu toteutetun ohjelman toiminta jollain tavalla. Testivetoinen kehitys kääntää järjestyksen päälaelleen ja käskee kirjoittamaan testin ensin ja vasta sen jälkeen kirjoittamaan koodin, jolla saavutetaan tuo testin määrittelemä selkeä tavoite. Rakenne suunnitellaan vasta lopuksi tutkimalla tuotettua koodia ja muokkaamalla sitä mahdollisimman yksinkertaiseen ja asiaansa palvelevaan rakenteeseen. Selvin eroavaisuus perinteiseen ohjelmistotestaamisen verrattuna on testien kirjoittaminen ennen ohjelmakoodia.

Yksinkertaistettuna testivetoisen kehityksen prosessi menee niin, että päätetään, mikä ominaisuus seuraavaksi halutaan toteuttaa ja mietitään, mikä on pienin osa tätä toteutettavaa ominaisuutta. Sitten kirjoitetaan testi, joka todistaa, että tämä pieni osa ominaisuudesta on olemassa ja toimii. Kun testi on kirjoitettu, se ajetaan ja todetaan, että se ei mene läpi. Sen jälkeen kirjoitetaan pieni määrä koodia, jolla juuri kirjoitettu testi saadaan menemään läpi. Tämän jälkeen refaktoroidaan kirjoitettu koodi eli muutetaan sen sisäistä rakennetta selkeämmäksi muuttamatta kuitenkaan sen toimintaa ulospäin. Refaktoroinnin tarkoituksena on pitää ohjelman koodi aina selkeänä ja sen nykyistä käyttötarkoitusta vastaavana. Refaktoroinnin jälkeen sykli aloitetaan alusta päättämällä, mitä ominaisuutta aletaan seuraavaksi toteuttaa. (Koskela 2008, 8.) Testivetoista kehitystä käytetään yleensä pääasiassa yksikkötestauksen tapaan, mutta sitä voidaan käyttää myös hyväksymistestaukseen. Prosessi toimii muuten samoin, mutta testit testaavat suurempia yksiköitä, kuten kokonaisia ominaisuuksia.

Sovelluskehittäjä Noel Rappin listaa artikkelissaan omat syynsä, miksi hän käyttää testivetoista kehitystä. Koodi, jota syntyy testivetoista kehitystä käyttämällä, on parempaa. Metodit ja luokat ovat koodimäärältään pienempiä, vaikutusalueet on eriytetty ja koodi sisältää vähän toistoa. Pienet metodit ovat helpompia testata ja näin testien kirjoittaminen ensin ajaa koodia hyvään rakenteeseen. (Rappin 2009.)

Testin kirjoittaminen vie luonnollisesti aluksi jonkin verran aikaa, mutta kun se on kerran kirjoitettu, se voidaan ajaa nopeasti uudelleen ja uudelleen. Näin voidaan varmistua siitä, että kaikki toimii ilman että ohjelmoijan tulisi itse manuaalisesti ja aikaa vievästi testata samat asiat uudelleen ja uudelleen. Kun ominaisuuteen palataan myöhemmin, testit ovat yhä tallella.

Vaikka testivetoinen kehitys ei estäkään kaikkien virheiden syntymistä, Rappinin kokemuksen mukaan se helpottaa niiden etsintää ja korjaamista. Tämä siitä syystä, että rakenne on modulaarisempi[1] ja sen myötä virheet vaikuttavat suppeampiin alueisiin. (Rappin 2009.)

Käyttäytymisvetoinen kehitys (behaviour-driven development) on testivetoisen kehityksen prosessista muokkautunut ja siinä havaittujen puutteiden korjaamiseen pyrkivä sovelluskehitysprosessi. (Behaviour-Driven Development - Introduction 2008). Esseessään *A New Look at Test-Driven Development* Dave Astels esitteli käyttäytymisvetoisen kehityksen ja sen eron testivetoiseen kehitykseen. Tekniseltä kannalta eroa ei juurikaan ole. Testivetoinen kehitys syntyi ohjelmistotestaamisesta, mutta Astellsin mielestä sen ydin on ohjelman halutun toiminnan määrittelyä, ei sen toimivuuden testaamista. Testivetoisen kehityksen avulla tajutaan mitä ollaan rakentamassa ennen kuin aletaan rakentaa sitä. Testivetoisen kehityksen myötä syntyvät testitapaukset, joita voidaan käyttää regressiotestaukseen, ovat vain tästä toiminnasta syntyvä hyödyllinen lisä. Testivetoisen kehityksen testauspainotteinen sanasto ja ajattelumaailma jättää suurimman hyödyn testaamisen varjoon ja tämän vuoksi Astels ehdotti tiettyjä muutoksia testivetoisen kehityksen terminologiaan ja työkalujen toteutukseen. Lopputulosta hän kutsuu käyttäytymisvetoiseksi kehitykseksi. Ajatuksena on siis vain selkeyttää ja nostaa testivetoisen kehityksen ydin esille. (Astels 2006.)

### 2.2.4 Työkalut

Automatisoituun yksikkö- ja hyväksymistestaamiseen on olemassa apuvälineitä; testaamiseen käytettäviä testauskirjastoja, joiden tarkoitus on helpottaa testaamista. Niiden avulla testit ovat helpompia kirjoittaa. Ne tarjoavat erilaisia oikoteitä ja testauksessa hyväksi havaittuja lisäominaisuuksia ja tekevät testien toistuvan ajamisen helpoksi. (Massol 2004, 16.) Ohjelmistotestaaminen on tietysti mahdollista myös ilman valmiiden kirjastojen käyttämistä, mutta koska ne nopeuttavat testaamista huomattavasti ja tarjoavat valmiiksi mietittyjä ratkaisuja helpottaen testaamisen aloittamista, niitä kannattaa käyttää. Testauskirjastoja on saatavilla kaikille merkittäville

1 Ohjelmoinnissa modulaarisia ovat ohjelmat, jotka on jaettu useampaan pienempään osaan (moduuliin). Hyvin Suunniteltu modulaarinen ohjelma koostuu siis useasta moduulista, joita voidaan vaihtaa ja muokata erilliseän muista moduuleista. Modulaarisuus lisää ohjelmien luotettavuutta, muokattavuutta ja testattavuutta.

ohjelmointikielille. Useimmille ohjelmointikielille on jopa monia vaihtoehtoja, joista valita.

Kaikkien yksikkötestauskirjastojen tulisi noudattaa seuraavia käytäntöjä:

- Jokainen yksikkötesti tulee suorittaa erillään muista yksikkötesteistä.
- Virheet tulee havaita ja raportoida testikohtaisesti.
- Ajettavien yksikkötestien määrittelemisen tulee olla helppoa. (Massol 2004, 10.)

Tyypillisin yksikkötestaus-kirjastotyyppi on xUnit-kirjastot, jotka noudattavat kaikkia yksikkötestauskirjastojen käytäntöjä. Nimi tulee Smalltalk-ohjelmointikielelle tehdystä SUnit- ja sitä seuranneesta Java-ohjelmointikielelle tehdystä JUnit-kirjastosta. Näiden kirjastojen suosion myötä xUnit-mallin mukainen testauskirjasto on kehitetty käytännössä kaikille ohjelmointikielille ja sitä voidaan pitää yksikkötestauksen standardina. Wreckamovie.com-projektissa käytetään xUnit-standardista poikkeavaa RSpec-testauskirjastoa, josta kerrotaan tarkemmin Wreckamovie.com-osion Testausstrategia-luvussa.

Yksikkö- ja hyväksymistestauksen lisäksi tarjolla on työkaluja muihinkin ohjelmistotestauksen tehtäviin. Saatavilla on esimerkiksi testikattavuusanalysaattoreita testien kattavuuden arviointiin ja jatkuvan integroinnin (continuos integration) mahdollistavia testiservereitä, jotka helpottavat automatisoidun regressiotestauksen suorittamista. (Haikala & Märijärvi 2004. 279.)

## 2.3 Verkkosovellusten ohjelmistotestaus

Verkkosovelluksella tarkoitan sovellusta, joka on käytettävissä internetselaimen kautta. Käytän termiä laveasti tarkoittamaan kaikkia toiminnallisuutta sisältäviä verkkosivuja, kuten wikeja, blogeja ja verkkokauppoja. Verkkosovellukset koostuvat useimmiten useista erilaisista servereistä, joista jokainen hoitaa tiettyä tehtävää. Esimerkiksi saatetaan käyttää arkkitehtuuria, jossa on webpalvelin hoitamassa HTTP-kutsujen käsittelyä, sovelluspalvelin hoitamassa dynaamisten kutsujen suorittamista ja tietokantaserveri huolehtimassa tiedon tallentamisesta. Aiheen laajuuden vuoksi käsittelen tässä vain sovelluspalvelimen ohjelmistotestaamista, koska se on taso, jolla

useimmiten kehitystyö tapahtuu ja siksi suurin osa ohjelmistotestaamisesta keskittyy siihen.

Myers toteaa, että verkkosovellusten ohjelmistotestaaminen ei eroa muiden sovellusten ohjelmistotestaamisesta millään tapaa (Myers ym. 2004, 193). Hän puhuu kuitenkin itse prosessista ja tavoitteesta löytää mahdollisimman paljon virheitä ennen sovelluksen julkistamista käyttäjille. Kaikenlaisista sovelluksista etsitään virheitä samalla tavalla, mutta testaamisen käytäntö esimerkiksi työpöytäsovelluksen ja verkkosovelluksen välillä on silti erilaista.

Verkkosovelluksessa yksi interaktiosessio on hyvin lyhyt. Käyttäjä menee selaimessa internet-osoitteeseen. Kutsu käsitellään palvelimella ja luodaan HTML-muotoinen vastaus, joka lähetetään käyttäjälle. Sessiot ja evästeet antavat palvelimelle mahdollisuuden yhdistää useita kutsuja toisiinsa, niin että esimerkiksi sisäänkirjautuminen säilyy kutsujen välillä. Käytännössä testaamisessa päästään hyvin pitkälle pelkästään yhtä kutsu-prosessointi-vastaus-sykliä testaamalla. Jokaisen syklin välillä sovellus nollaa itsensä oletustilaan, josta taas lähdetään liikkeelle, kun uusi kutsu saapuu. Työpöytäsovelluksissa käyttökerta voi kestää vaikkapa tunnin. Esimerkiksi tekstinkäsittelyohjelmassa on runsaasti erilaisia ohjelman tilaan liittyviä tietoja, jotka muuttuvat jatkuvasti, kun käyttäjä käyttää ohjelmaa. Tähän sisältyy suurempi mahdollisuus virheisiin, kun erilaisia polkuja on monia ja asiat vaikuttavat ristiin toisensa kanssa.

Verkkosovellus on interaktiossa käyttäjän kanssa HTML:n välityksellä. HTML:n sisältämien elementtien hallinnointi ja testaus on helppoa verrattuna työpöytäsovelluksen ohjelmistotestaamiseen, jossa joudutaan rakentamaan monimutkaisia testausjärjestelmiä, joilla simuloidaan käyttäjän toimintaa. Tosin nykyisin selaimet ovat jo niin tehokkaita, että myös selaimen sisälle pystytään JavaScriptin avulla luomaan työpöytäsovelluksia vastaavia sovelluksia, joiden ohjelmistotestaaminen onkin lähempänä työpöytäsovelluksien testaamista ongelmineen.

# 3 WRECKAMOVIE.COM-VERKKOPALVELU

Wreckamovie.com on verkkopalvelu, joka mahdollistaa eri tyyppisten elokuvien tekemisen yhteisöllisesti suuren ihmismäärän yhteisellä työpanoksella. Palvelun ytimessä ovat elokuvatuotannot, joita käyttäjät voivat perustaa. Tuotannon perustanut käyttäjä voi julkaista tuotantoonsa liittyviä tehtäviä, joiden suorittamiseen muut käyttäjät voivat eri tavoin osallistua. Tehtävien lopputulokset voivat olla mitä tahansa ideoista ja käsikirjoitusten osasista 3D-malleihin ja musiikkikappaleiden remix-versioihin. Palvelun tavoitteena on tarjota käyttäjille mahdollisuus toteuttaa itseään ja elokuvien tuottajille mahdollisuus luoda ammattilaistason elokuvia hyödyntämällä internetin ihmisiä yhdistävää vaikutusta. Lisäksi palvelussa on erilaisia "social networking" -ominaisuuksia vahvistamassa palvelun yhteisöllisyyttä ja tukemassa elokuvien tekoa.

Idea palveluun syntyi vuonna 2005 julkaistun elokuvan *Star Wreck: In the Pirkinning* kehitysprosessista. Harrastelijavoimin tehdyn elokuvan mahdollisti suuri avustajajoukko, johon pidettiin yhteyttä internetin välityksellä. Elokuva julkaistiin ilmaiseksi internetissä ja aktiivisen avustaja- ja faniyhteisön avustuksella se saavutti alle kuukaudessa yli kolmen miljoonan latauksen rajan (Star Wreck - In the Pirkinning - Distribution).

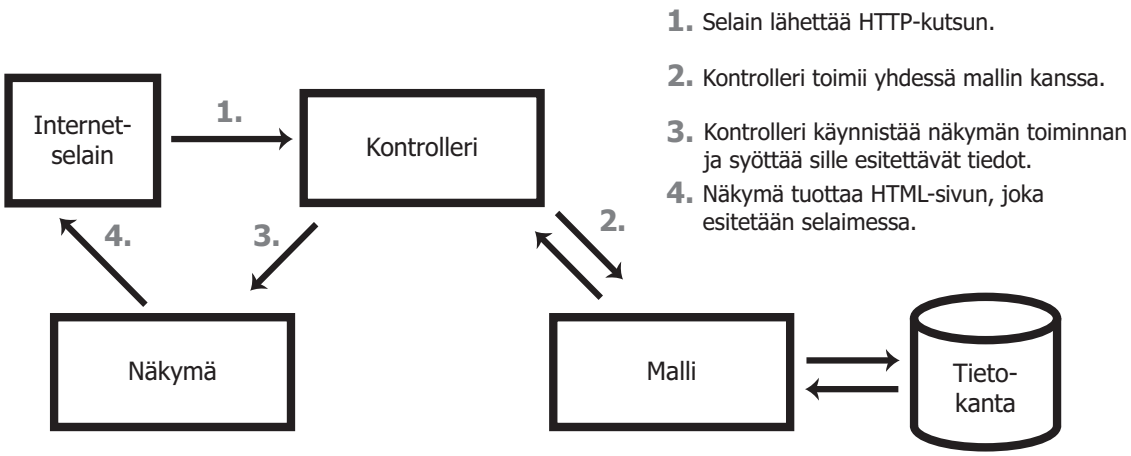
Palvelun avoimen hengen mukaisesti lähes kaikki sen sisältämät tiedot ovat luettavissa ilman käyttäjän rekisteröitymistä. Tietoa lisätäkseen tai muokatakseen sekä käyttäjäkohtaisten ominaisuuksien käyttöä varten tulee palveluun kuitenkin rekisteröityä.

## 3.1 Teknologia

Tekniseltä toteutukseltaan Wreckamovie on perinteinen tietokantaa tietovarastonaan käyttävä verkkosovellus. Se on toteutettu käyttämällä avoimen lähdekoodin Ruby on Rails -verkkosovellusohjelmistokehystä. Ruby on Rails on Ruby-ohjelmointikielellä toteutettu ohjelmistokehys verkkosovellusten kehittämiseen. Kirjaston tavoitteena on tarjota verkkosovellusten rakentamiseen täydelliset työkalut, jotka ovat yksinkertaisia ja tehokkaita.

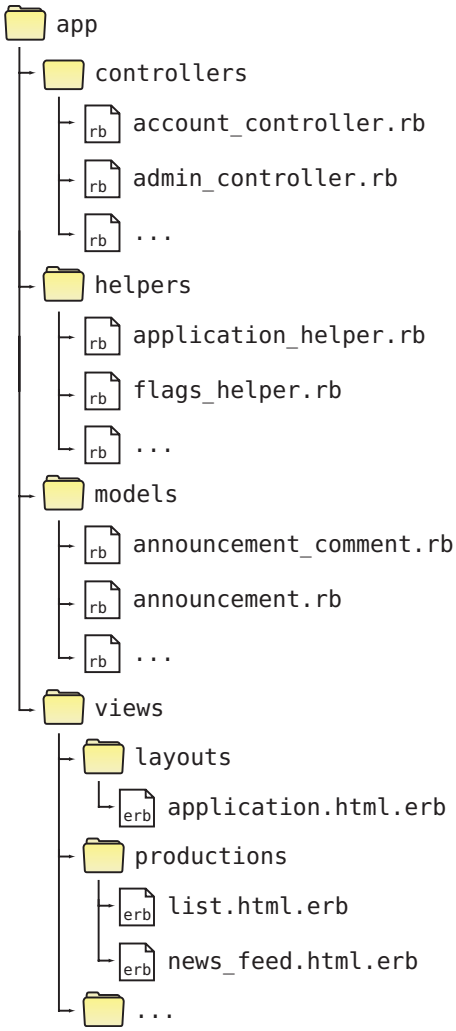
Kuten suuri osa nykyaikaisista verkkosovellusohjelmistokehyksistä, Ruby on Rails käyttää Model-View-Controller -arkkitehtuuria (MVC-arkkitehtuuri) ohjelmoinnin organisointiin. MVC-arkkitehtuuri on vuonna 1979 julkistettu interaktiivisten sovellusten kehitykseen tarkoitettu arkkitehtuurimalli. Sen tavoitteena on erottaa sovelluksen toimintalogiikka sen ulkoasusta niin, että molempia voidaan muokata vaikuttamatta toiseen. MVC-arkkitehtuurissa sovellus jaetaan nimensä mukaisesti kolmeen palaan, jotka ovat model, view ja controller, joista käytän suomennoksia malli, näkymä ja kontrolleri. Malli edustaa sovelluksen dataa ja tilaa sekä niihin liittyviä sääntöjä. Näkymä vastaa ohjelman osasta, joka on kosketuksissa käyttäjään eli käyttöliittymästä. Kontolleri hoitaa näiden kahden välistä kommunikaatiota ja manipulointia. Se ottaa vastaan tapahtumia sovelluksen ulkopuolelta (esimerkiksi käyttäjän käyttöliittymän kautta suorittamia toimintoja), toimii yhteistyössä mallien kanssa ja muuttaa näkymää, joka luo käyttäjälle esitettävän käyttöliittymän. (Thomas ym. 2006, 11–12.)

Kuviossa 3 esitetään MVC-arkkitehtuuria noudattavan verkkosovelluksen yhden selainkutsun käsittelyprosessi. Selain lähettää kutsun palvelimelle, joka ohjataan kontrolleriin. Kontrolleri on yhteydessä malleihin, jotka noutavat tietoa tietokannasta ja muokkaavat sitä. Kun tarvittavat toiminnot on tehty ja käyttäjälle esitettävä data on koottu, kontrolleri antaa datan eteenpäin näkymälle. Näkymä huolehtii HTML-sivun luomisesta, joka esitetään käyttäjän selaimessa.




Kuvio 3. MVC-arkkitehtuuri

Ruby on Rails vie MVC-arkkitehtuurin hyvin pitkälle ja tarjoaa vakiopaikat ja -toimintamallit arkkitehtuurin kolmelle osalle (Thomas ym. 2006, 1). Tällä tarkoitetaan sitä, että MVC-arkkitehtuurin eri kerrosten koodi jaotellaan erillisiin kansioihin tietyn käytännön mukaisesti. Kuviossa 4 näkyy perinteinen Rails-projektin rakenne. Controllers-, models- ja views-kansio sisältävät nimensä mukaiset MVC:n osat. Helpers-kansio sisältää tiedostot, jotka sisältävät näkymien käyttämiä apuluokkia, joiden metodit auttavat esimerkiksi monimutkaisen HTML:n muodostamisessa. Views-kansio on jaoteltu kontrollereiden mukaisesti alakansioihin niin, että jokaisen kontrollerin käyttämät näkymätiedostot ovat yhdessä kansiossa. Layouts-kansio sisältää useiden näkymien käyttämiä sapluunoja, joilla voidaan toteuttaa esimerkiksi yhtenäisiä, useille sivuilla näkyviä peruselementtejä.




Kuvio 4. Rails-kansiorakenne

## 3.2 Testaustapa

Wreckamovie.com-verkkopalvelu on ohjelmistotestattu kehittäjätestauksella. Koodin kirjoittaja on itse huolehtinut, että koodi tulee testatuksi. Testausta ei ole harjoitettu aivan projektin alusta asti, mutta koodimäärän kasvaessa on havaittu tarpeelliseksi luoda testejä, että voidaan varmistaa toimivuus uusia ominaisuuksia lisätessä ja palvelun käyttämiä ulkopuolisten tekemiä kirjastoja päivitettäessä. Kaikki testit ovat lasilaatikkotestejä. Käytetyt testityypit ovat yksikkö-, hyväksymis- ja regressiotestit sekä funktionaaliset testit. Testityypit ovat valikoituneet resurssien ja koetun tarpeen mukaisesti.

Mitä pidemmälle testauksessa on edetty sitä enemmän se on muuttunut niin, että testit kirjoitetaan ennen koodia. Kun vanhaan koodiin lisätään testejä jälkeenpäin, tämä ei tietenkään ole mahdollista, mutta uutta luodessa olen huomannut ”testit ensin” -lähestymistavan helpottavan testaamista. Testit tulevat tehdyiksi ja niiden tekeminen ei tunnu niin puuduttavalta, kun ne kirjoittaa ennen koodia. Lisäksi saa avukseen muut testivetoisen kehityksen edut kuten sen, että virheet löytyvät mahdollisimman nopeasti säästäen niiden korjaamiseen käytettävää aikaa.

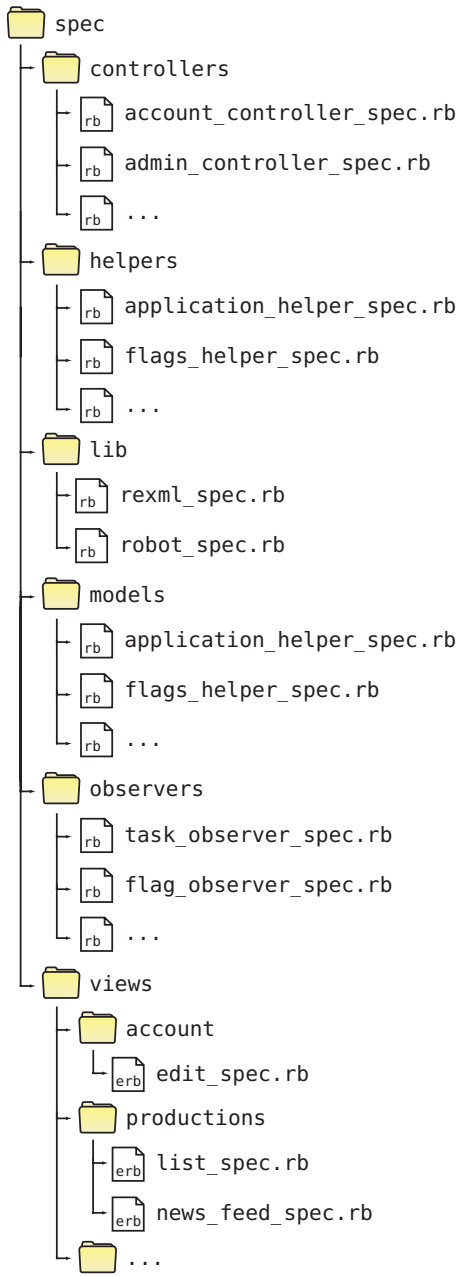
## 3.3 Testausstrategia

Yksikkötestit ja funktionaaliset testit on toteutettu käyttämällä RSpec-testauskirjastoa. Hyväksymistesteissä on RSpec-kirjaston lisäksi käytetty Cucumber- ja Webrat-kirjastoja, jotka tarjoavat apuvälineitä erityisesti kyseiselle testaustyypille.

Rspec on Ruby-ohjelmointikielelle saatavissa oleva testauskirjasto. Se vastaa toiminnallisuuksiltaan hyvin pitkälti xUnit-kirjastojen luomia käytäntöjä, mutta on syntaksiltaan erilainen. Syntaksimuutoksen taustalla on halu mukautua käyttäytymisvetoisen kehityksen prosessiin. Valitsin RSpecin projektin testauskirjastoksi juuri sen selkeän ja tarkoituksenmukaisen syntaksin vuoksi. Wreckamovie.com-projektin aloittamisen jälkeen monia RSpecin hyviä puolia on lisätty erilaisiin xUnit-tyyppisen Test::Unit-kirjaston laajennuksiin. Tästä syystä saattaisin valita nyt aloittaessani RSpecin sijaan Test::Unitin laajennettuna Shoulda-kirjastolla. Tähän pääasiallisena syynä on RSpecin monimutkaisuus, joka saattaa aiheuttaa

ongelmatilanteita esimerkiksi muita kirjastoja päivitettäessä. Itselläni ei kuitenkaan ole ollut suurempia ongelmia ja olen tyytyväinen testikirjastovalintaani.

Seuraavaksi esittelen esimerkin omaisesti, miten Wreckamovie.com:n ohjelmistotestaus on toteutettu. Koska sovellus on jaettu MVC-arkkitehtuurin mukaisesti, on luonnollista että testaaminenkin hoidetaan käyttämällä samaa jakoa. Näin testaus jakautuu eri alueille niin, että ne testataan eristyksissä toisistaan, mikä vähentää testien ja sovelluskoodin riippuvuutta toisistaan.




Kuvio 5. Testien kansiorakenne

Testit on siis jaoteltu niin, että ne testaavat eri alueita melko eristyksessä toisistaan. Testien kansiorakenne, joka on näkyvissä kuviossa 5, vastaa hyvin pitkälti Rails-sovelluksen kansiorakennetta. Mallitestit testaavat malleja, näkymätestit näkymiä ja kontrolleritestit kontrollereita. Lisäksi on olemassa testejä helpereille, observereille ja yleisille lib-kansioon sijoitetuille tiedostoille.

Mallitestit ovat testauksen selkäranka, jolla testataan mallien toimintalogiikka. Ne on jaettu niin, että yhtä mallia koskevat testit ovat yhdessä tiedostossa. Liite 1 on ProductionNewsItem-mallin testitiedosto. Rakenne on kaikissa hyvin samanlainen. Tämä ei ole testauskirjaston pakottamaa, vaan projektissa käyttöön ottamaani yhtenäistä testaustyyliä, mikä helpottaa asioiden löytämistä ja tekee testien lisäämisestä ja muokkaamisesta helppoa.

Ensiksi tulee määrittely, joka kertoo millä datalla voidaan luoda uusi objekti kyseisestä luokasta. Seuraavaksi testataan, että kaikki luokan assosiaatiot on määritelty tarkoituksenmukaisesti. Testeissä kysytään luokalta, että tunteeko se tietyn tyyppisiä ja nimisiä assosiaatioita. Jos halutut assosiaatiot löytyvät, niiden määrittelyn kautta saatavien toiminnallisuuksien uskotaan toimivan, koska luokkien väliset assosiaatiot ovat Ruby on Railsin mukana tuleva ominaisuus ja Ruby on Rails sisältää testit omille ominaisuuksilleen. Ohjelmoija voikin keskittyä vain itse luomansa toiminnallisuuden testaamiseen. Seuraavaksi testataan validaatiot tiedoston alussa määriteltyä referenssidataa vasten. Sen jälkeen testataan kaikki luokan määrittelemät julkiset metodit.

Kontrolleritestit on toteutettu ajan säästön vuoksi niin, että ne testaavat koko järjestelmää. Ne toimivat siis ikään kuin integraatiotesteinä, joiden painopiste on järjestelmän eri osien välisten rajapintojen toimivuuden varmistamisessa (Haikala & Märijärvi 2004. 290). Tämä tarkoittaa, että kaikki käytetty tieto tallennetaan tietokantaan ja ladataan sieltä, mitään rajapintoja ei korvata tynkämoduleilla[2] ja näkymien HTML tuotetaan loppuun saakka. Nämä testit toimivat siis funktionaalisina testeinä ja varmistavat, että koko pakka toimii niin kuin on tarkoitus. Olen päätynyt tähän ratkaisuun, koska kontrollerit ovat yleensä melko yksinkertaisia ja sen vuoksi niissä ei ole juurikaan paikkoja, joihin virheet voisivat jäädä piiloon. Erillisten

2 Tynkämoduulien käytöllä tarkoitetaan jonkin moduulin korvaamista tynkämoduulilla, joka säilyttää korvatun moduulin rajapinnan, mutta muuttaa sen ajaman koodin joksikin toiseksi. Tätä käytetään testeissä eriyttämään testattava luokka täysin toisista luokista.

eristettyjen kontrolleritestien ja funktionaalisten testien kirjoittaminen ja ylläpito olisi niin suuri vaiva, että mielestäni se ei ole kannattavaa projektissamme. Jos kontrollerissa on virhe, testit saavat sen kiinni ja sen yksinkertaisuuden vuoksi virhe on helppo paikantaa, vaikka kontrollerille ei olekaan muista MVC-arkkitehtuurin osasista eristettyä testiä.

Liitteenä 2 on ToDoController-luokan testitiedosto. Kontrolleritestit seuraavat tiettyä kaavaa, missä järjestyksessä asiat testataan. Ensiksi tulee routes-testaus, jos se poikkeaa käytännöistä. Siinä testataan että Railsin sisäinen URL-kartoitus liittää oikean URL:n oikeaan kontrollerin toimintoon (action). Sen jälkeen testataan jokainen toiminto läpi, haarautuen aina eri skenaarioihin. Esimerkiksi rekisteröityneen tai rekisteröimättömän käyttäjän tekemä kutsu ja filmiproduktion luominen onnistui tai epäonnistui tuottavat eri tuloksen ja sen vuoksi ne testataan erillisissä testeissä. Testeissä varmistetaan, että toiminnon kutsuminen tekee ne muutokset, mitkä pitääkin ja että lopuksi käyttäjälle lähetetään sellaista HTML:aa kuin pitää.

Helper-, observer- ja lib-testit ovat yksinkertaisia, kuten testattavat moduulinsakin. Helpereitä käytetään näkymien tukena ja ne toimivat useimmiten niin, että niille syötetään dataa ja ne palauttavat HTML:aa. Testeissä testataan erilaisten syötteiden avulla, että apumetodit toimivat niin kuin halutaan.

Observerit ovat erityisiä malleihin liitettyjä tarkkailijoita, joilla on Wreckamovie.com-palvelussa toteutettu käyttäjäkohtainen tapahtumavirta, joka kertoo mitä kaikkea käyttäjälle merkityksellistä järjestelmässä on tapahtunut. Esimerkiksi kun käyttäjän kommenttiin vastataan, observer huomaa tämän kommentin luonnin jälkeen ja luo käyttäjän tapahtumavirtaan ilmoituksen tästä. Testit ovat melko yksinkertaisia ja niillä testataan, että tietyillä syötteillä luodaan halutut ilmoitukset.

Lib-testit testaavat koodia, joka ei kuulu MVC-arkkitehtuurin sisälle ja sen vuoksi on sijoitettu sen ulkopuolelle. Wreckamovie.com-palvelun tapauksessa tämä tarkoittaa erilaisia automaattisella ajastuksella ajettavia huolto- ja päivitystoimenpiteitä. Kyseinen Robot-moduuli on aivan perinteinen luokka ja sen testit toimivat normaaliin yksikkötestaustapaan.

## 3.4 Testauskäytäntöjen analyysi

Analysoin tässä luvussa Wreckamovie.com-palvelun testauskäytäntöjä. Palvelun rakenteen ja testien jaottelun pohjalta olen jaotellut tarkastelun MVC-arkkitehtuurin osa-alueiden mallin, näkymän ja kontrollerin välille. Käyn läpi testauskäytäntöjen heikkoudet, vahvuudet ja niiden pohjalta parannusehdotuksia. Vahvuudet ja heikkoudet on tiivistetty taulukkoon 1.

Mallit sisältävät suuren osan Wreckamovie.com-palvelun toimintalogiikasta ja niiden testit toimivat testauksen selkärankana. Ne varmistavat, että palvelun toimintalogiikka toimii. Testien rakenne on yhdenmukainen, mikä tekee testien navigoimisesta helppoa ja varmistaa, että kaikki mallien toiminnallisuudet tulevat testattua. Mallitestit kannustavat jakamaan toiminnallisuuden pieniin helposti testattaviin metodeihin, mikä on olio-ohjelmoinnin periaatteiden mukaista.

Näkymätestien pääasiallinen tarkoitus tässä projektissa on varmistaa, että tuotetussa HTML:ssa on JavaScriptin tarvitsemat CSS-luokat (class) ja -tunnisteet (id). Näkymän tuottamaa HTML:aa analysoidaan vain tekstimuotoisena toisin kuin selaimen sisällä ajettavissa testausratkaisuissa, mikä tekee testien ajamisesta todella paljon nopeampaa. Lisäksi näkymätestit ajetaan eriytettyinä kontrollereista, mikä tekee testien ajamisesta myös hieman nopeampaa.

Selaimessa ajamattomuus on myös näkymätestien heikkous. JavaScriptia ei ajeta missään vaiheessa, ja näin sen vaikutukset jäävät kokonaan testien ulkopuolelle.

Kontrolleritestit toimivat samalla myös funktionaalisina testeinä. Ne varmistavat, että ohjelma toimii alun HTTP-kutsusta lopun selaimelle palautettuun HTML:n. Kontrolleritestit varmistavat myös koko MVC-arkkitehtuurin toimivuuden yhdessä niin, että jos niiden rajapinnat eivät ole toistensa kanssa yhteensopivia testi aiheuttaa virheen. Kontrolleritestien ongelma on yhtenäistämisen puute. Ohjelman kehityksen aikana näkemys kontrolleritestien roolista on muuttunut ja sen vuoksi eri kontrollereita testataan hieman eri tavalla.

Kontrollereiden testaaminen on myös useaan otteeseen paljastanut koodia, joka on ollut rakenteeltaan MVC-arkkitehtuurin vastaista. Toimintalogiikka tulisi olla aina

eriytettynä malli-tasolle niin, että kukin luokka vastaa omasta toiminnastaan. Välillä tapahtuu kuitenkin niin, että logiikkaa eksyy kontrollereiden puolelle. Malleja on helpompi testata kuin kontrollereita, koska niitä testatessa ei tarvitse pitää huolta HTTP-kutsun, sessioiden ja evästeiden simuloimisesta niin, että järjestelmän tila on oikea testattavalle tilanteelle. Kun kontrollereihin on eksynyt toimintalogiikkaa, joka ei sinne kuulu, yksittäisen kutsun hoitavasta koodista on tullut todella pitkä, mikä on hankaloittanut testausta. Testauksessa esille nousseet ongelmat ovat havahduttaneet ohjelmoijan vallitsevaan tilanteeseen ja kannustaneet refaktoroimaan rakenteen helpommaksi testata ja sitä kautta myös selkeämmäksi.

Taulukko 1. Testauskäytäntöjen vahvuudet ja heikkoudet

| | **Malli** | **Näkymä** | **Kontrolleri** |
|---|---|---|---|
| Vahvuudet | • Testauksen selkäranka<br>• Auttaa eriytetyn logiikan ylläpitämisessä<br>• Testit pieniä ja yksinkertaisia | • Paljastaa JavaScriptille tarpeellisten HTML-osien luomisen<br>• Nopeampi suorittaa kuin selaimen sisällä ajettavat työkalut kuten Selenium. | • Eri osasten yhteistoiminnan testaus varmistaa että koko paketti toimii |
| Heikkoudet | | • JavaScript-koodia ei testata, joten ongelmat voivat jäädä piileviksi<br>• Testit vain tulkitsevat HTML:aa, selaimen sisällä ei testata | • Eri kontrollereiden välinen epäyhteneväisyys |

## 3.5 Parannusehdotukset

Näkymätasolla hyvä lisäys olisi selaimessa tapahtuvan toiminnallisuuden eli JavaScript-koodin yksikkötestaus. Tämä on ollut suunnitelmissa jo pidemmän aikaa, mutta en ole pystynyt täysin perustelemaan itselleni sen aloittamista. Syitä on useita. Testaus on JavaScript-maailmassa vielä melko uutta, minkä vuoksi sille ei ole olemassa laajalle levinneitä vakiintuneita testaustyökaluja eikä käytäntöjä. Tämä tekee aloittamisesta melko työlästä, koska aluksi tulisi vertailla eri kirjastoja ja päättää, minkä ottaa käyttöön. Tämän jälkeen käyttö pitäisi opetella pitkälti yrityksen ja erehdyksen kautta,

koska opetusmateriaali on vielä suppeaa. Lisäksi Wreckamovie.com-palvelu on JavaScript-toiminnallisuuksiltaan melko yksinkertainen, mikä vähentää virheiden mahdollisuutta. Lisäksi JavaScriptin testaaminen olisi siitä erityistä, että se olisi ainoa testausmuoto, jolla testattaisiin muuta kuin Ruby-koodia. Tämän vuoksi integrointi muihin testien ajotyökaluihin voisi olla työlästä.

Toisaalta JavaScript-koodin testaaminen olisi myös hyvin tärkeää. Palvelimella suoritettavassa koodissa tapahtuvat virheet on mahdollista saada palvelun ylläpitäjien tietoon pienellä vaivalla. Wreckamovie.com:ssa palvelimella tapahtuneet virheet saa näkyviin admin-alueen virhelokin kautta, josta saadaan näkyviin kaikki tiedot virheen aiheuttaneesta HTTP-kutsusta ja sen tehneestä selaimesta. JavaScript suoritetaan käyttäjän selaimessa ja siinä tapahtuneet virheet eivät tule mitenkään palvelun ylläpitäjien tietoon, minkä vuoksi JavaScript-virheiden välttäminen olisi tärkeämpää kuin palvelinpuolen virheiden.

Yleisiä käytännöllisiä parannuskohteita Wreckamovie.com:n testauksessa olisi ainakin tiettyjen useasti toistuvien testityyppien ja skenaarioiden paketoiminen makroiksi. Tämän voisi tehdä joko itse tai käyttää muiden jakoon laittamia makroja. Esimerkiksi mallien tapauksessa Rails tarjoaa aputoiminnon, joka varmistaa että olion attribuutti voi olla vain numero:

```
class Person < ActiveRecord::Base
  validates_numericality_of :age
end
```

Tämän iän numeroksi määrittämisen testaamiseksi tulisi käydä läpi vaiheet:

Positiivinen tapaus:

1. Luodaan uusi olio Person-luokasta.
2. Asetetaan sen iäksi merkkijono.
3. Tarkastetaan, että olio on epävalidi.

Negatiivinen tapaus:

1. Luodaan uusi olio Person-luokasta.
2. Asetetaan sen iäksi numero.
3. Tarkastetaan, että olio on validi.

Nämä vaiheet voidaan korvata käyttämällä Shoulda-kirjaston tarjoamaa makroa:

```
should_only_allow_numeric_values_for :age
```

Tämä makro varmistaa samat asiat, kuin mitä aiemmin esitetyt vaiheet. Sen käyttäminen on vain paljon helpompaa ja nopeampaa.

Erilaisten skenaarioiden paketoiminen tarkoittaisi, että kontrollereissa toistuvien tapahtumien testit laitettaisiin jaettavaan testiin, jota sen jälkeen käytettäisiin kaikissa tapauksissa, joissa testattava tapahtuma toistuu. Nykyisin tätä käytetään esimerkiksi rekisteröityneille käyttäjille tarkoitettujen alueiden rajaamisen varmistamiseen. Kun käyttäjä menee sivulle, joka on tarkoitettu vain rekisteröityneille käyttäjille, hänet ohjataan sisäänkirjautumissivulle ja näytetään viesti, jossa kerrotaan miksi hänen tulee kirjautua sisään sivustolle. Näiden tapahtumien toteutumisen testaavat testitapaukset on paketoitu skenaarioksi, jota voidaan uudelleenkäyttää helposti. Käytäntöä voisi laajentaa ja vähentää näin testitapausten toistoa lisäämättä kuitenkaan monimutkaisuutta.

Lisäksi testien yhteneväisyyttä voisi parantaa. Tämä projekti on ensimmäinen, jossa olen tehnyt ohjelmistotestausta. Testaamista ei ole harjoitettu alusta asti, joten testien kattavuudessa ja tavoissa on yhtenäistämistä. Testauksen jatkuttua jo yli vuoden, siihen on löytynyt uusia parempia tapoja, kun työkalut ja testaaminen yleensä ovat tulleet tutummiksi. Olen näillä uusilla tavoilla korvannut vanhoja aina, kun olen tehnyt muutoksia vanhaan koodiin, mutta alueiden välillä on silti eroja.

Testaus on aina kompromissi käytettyjen resurssien ja saavutettujen hyötyjen välillä. Testien kattavuutta ja laatua voi aina parantaa, mutta loputtomiin se ei kannata. Jossain vaiheessa saavutetaan saturaatiopiste, jonka jälkeen lisätestaamisella ei enää saada vaadittuihin panostuksiin verrattavaa hyötyä. Wreckamovie.com:in nykyiset

testauskäytännöt ovat muodostuneet nykyisenlaisiksi niiden realiteettien puitteissa, joissa projekti toimii. Käytännöissä on varmasti aina parannettavaa, mutta yleisenä tavoitteena on ollut päätyä harkittuihin kompromisseihin ajan käytön ja saavutetun laadunparannuksen välillä. Jos jostakin käytännöstä ei ole saatu panostukseen vastaavaa hyötyä, sitä ei ole enää tehty.

Tästä on esimerkkinä Cucumber-kirjaston tarinatestityypin käyttö. Tämä tarina-testityyppi on hyväksymistestien tekoon tarkoitettu työkalu, jolla luodaan tekstimuotoisia tarinoita, joita voidaan suorittaa automatisoidusti. Itse tarina voi näyttää esimerkiksi tältä:

```
Ominaisuus: Uutisen luominen
  Jotta sivustolla olisi uutisia
  reportterien tulisi olla mahdollista lisätä niitä

  Tapaus: Sisäänkirjautunut reportteri luomassa uutista
    Oletetaan että olen sisään kirjautunut reportteri
    Ja että olen uutistenlisäys-sivulla
    Kun täytän "Jymy-uutinen" kohtaan "Otsikko"
    Ja täytän "Tämä on uutiseni sisältö." kohtaan "Sisältö"
    Ja painan "Tallenna"
    Niin näen tekstin "Uutinen luotu onnistuneesti"

  Tapaus: Sisäänkirjautunut reportteri luomassa
virheellistä uutista
    Oletetaan että olen sisään kirjautunut reportteri
    Ja että olen uutistenlisäys-sivulla
    Kun täytän kohdan "Otsikko" tekstillä ""
    Ja täytän kohdan "Sisältö" tekstillä ""
    Ja painan "Tallenna"
    Niin näen tekstin "Uutista ei voitu luoda puuttuvien
tietojen vuoksi"
```

Tarinan suorittava testauskoodi näyttäisi silloin tältä::

```
Given /^että olen sisään kirjautunut (.*)$/ do |role|
  @user = create_user
  login_user_as(@user, role)
end

Given /^että olen uutistenlisäys-sivulla$/ do
  visit '/uutiset/uusi'
end

When /^täytän "(.*)" kohtaan "(.*)"$/ do |value, field|
  fill_in(field, :with => value)
end

When /^painan '(.*)'$/ do |name|
  click_button(name)
end

Then /^näen tekstin '(.*)'$/ do |text|
  response.should contain(/#{text}/m)
end
```

Tarina on normaalia tekstiä ja tekstikoodi on Ruby-ohjelmointikieltä. Testikoodi löytää tarinan eri rivit säännöllisten lauseiden avulla ja asettaa tarinassa määritetyt arvot muuttujiin. Näitä arvoja käytetään edelleen valmistelemaan testiä, suorittamaan käyttäjän toiminto ja varmistamaan, että lopputulos on haluttu. Tähän käytetään Webrat-lisäkirjaston tarjoamia metodeja, jotka osaavat tehdä HTTP-kutsuja testattavalle sovellukselle ja tulkita sen palauttamaa HTML-koodia. Cucumber-kirjasto sisältää tuen lukuisille muillekin kirjastoille, joiden avulla Cucumberilla voidaan testata sovelluksia eri selainten sisällä ja lisäksi hyväksymistestata Adobe Flex-sovelluksia.

Tällä hetkellä käytössä ovat tarinamuotoiset hyväksymistestit joistakin tärkeimmistä käyttäjän läpikäymistä vaiheista esimerkiksi rekisteröinti, sisäänkirjautuminen ja tehtävän luonti. Nämä testit ovat hyviä varmistamaan, että käyttäjille tärkeimmät toiminnot toimivat takuuvarmasti. Kuitenkaan näistä testeistä saatu hyöty ei osoittautunut niihin kulutetun ajan arvoiseksi. Sama varmistus toimivuudesta saadaan

funktionaalisina testeinä toimivien kontrolleritestien kautta. Tarinamuotoisten testien suurin etu eli kommunikointikanava kehittäjätiimin ja asiakkaan välillä ei osoittautunut tässä projektissa tarpeeksi suureksi verrattuna käytettyyn aikaan. Tähän on syynä oletettavasti Wreckamovie.com-projektin pieni ja yhteen nivoutunut kehitystiimi, joka pääsi helposti yhteisymmärrykseen siitä, mihin sovellusta ollaan viemässä ja pystyi varmistamaan ilman virallista testausta, että halutut tavoitteet on saavutettu. Siten tarve hyväksymistesteille kehityksen ohjaamisen työkaluina oli vähäinen. Uskon, että toisenlaisissa projekteissa, erityisesti ulkoiselle asiakkaalle tehtävissä, hyväksymistestaamisesta voi olla suuriakin hyötyjä.

# 4 YHTEENVETO

Olen käsitellyt tässä työssä ohjelmistotestausta ja erityisesti kehittäjätestausta hyötyineen. Olen esitellyt Wreckamovie.com-projektin testauskäytäntöjä ja analysoinut niitä. Keskiverto projekti käyttää puolet ohjelmistokehitysajastaan virheiden paikallistamiseen, niiden korjaamiseen ja siitä seuraaviin muutostöihin (McConnell 2004, 30). Virheiden vähentämiselle ja niiden paikallistamisen helpottamiselle on siis tarvetta.

Ohjelmistotestaus on mielestäni todella tärkeä osa laadukasta ohjelmistokehitystä, mutta se ei kuitenkaan ole ainoa ja oikea laadunvarmistusmenetelmä. Automatisoidut ohjelmistotestit mahdollistavat regressiotestauksen ja koodin refaktoroinnin tekemällä niistä mahdollisimman helppoja ja kivuttomia suorittaa. Dynaamisten kielten tapauksessa yksikkötestit tuovat luottamusta, kun kääntäjä ei tarkasta ja tuo esille yksinkertaisten virheiden olemassaoloa. Automatisoidut testit eivät kuitenkaan ole ihmelääke laadukkuuteen.

Ohjelmistotestauksen menetelmillä yksistään päästään vain tiettyyn pisteeseen asti. *Code Complete* -kirjaan on koostettu taulukko kolmessa tutkimuksessa selvinneistä eri menetelmien virheidenlöytämiskyvyistä. Yksikkötestauksella löydetään keskiarvolta 30 prosenttia ja integraatiotestillä 35 prosenttia ohjelman virheistä. Kun sitä verrataan arkkitehtuuri- ja koodikatselmuksiin, joista saadut luvut ovat 55 ja 60 prosenttia, huomataan, että testien virheiden löytämiskyky jää huomattavasti alemmaksi. Huomioitavaa on myös, että tehokkainkaan menetelmä eli yli 1000 asennuksen beta-testaus, löytää parhaimmillaankin vain 85 prosenttia virheistä. Eri menetelmät

paljastavat erityyppisiä virheitä, joten lähes täydellisen virheiden löytymisen saavuttamiseksi täytyy menetelmiä yhdistellä. (McConnell 2004, 470.)

Lisäksi virheettömyys ei luonnollisestikaan takaa sovelluksen hyvää laatua. Ohjelman täytyy sisältää oikeat ominaisuudet ja sen tulee olla käytettävyydeltään hyvätasoinen ollakseen laadukas. Kaikkia resursseja ei kannatakaan käyttää teknisen laadukkuuden saavuttamiseen, vaan käyttää niitä myös esimerkiksi käyttäjä- ja käytettävyystutkimuksiin.

# 5 POHDINTA

Työn tavoitteena oli antaa koodin kanssa työskenteleville ihmisille katsaus ohjelmistotestaukseen ja antaa esimerkki ohjelmistotestauksen toteuttamisesta ohjelmistoprojektissa. Lisäksi tavoitteena oli analyysin kautta esittää Wreckamovie.com-palvelun ohjelmistotestaukselle kehittämisehdotuksia. Ensimmäistä tavoitetta ei voida suoranaisesti todentaa, mutta mielestäni työ antaa hyvän kuvan käsittelemästään osa-alueesta ja esittää ohjelmistotestausta aloittavalle tarpeellista tietoa. Wreckamovie.com-palvelun kehittämisehdotukset ovat osaltaan jo otettu käyttöön ja suurin osa tullaan ottamaan käyttöön, kun eri osien muokkauksen yhteydessä vanhoja käytäntöjä voidaan helposti vaihtaa uusiin. Olen yleisesti tyytyväinen työhöni.

Parannuksena työskentelyyn aikaistaisin rakenteen pääpiirteiden määrittelyä. Nyt tein pohjatutkimusta lukemalla kirjallisuutta ja kirjoittamalla sitten osia työstäni, mutta en mielestäni tarpeeksi aikaisin lyönyt lukkoon lopullista lukujakoa ja rakennetta. Tämän vuoksi jouduin melko myöhään työn tekemisessä tekemään suurehkoja muutoksia rajauksiin. Toisaalta tästä oli se etu, että työ sai muotoutua sen mukaan miten asiat tuntuivat parhaiten järjestyvän. Luultavasti tämän tyyppinen eteneminen johtui siitä, että minulla ei aloittaessani ollut valmista näkökulmaa siihen, mitä teoriaosuuden tulisi sisältää. Sen sisällys määräytyikin pitkälti Wreckamovie.com-palvelun ohjelmistotestauskäytäntöjen analyysistä saatujen tulosten perusteella.

Jatkossa olisi mielestäni mielenkiintoista tutkia, miten saada ohjelmistokehittäjät testaamaan. Tällä hetkellä iso osa ohjelmoinnista tehdään ilman mitään testejä, ja iso osa projekteista testataan vain ulkoisilla testaajilla. Kehittäjätestauksesta on kuitenkin

paljon hyötyä ja sen käytön levittäminen on mielestäni tärkeää. Kehittäjätestauksen käyttöönotto on pitkälti psykologinen asia ja kehittäjätestauksen hyötyjen esittelyn lisäksi kehittäjätestauksesta tulisi tehdä mielekästä ohjelmistokehittäjille. Testivetoinen kehitys on mielestäni menetelmä, joka tekee testaamisesta miellyttävämpää verrattuna perinteiseen testaamiseen. Testivetoisessa kehityksessä testien teko tulee osaksi kehitysprosessia, eikä se jää jälkeenpäin suoritettavaksi ikäväksi velvollisuudeksi. Testi-toteutus-refaktorointi-sykli saa aikaan monilla ohjelmistokehittäjillä hyvän etenemisrytmin, joka tukee koko ohjelmointiprosessia.

Lisäksi minusta olisi mielenkiintoista tutkia, mikä on ohjelmistotestauksen rooli projektien kokonaisprosessissa. Miten kehittäjätestaus integroituu erilaisiin prosessimalleihin ja muiden kuin testien tekijän työprosesseihin? Esimerkiksi miten kehittäjätestaus soveltuu designer-vetoisiin projekteihin?

LÄHTEET

Astels, Dave 2006. [Verkkodokumentti]
<http://blog.daveastels.com/files/BDD_Intro.pdf> (luettu 4.4.2009).

Behaviour-Driven Development – Introduction 2007. [Verkkodokumentti]
<http://behaviour-driven.org/Introduction> (luettu 19.2.2009).

Haikala, Ilkka & Märijärvi, Jukka 2004. Ohjelmistotuotanto. Helsinki: Talentum.

Hunt, Andrew, Thomas, David & Hargett, Matt 2007. Pragmatic Unit Testing In C# with NUnit. Dallas & Raleigh: The Pragmatic Bookshelf.

Massol, Vincent & Husted, Ted 2004. JUnit in Action. Greenwich: Manning Publications.

McConnell, Steve 2004. Code Complete, Second Edition. Redmond: Microsoft Press

Myers, Glenford J. Revised and Updated by Badgett, Tom and Thomas, Todd M. with Sandler, Corey 2004. The Art of Software Testing, Second Edition. Hoboken: John Wiley & Sons.

Rappin, Noel 2009. Again With The Test Driven Development. [Verkkodokumentti]
<http://www.pathf.com/blogs/2009/02/again-with-the-test-driven-development/> (luettu 2.3.2009).

Sisson, Derek 2000. Types of Tests. [Verkkodokumentti] <http://www.philosophe.com/testing/tests.html> (luettu 19.2.2009).

Star Wreck - In the Pirkinning – Distribution. [Verkkodokumentti]
<http://www.starwreck.com/distribution.php> (luettu 2.3.2009).

Thomas, Dave, Hansson, David Heinemeier, Breedt, Leon, Clark, Mike, Davidson, James Duncan, Gehtland, Justin & Schwarz, Andreas 2006. Agile Web Development with Rails. Second Edition. Dallas & Raleigh: The Pragmatic Bookself.

Liite 1. ProductionNewsItem-mallin testitiedosto.

```
module ProductionNewsItemSpecHelper
  def valid_production_news_item_attributes
    { :body => 'We just signed distribution with Universal.',
      :title => 'Great news!'
       }
  end
end

require File.dirname(__FILE__) + '/../spec_helper'

describe ProductionNewsItem do
  describe "associations" do
    it "should respond to user" do
      ProductionNewsItem.reflect_on_association(:user).should_not
     be_nil
      ProductionNewsItem.reflect_on_association(:user).macro.should
     == :belongs_to
    end

    it "should respond to production" do
      ProductionNewsItem.reflect_on_association(:production).should_no
     t be_nil
      ProductionNewsItem.reflect_on_association(:production).macro.sho
     uld == :belongs_to
    end

    it "should respond to comments" do
      ProductionNewsItem.reflect_on_association(:comments).should_not
     be_nil
      ProductionNewsItem.reflect_on_association(:comments).macro.shoul
     d == :has_many
    end
  end

  describe "validations" do
    include ProductionNewsItemSpecHelper

    it "should validate presence of title" do
      @news_item =
     ProductionNewsItem.new(valid_production_news_item_attributes.exce
     pt(:title))
      @news_item.valid?
      @news_item.errors.on(:title).should == "can't be blank"
    end

    it "should validate presence of body" do
      @news_item =
     ProductionNewsItem.new(valid_production_news_item_attributes.exce
     pt(:body))
      @news_item.valid?
      @news_item.errors.on(:body).should == "can't be blank"
    end
  end
```

```
describe "named scopes" do
  describe "recent" do
    it "should generate correct conditions" do
      now = Time.now
      Time.stub!(:now).and_return(now)
      expected = {:conditions => ['created_at > ?', 2.weeks.ago]}
      ProductionNewsItem.recent.proxy_options.should == expected
    end
  end
end

describe "#find_by_production" do
  it "should find 5 most recent news items by production" do
    @news_item = mock("news_item")
    ProductionNewsItem.should_receive(:find).with(:all, :limit =>
   5, :conditions => ["production_id = ?", 100], :order =>
   'created_at DESC').and_return([@news_item])
    ProductionNewsItem.find_by_production(100).should ==
   [@news_item]
  end

  it "should use limit argument if passed" do
    @news_item = mock("news_item")
    ProductionNewsItem.should_receive(:find).with(:all, :limit =>
   20, :conditions => ["production_id = ?", 100], :order =>
   'created_at DESC').and_return([@news_item])
    ProductionNewsItem.find_by_production(100, 20).should ==
   [@news_item]
  end
end

describe "#publish_events?" do
  before(:each) do
    @news_item = ProductionNewsItem.new
  end

  it "should return true if production is published" do
    @news_item.stub_association!(:production, :visible? => true)
    @news_item.publish_events?.should be_true
  end

  it "should return false if production is unpublished" do
    @news_item.stub_association!(:production, :visible? => false)
    @news_item.publish_events?.should be_false
  end
end

describe "#editable_by?" do
  before(:each) do
    @news_item = ProductionNewsItem.new
    @user = mock_model(User)
  end

  it "should ask from production" do
    production = mock_model(Production)
    production.should_receive(:leader?).with(@user)
    @news_item.stub!(:production).and_return(production)
```

```
      @news_item.editable_by?(@user)
    end

    it "should return true" do
      @news_item.stub_association!(:production, :leader? => true)
      @news_item.editable_by?(@user).should be_true
    end

    it "should return false" do
      @news_item.stub_association!(:production, :leader? => false)
      @news_item.editable_by?(@user).should be_false
    end
  end
end
```

Liite 2. ToDoController-kontrollerin testitiedosto.

```
require File.expand_path(File.dirname(__FILE__) +
     '/../integration_spec_helper')

describe ToDoController do
  integrate_views

  describe "POST create" do
    describe "succesful" do
      def act!
        post :create, {:id => @task.id }
      end

      before(:each) do
        ToDoItem.destroy_all
        @user = Factory(:normal_user)
        controller.stub!(:current_user).and_return(@user)

        @task = Factory(:task)
      end

      it "should not be accessible for users not logged in" do
        controller.stub!(:current_user).and_return(:false)
        act!
        response.should redirect_to(login_path)
      end

      it "should mark task read" do
        NewTaskEvent.should_receive(:mark_seen).with(@task, @user)
        act!
      end

      it "should create new to do item" do
        act!
        to_do_item = ToDoItem.first
        to_do_item.task_id.should == @task.id
        to_do_item.user_id.should == @user.id
      end

      it "should set flash" do
        act!
        flash[:notice].should == "Task was added to your <a
     href='/user/todo'>Todo list</a>."
      end

      it "should redirect" do
        act!
        response.should redirect_to(:controller => 'tasks', :action =>
     'show', :id => @task)
      end

      it "should render text 'success' when called with xhr" do
        xhr :post, :create, {:id => @task.id}
        response.body.should == "success"
      end
    end
```

```
describe "already on users todo list" do
  def act!
    post :create, {:id => @task.id }
  end

  before(:each) do
    ToDoItem.destroy_all
    @user = Factory(:normal_user)
    controller.stub!(:current_user).and_return(@user)

    @task = Factory(:task)
    # add task to users to do list
    Factory(:to_do_item, :task => @task, :user => @user)
  end

  it "should not create to do item" do
    act!
    ToDoItem.count.should == 1
  end

  it "should set flash" do
    act!
    flash[:notice].should == "This task is already on your <a
 href='/user/todo'>Todo list</a>."
  end

  it "should redirect" do
    act!
    response.should redirect_to(:controller => 'tasks', :action =>
 'show', :id => @task)
  end
end

describe "save unsuccesful" do
  def act!
    post :create, {:id => @task.id }
  end

  before(:each) do
    ToDoItem.destroy_all
    @user = Factory(:normal_user)
    controller.stub!(:current_user).and_return(@user)

    @task = Factory(:task)
    @to_do_item = mock_model(ToDoItem, :save => false)
    ToDoItem.stub!(:new).and_return(@to_do_item)
  end

  it "should set flash" do
    act!
    flash[:notice].should == 'There was an error while adding task
 to list.'
  end

  it "should redirect" do
    act!
```

```
      response.should redirect_to(:controller => 'tasks', :action =>
    'show', :id => @task)
    end

    it "should render text 'error' when called with xhr" do
      xhr :post, :create, {:id => @task.id}
      response.body.should == "error"
    end
  end
end

describe "POST destroy" do
  describe "succesful" do
    def act!
      post :destroy, :id => @to_do_item.id
    end

    before(:each) do
      @user = Factory(:normal_user)
      controller.stub!(:current_user).and_return(@user)

      @task = Factory(:task)
      @to_do_item = Factory(:to_do_item, :user => @user, :task =>
   @task)
    end

    it "should destroy to do" do
      act!
      lambda {@to_do_item.reload}.should
   raise_error(ActiveRecord::RecordNotFound)
    end

    it "should set flash" do
      act!
      flash[:notice].should == "Task #{@task.title} was removed from
   your Todo list."
    end

    it "should redirect" do
      act!
      response.should redirect_to(:controller => 'user', :action =>
   'todo')
    end
  end

  describe "current user is not the author of the to do" do
    def act!
      post :destroy, :id => @to_do_item.id
    end

    before(:each) do
      @user = Factory(:normal_user)
      controller.stub!(:current_user).and_return(@user)

      @to_do_item = Factory(:to_do_item, :user =>
   Factory(:passive_normal_user))
    end
```

```
      it "should not destroy to do" do
        act!
        lambda {@to_do_item.reload}.should_not raise_error
      end

      it "should redirect" do
        act!
        response.should redirect_to(:controller => 'user', :action =>
      'todo')
      end
    end
  end
end
```