Joonas Jauhiainen

# MALTTI – työkalu matahiilisen aluekehityksen tueksi

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinöörityö

15.9.2012

Metropolia

| | |
|---|---|
| Tekijä(t)<br>Otsikko<br><br>Sivumäärä<br>Aika | Joonas Jauhiainen<br>MALTTI - Matalahiilisen aluekehityksen työkalu<br><br>40 sivua<br>15.9.2012 |
| Tutkinto | insinööri (AMK) |
| Koulutusohjelma | tietotekniikka |
| Suuntautumisvaihtoehto | ohjelmistotekniikka |
| Ohjaaja | lehtori Kimmo Sauren |

Tässä työssä esitellään Eficode Oy:n asiakkaalle Aalto -yliopistolle tehtyä MALTTI -projektia. MALTTI on web-pohjainen järjestelmä, jonka toteuttamiseen valittiin Ruby on Rails -ohjelmistokehys. Sovelluksen kehitysympäristönä toimi Linux pohjainen Ubuntu, sekä avoimen lähdekoodin kehitysympäristö Aptana Studio 3.

Sovelluksella pystyy simuloimaan alueen muodostamia hiilijalanjäkiä jopa 50 vuoden aikavälillä. MALTTI projektin tuotanto aloitettiin keväällä 2012 ja päärahoittajana on toiminut EAKR. Sovelluksen avulla toivotaan parantamaan alueiden rakentamisen ja käytön aikaisia hiilijalanjälki päästöjä, ottamalla huomioon hiilijalanjälkeen vaikuttavat tekijät jo suunnitteluvaiheessa.

Järjestelmän tuottamiseen käytettiin Scrum vaihejakomallia ja testivetoisen kehityksen menetelmää.Projektia toteutettiin kolmessa sprintissä, yhteensä noin kaksi kuukautta. Kolmannen sprintin jälkeen asiakkaalle saatiin toteutettua toimiva järjestelmä ja tällä hetkellä järjestelmä on asiakkaan testattavana.Jokaisen sprintin jälkeen oli asiakkaan luona palaveri jossa kerrottiin edistymisestä ja selvitettiin oliko tullut kysymyksiä tai ongelmia projektin suhteen.

| | |
|---|---|
| Avainsanat | Ruby on Rails, Scrum, Test Driven Development, Hiilijalanjälki |

| | |
|---|---|
| Author(s)<br>Title<br><br>Number of Pages<br>Date | JoonasJauhiainen<br>MALTTI - Tool to support low-carbon neighborhood construction<br><br>40 pages<br>15 September 2012 |
| Degree | Bachelor of Engineering |
| Degree Programme | Information Technology |
| Specialisation option | Software Engineering |
| Instructor | Kimmo Sauren, Senior Lecturer |

Eficode Oy's client Aalto University has placed an order for a web application to be used to simulate carbon emissions when planning neighborhood construction and use; this application is called MALTTI (low-carbon emission neighborhood construction support tool). Developing the application was the topic of the thesis. The application was created using the Ruby on Rails framework.

The project was managed by using the Scrum process which is widely used in agile projects. Ruby on Rails offers a complete set to create functional and unit tests for the web applications. This made it easy to implement the Test Driven Development method, which is connected to Scrum. The application was developed using the Ubuntu and Aptana Studio 3 open source operation systems.

The project was developed in three sprints and the development continued for about two months. After each sprint there was a meeting with the customer where the progression of the work was shown and issues relating to the project were talked about. After the third sprint the customer was given a working version of the application and currently it is tested by the customer.

| | |
|---|---|
| Keywords | Ruby on Rails, Scrum, Test Driven Development |

## Sisällys

Lyhenteet

1 Johdanto 1

2 MALTTI 1

3 Työkalut ja menetelmät 2

3.1 Ketterät menetelmät: Scrum 2

3.2 Scrum verrattuna vesiputousmalliin 6

3.3 Testivetoinen kehitys 7

3.4 Työkalut 8

3.5 Versionhallinta 9

4 Ruby On Rails -viitekehys 10

4.1 Ruby 10

4.2 Rails 11

4.3 MVC-suunnittelumalli 12

5 MALTTI-työkalun toteutus 13

5.1 Työn vaiheistus 13

5.2 Käyttäjätarinat 14

5.2.1 Sprint 2:n käyttäjätarinat 16

5.3 Käyttöliittymä 22

5.4 Tekninen toteutus 23

5.5 Testaus 37

6 Yhteenveto 37

Lähteet 39

## Lyhenteet

| | |
|---|---|
| ASP.NET | Microsoftin web-ohjelmointikehys |
| CRUD | Muodostuu sanoista Create, Read, Update ja Delete. |
| HTTP | Hypertext Transfer Protocol on protokolla, jota selaimet ja WWW-palvelimet käyttävät tiedonsiirtoon. |
| jQuery | jQuery on avoimen lähdekoodin JavaScript-kirjasto. |
| PHP | Hypertext Preprocessor on ohjelmointikieli, jolla voi luoda dynaamisia websivuja tai -sovelluksia. |
| REST | Representational State Transfer on HTTP-protokollaan perustuva arkkitehtuurimalli ohjelmointirajapintojen toteuttamiseen |
| ROR | Ruby on Rails. Viitekehys web -sovelluksille, joka on kirjoitettu Ruby -skritpikielellä. |
| TDD | Test Drive Development eli testivetoinen kehitys |

# 1 Johdanto

Tämä insinöörityö esittelee Eficode Oy:n asiakkaalle Aalto -yliopistolle tehtyä hiilijalanjälki simulointityökalua. Kyseinen työkalu on tehty web-pohjaiseksi ja tarkoituksena on konkretisoida kaupunkisuunnittelijan suunnitteluvaiheessa tehtyjä valintoja, joilla voisi vähentää hiilijalanjälkeä rakentamisen ja käytön aikana. Järjestelmään syötetty tieto muutetaan helposti luettaviksi diagrammeiksi ja se mahdollistaa myös kahden eri skenaarion vertailun keskenään. Lisäksi tämän tutkintotyön tarkoituksena on tarkentaa teknistä toteutusta sekä menetelmiä, joita käytetään yleisesti ketterissä ohjelmistoprojekteissa.

Järjestelmän toteutustavaksi valittiin Ruby on Rails -ohjelmistokehys, sillä se vaikutti parhaimmalta ratkaisulta projektia aloitettaessa. Syitä oli muun muassa nopea ja kevyt kieli ja pitää sisällään valmiit työkalut verkkosovellusten testaamiseen.

# 2 MALTTI

Projektin tausta

Hankkeen tausta oli EU-projekti, jonka päärahoittajana on Euroopan aluekehitysrahasto (EAKR). Hankkeen tarkoituksena on hakea keinoja energia- ja ekotehokkuuden parantamiseksi rakennuksissa ja hiilijalanjäljen pienentämiseksi. Kehityskohteisiin kuuluivat seuraavat asiat: kiinteistöjen energiatehokkuus, jätehuolto, kierrätys ja älykäs talotekniikka. (1.) EAKR tukee hankkeita jotka kehittävät yrityksiä, innovaatioiden syntymistä, verkottumista, osaamista ja alueiden saavutettavuutta. MALTTI -projektia on rahoittaneet myös toiminta-alueen kunnat sekä yritykset. Eficode Oy:n osalta oli tehdä Aaltoyliopistolle web -sovellus kaupunkisuunnittelijoille nimeltä MALTTI (Matalahiilisen aluekehityksen työkalu). Aalto -yliopisto oli tehnyt tarvittavat taustatyöt ja laskelmat, joiden pohjalta tehtiin ROR (Ruby on Rails) -sovellus.

Hiilijalanjälki

Hiilijalanjäljellä viitataan yleisesti tuotteen, toiminnan tai palvelun aiheuttamaan ilmastokuormaan eli siihen kuinka paljon kasvihuonekaasuja tuotteen tai toiminnan

elinkaaren aikana on syntynyt. Lisäksi voidaan myös viitata kasvihuonekaasujen kokonaispäästöjen sijasta pelkästään hiilidioksidipäästöihin. Hiilijalanjäljen suuruus vaikuttaa ilmaston lämpenemiseen, mikä vaikuttaa alailmakehän ja merien keskilämpötilan muutoksiin. (2.)

Hiilijalanjäljen käsitteet on kehitetty mittariksi, joiden avulla voidaan arvioida erilaiset tekojen ja kulutusvalintojen vaikutus ilmaston lämpenemiseen. Hiilijalanjäljellä voidaan esimerkiksi esittää, että kuinka paljon autolla ajaminen tai talotyypin ylläpito vaikuttaa ilmastoon. (2.)

Hiilijalanjälki ilmoitetaan massana eikä pinta-alana. MALTTI sovelluksessa käytetäänkin yksikköä t $CO_2$e eli tonneina hiilidioksidiekvivalenttia. Hiilidioksidiekvivalentti lasketaan kaikista kasvihuonekaasuista, ottamalla huomioon ilmaston lämpöön kohdistuva vaikutus verrattuna hiilidioksidiin. (2.)

Hiilijalanjälki voidaan jakaa suoriin ja epäsuoriin päästöiin. Kuluttajan toiminnan aiheuttamat päästöt luokitellaan suoriin päästöihin, kuten esimerkiksi autolla ajaminen. Toisaalta taas joukkoliikenteen aiheuttamat päästöt lasketaan epäsuoriin päästöihin, kuten myös kulutushyödykkeet ja palvelut. (2.)

# 3 Työkalut ja menetelmät

## 3.1 Ketterät menetelmät: Scrum

Ketteristä menetelmistä puhutaan silloin, kun ohjelmistokehityksen aikana tapahtuvat iteraatiot hyvin lyhyitä. Iteraatioiden lyhyyden mahdollistaa testitapausten suorittaminen automatisoidusti. Käytännössä uuden ominaisuuden tai toiminnallisuuden lisääminen aloitetaan tekemällä testitapauksia, jonka jälkeen testitapaukset ajetaan ja todetaan epäonnistuvan, sillä tarvittavaa koodia ei ole vielä tässä vaiheessa tehty, minkä jälkeen aloitetaan muutosten ohjelmointi ja ohjelmaa kirjoitetaan niin kauan, kunnes testitapaukset menevät läpi. MALTTI-projektissa käytettiin Scrum -projektinhallinnan viitekehystä. (3, s. 47.)

Scrum on yksi versio ketterien kehitysmenetelmien viitekehyksistä, ja sen ensimmäisen version esittelivät Hirotaka Takeuchi ja Ikujiro Nonaka vuonna 1986. Scrum

mahdollistaa useiden prosessien ja tekniikoiden käytön. Tämän vuoksi menetelmiä voidaan parantaa kesken projektin, ja se tarjoaa myös puitteet monimutkaisten tuotteiden kehittämiseen. Scrum perustuu empiiriseen prosessihallintateoriaan, jonka lähestymistapa on interaktiivis-inkrementaalinen, ja sen avulla voidaan käsitellä paremmin ennustettavuutta, optimointia ja riskejä. Empiirinen prosessihallinta pitää sisällään kolme tukijalkaa: läpinäkyvyys, tarkastelu ja sopeutuminen. (4, s. 4 − 5.)

*Läpinäkyvyydellä* tarkoitetaan prosessiin ja projektiin vaikuttavien tekijöiden näkymistä myös muille, kuin pelkästään ohjelmoijille. Sen lisäksi, että lopputulos ja siihen vaikuttavat tekijät ovat näkyvissä, ne tulee myös olla yksiselitteisesti tulkittavissa.

Toinen tukijaloista on *tarkastelu*, jota tulee tehdä prosessin aika usein ja myös prosessin eri osille, jotta mahdolliset ongelmatilanteet voidaan havaita ajoissa. Kolmantena tukijalkana pidetään *sopeuttamista*, jossa prosessin tarkastelijan tehtävänä on päätellä, mikäli yksi tai useampi prosessin osa on hyväksyttävien raja-arvojen ulkopuolella eikä tämän vuoksi toteuta haluttua lopputulosta. Mikäli tämän kaltaiseen tilanteeseen on ajauduttu, tulee tarkastelijan säätää prosessia tai mahdollisia resursseja. Säätö tulee kuitenkin toteuttaa mahdollisimman nopeasti, jotta muita ongelmia ei ehdi toteutua.

Scrum -tiimit

Scrum on siis viitekehys, joka koostuu scrum -tiimeistä, jotka sisältävät eri rooleja, aikarajat, dokumentit ja säännöt. Scrum -tiimin ajatuksena on optimoida työn joustavuus ja tuottavuus. Tämän vuoksi ryhmän tulisi olla itseorganisoituva, osata tarvittavat teknologiat ja työskennellä kehitysjaksoissa, joita kutsutaan sprinteiksi. Scrum -tiimi muodostuu scrum masterista, tuotteenomistajasta ja kehitystiimistä.

Scrum masterin tarkoitus on seurata ja pitää huolta, että srcumin periaatteita noudatetaan projektin aikana. Scrum master voi myös olla linkkinä tuotteenomistajan, kehitystiimin ja muun organisaation välillä. Tuoteomistaja vastaa tuotteesta, eli on hyvin usein asiakas, jolle tuote tai projekti toteutetaan. Scrumtiimin tarkoituksena on suorittaa kehitysjonossa olevat tehtävät sprintin sisällä. (4, s. 5 − 6).

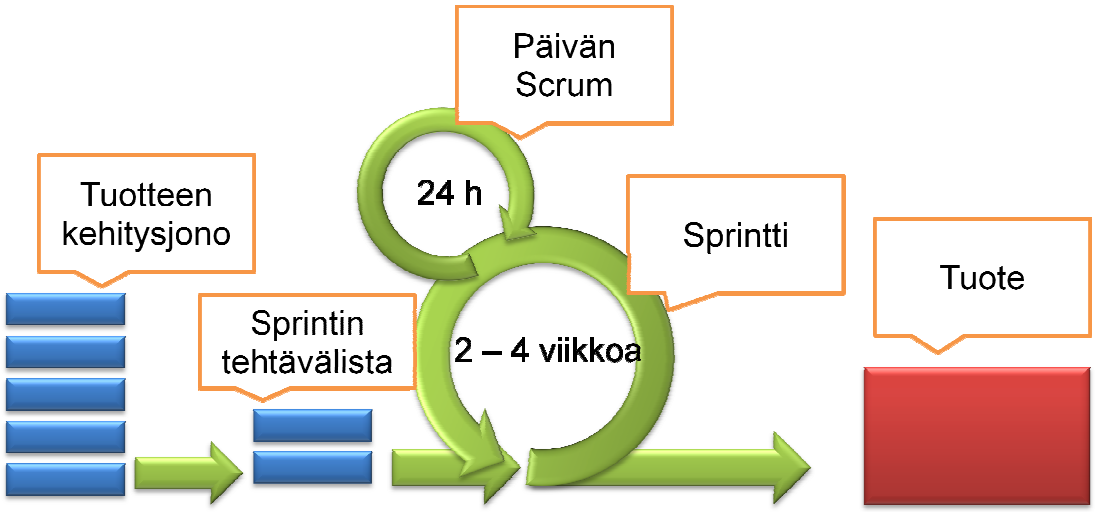
Aikajanat

Scrum sisältää kuusi eri aikajanaa: julkaisun suunnittelukokous, sprintin suunnittelukokous, sprintti, päivän scrum, sprintin katselmointikokous ja sprintin retroperspektiivi. Julkaisukokous on yleensä ensimmäinen kokous asiakkaan kanssa ja tarkoituksena on saada kaikki osapuolet ymmärtämään, mitä tuotteelta halutaan ja keinot, joilla osapuolet pystyvät viestimään toisilleen. Sprintin suunnittelukokuksen kesto on enintään kahdeksan tuntia ja kokous käsittelee seuraavan sprintin aikana toteutettavia tehtäviä.

Sprintillä tarkoitetaan siis lyhyttä kehitysjaksoa, joka on pituudeltaan enintään yhden kuukauden, kuten kuvassa 1 on kuvattu. Sprintin aikana scrum master pitää huolen, että kehitystiimi tekee vain suunnittelukokouksessa määrätyt tehtävät. Sprintti sisältää suunnittelun, kehitystyön, päivän scrumin, katselmoinnin ja retroperspektiivin. Sprinttien välillä ei ole erillisiä taukoja.

Päivän scrum on yleensä päivittäinen noin 15 minuuttia kestävä palaveri, johon osallistuu kehitystiimi, scrum master ja yleensä myös tuoteen omistaja. Päivän scrumin aikana käydään läpi, mitä kehitystiimi on tehnyt ja mitä se tulee kyseisenä päivänä tekemään ja myös ilmoittaa mahdollisista ongelmista. Ongelmien ratkaisuja ei ole kuitenkaan tarkoitus käydä päivän scrumin aikana, vaan se tapahtuu hyvin usein esimerkiksi kokouksen jälkeen.

Sprintin lopussa on yleensä sprintin katselmointi kokous, johon varataan noin 2,5 % koko sprintin ajasta ja kokouksessa käydään läpi sprintin saavutukset. Tuoteen omistaja yleensä katselmukokouksessa ilmoittaa mahdollisista päivityksistä ja muutoksista. Katselmoinnin jälkeen seuraa seuraavan sprintin suunnittelukokous, mutta näiden kahden kokouksen välissä voi myös olla sprintin retroperspektiivikokous, mikäli se nähdään tarpeelliseksi. Retroperspektiivikokouksen tarkoituksena on tarkastella sprintin onnistumista, ihmisten, ihmissuhteiden, prosessien ja työkalujen näkökulmasta ja niiden avulla määrittää hyvin menneet asiat ja se mitä voitaisiin tehdä paremmin. (4, s. 8 – 12.)




Kuva 1. Scrumin eteneminen

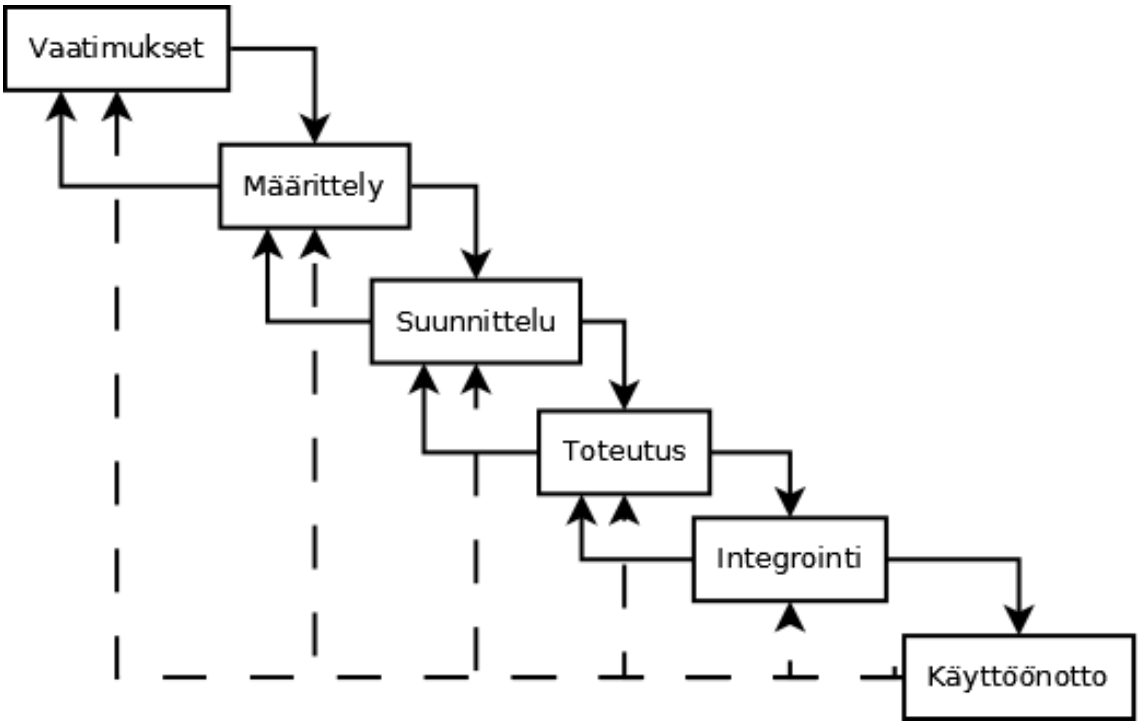
Dokumentit

Scrumia varten tulee olla neljä erilaista dokumenttia: tuotteen kehitysjono (backlog), sprintin tehtävälista (sprintlog), julkaisun edistymiskäyrä ja sprintin edistymiskäyrä. Tuotteen kehitysjonossa listataan kaikki vaatimukset tuotteelle, jota kehitysryhmä kehittää. Tuoteen omistaja vastaa kehitysjonosta, sisällöstä, saatavuudesta ja priorisoinnista. Kehitysjono kehittyy tuotteen kehittyessä. Julkaisun edistymiskäyrä on diagrammi, joka kertoo, kuinka paljon työtä on jäljellä suhteessa aikaan. Aikayksikkö on scrum -tiimin ja organisaation yhdessä valitsema. Esimerkiksi tunti tai päivä.

Sprintin tehtävälista koostuu tehtävistä, jotka kehitystiimi toteuttaa, jotta halutut toimenpiteet tulisivat julkaisukelpoisiksi. Listan tehtävät yleensä määritellään suunnittelukokouksessa, ja kehitystiimi voi tehdä tarvittaessa muutoksia tehtävälistaan. Sprintin edistymiskäyrä on sama kuin julkaisun edistymiskäyrä, mutta kuvaa työnmäärää suhteessa sprintissä jäljellä olevaan aikaan. (4, s. 12 – 14.)

## 3.2 Scrum verrattuna vesiputousmalliin

Vesiputousmallia pidetään yleensä perinteisenä vaihejakomallina projektin kehityksessä ja se sisältää lähes aina määrittely-, suunnittelu- ja toteutusvaiheen. Vaiheita voi olla enemmän, ja vesiputousmallista löytyy useita eri versioita. (3. s. 37.)




Kuva 2. Esimerkki vesiputousmallin vaiheista.

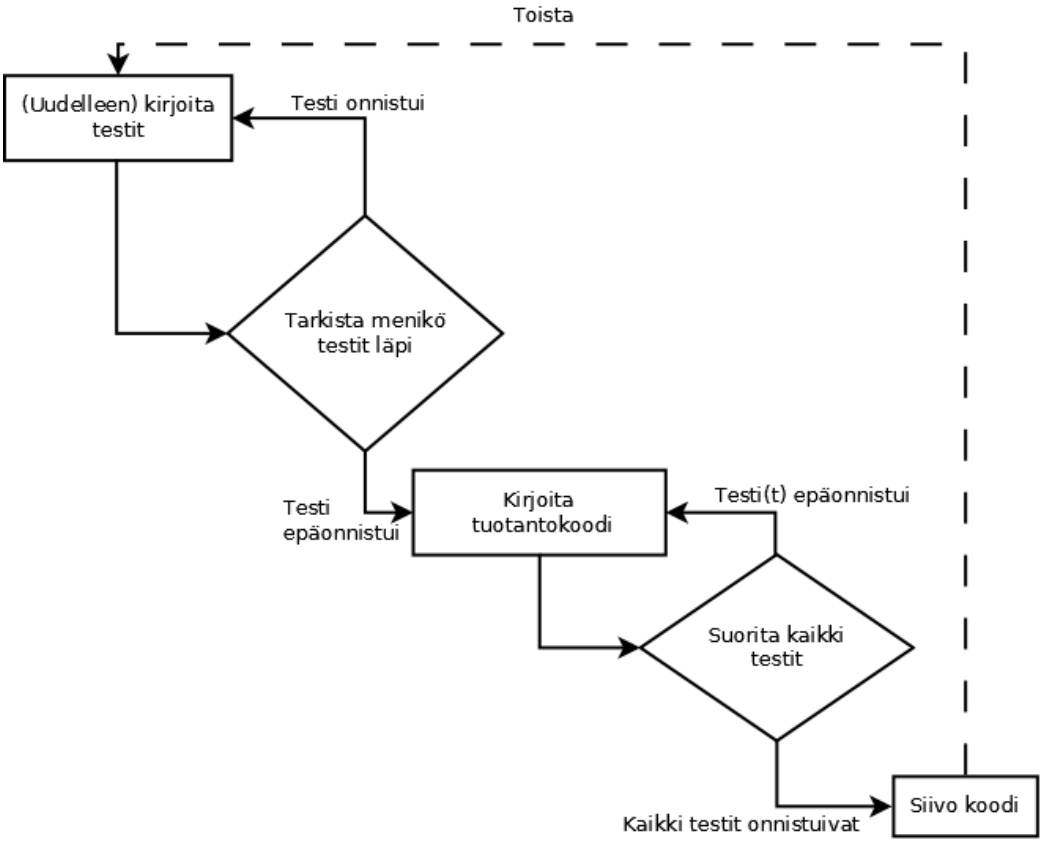
Vesiputousvaihejakomallissa on tavallista toteuttaa koko projekti vaiheittain ja harvoin on tapana enää siirtyä aikaisempaan vaiheeseen, kun se on tehty. Haittapuolina tässä mallissa verrattuna Scrum -vaihejakomalliin on, että projektin lopputulosta ei nähdä, kunnes ihan projektin lopussa. Scrumin ajatuksena on toteuttaa kaikki toiminnot pienissä paloissa, ja näin ollen jokaisen sprintin jälkeen tulisi olla täysin testattu ja valmis prototyyppi, joka on integroitu aikaisempien sprinttien tuotoksiin.

Vesiputousmallissa myös harvemmin keskustellaan asiakkaan kanssa projektin edetessä, kun taas Scrumissa on päivittäiset lyhyet palaverit ja jokaisen sprintin jälkeen sprintin katselmointikokous. Mikäli huomataan esimerkiksi puutteita määrittelyssä toteutuksen aikana vesiputousmallissa, joudutaan kuvan 2 mukaisesti palaamaan takaisin määrittelyyn ja toteuttamaan myös suunnittelu uudestaan, ennen kuin voidaan palata takaisin toteutukseen.

Vesiputousmallissa projektipäällikkö on yleensä vastuussa koko projektin kehityksestä ja vaikuttaa ratkaisuihin, joilla projekti toteutetaan. Scrumissa kehitystiimi taas tekee tarvittavat päätökset ongelmien ratkaisuihin ja Scrum Master lähinnä pitää huolen, että kehitystiimi toimii Scrummin ohjeiden mukaisesti ja on myös tarvittaessa välikätenä asiakkaan ja muun organisaation välissä. (5.)

## 3.3 Testivetoinen kehitys

Testivetoinen kehitys on tekniikka, joka tukee ohjelmointia. Ennen kuin yhtään tuotantokoodia on kirjoitettu, määritellään testitapaukset, minkä jälkeen kirjoitetaan tuotantokoodia sen verran, että testit menevät läpi. Yksikkötestit ovat yleisimpiä testejä, joita kirjoitetaan. Lisäksi voidaan myös määritellä esimerkiksi käyttäjätarinatestit, joilla testataan ohjelman toimivuutta haluttuihin toiminnallisuuksiin. Kuvan 3 mukaisesti testit kirjoitetaan ensiksi, minkä jälkeen tarkistetaan, menivätkö testit läpi, ja mikäli puutteita huomattiin, kirjoitetaan tuotantokoodia niin kauan, kunnes testit menevät läpi. Kun kaikki testit on tehty, siivotaan koodi, mikäli se nähdään tarpeelliseksi.




Kuva 3. Testivetoisen kehityksen työn kulku

Testivetoisella kehityksellä pyritään parempaan rajapintasuunnitteluun, varmistumaan ohjelmiston oikeasta toiminnasta ja pääsemään eroon perinne koodista. Mikäli testit on kirjoitettu ja ne ovat loogisia, myös jatkokehitys on helpompaa, sillä jos projektiin tulee ohjelmoija, joka ei ole aikaisemmin kyseiseen projektiin tuottanut mitään, voi varmistaa testeillä, että uusi toiminnallisuus ei ole vaikuttanut vanhaan järjestelmään. Testivetoinen kehitys ei ole testausmenetelmä vaan suunnittelumenetelmä. (6, s. 70 – 71.)

## 3.4 Työkalut

### Ubuntu työpöytä käyttöjärjestelmä

Ubuntu- ja Unix-pohjaiset käyttöjärjestelmät ovat parhaimpia ROR -kehitykseen, sillä Ruby itsessään on Unix -pohjainen kieli. Lisäksi Eficode Oy on keskittynyt käyttämään avoimen lähdekoodin työkaluja. Windows-käyttöjärjestelmälle on omat kääntäjät Rubya ja ROR:ia varten, mutta esimerkiksi kaikkia Ruby Gemejä ei saa asennettua Windows-koneille.

### Aptana Studio 3

Aptana Studio 3 on avoimen lähdekoodin ohjelmointiympäristö, ja se sisältää monipuoliset työkalut internetsivujen ja palveluiden tekoon. Aptana Studio 3:sta on esimerkiksi syntaksin huomiovärien lisääminen Ruby-kielen syntaksia hyväksikäyttäen. Lisäksi eri ohjelmointikoodilohkojen piilottaminen onnistuu Aptana Studio 3:lla myös Rubyn kanssa. Aptana Studio 3:ssa on sisälletytetty myös terminaali, mikä on hyödyllinen ROR-ohjelmoinnissa. (7.)

### JIRA

JIRA on tehtävienhallintaohjelmisto, joka on suunniteltu lähinnä ohjelmistotuotantoa varten (8). JIRA:n avulla kehittäjät, projektipäällikkö, tuotteenomistaja ja organisaatio näkevät, miten projekti etenee. MALTTI-projektissa tuotteenomistaja ja projektipäällikkö tekivät tarvittavat käyttäjätarinat tuotteen kehitysjonoon.

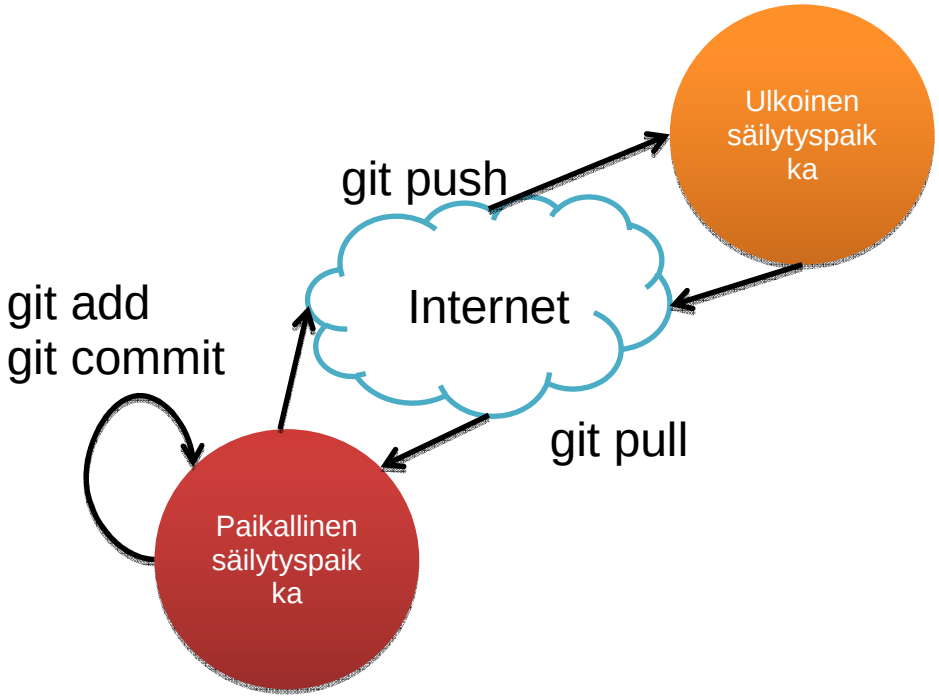
### Deveo

Deveo on projektinhallintatyökalu, jossa voi jakaa koodia muiden projektiin kuuluvien jäsenten kanssa. Deveossa voi määrittää eri rooleja projektiin kuuluvien kesken. Deveossa näkee myös, mitä muutoksia lähdekoodiin on tehty, milloin ja kenen toimesta. (9).

## 3.5 Versionhallinta

Versionhallinnalla tarkoitetaan menetelmää, tai ohjelmaa jolla voidaan hallita eri versioita lähdekoodista. Ohjelmistotuotannon projekteissa on tärkeää, että kehittäjät eivät vahingossa häiritse toistensa työskentelyä. Esimerkiksi tekemällä eri muutoksia samaan moduuliin voi vioittaa ohjelman toimintaa. Versionhallintaan on useita eri työkaluja ja menetelmiä. MALTTI-projektin kehityksessä käytimme Git -nimistä versionhallinta ohjelmistoa. (3. s. 260.)

Git on ilmainen avoimen lähdekoodin versionhallintaohjelmisto, joka on suunniteltu toimimaan sekä pienissä että isoissa projekteissa nopeasti ja luotettavasti. Git on tuotettu Linuxin kehitysyhteisön avulla, ja suurimpia vaikuttajia on ollut Linus Torvalds.




Kuva 4. Gitin toimintaperiaate

Gitin toimintaperiaate perustuu paikalliseen säilytyspaikkaan sekä ulkoiseen säilytyspaikkaan (ks. kuva 4), jota käytetään yleensä internetin kautta. Käytännössä paikallinen säilytyspaikka sisältää kaikki tehdyt muutokset ja haarat. Ulkoinessa säilytyspaikassa vain ne haarat, jotka halutaan jakaa muille kehittäjille. Uuden paikallisen säilytyspaikan luomisessa tehdään oletuksena master niminen haara, joka on yleensä myös se haara, jossa säilytetään viimeisin koodi. Muutokset, joita tehdään paikalliseen säilytyspaikkaan säilyvät eri versioina. Gitissä on olemassa takaisinpaluutoiminto, sillä mikäli joku muutos häiritsee jo olemassa olevaa toimintoa, voidaan palauttaa vanhat muutokset takaisin. Nämä muutostiedot siirtyvät myös ulkoiseen säilytystilaan, kun paikallisesta säilytystilasta siirretään tietoja ulkoiseen säilytystilaan. (10 s. 4 − 5).

# 4 Ruby On Rails -viitekehys

## 4.1 Ruby

Ruby on dynaaminen ja dynaamisesti tyypittettävä oliopohjainen kuvauskieli, jonka ensimmäinen versio kehitettiin vuonna 1995. Rubyn on kehittänyt Yukihiro Matsumoto ja hänen ajatuksenaan oli toteuttaa kieli, joka olisi suoraviivainen ja nopea. Ruby pohjautuu Lispiin, Dylaniin, CLU:hun ja Perliin. Ruby julkaistaan avoimen lähdekoodin lisenssin alaisena. (11. s. 14.)

Rubyllä ei ole tarkoitus toteuttaa kokonaisia ohjelmia, vaikka myös tämä on mahdollista, vaan toimia tietynlaisena kokoajakielenä, jolla voidaan tehdä kirjastoja, työkaluja ja käyttöliittymämuokkauksia suurempaan ohjelmaan.

Rubyn kaikki muuttujat ovat olioita, mikä tekee siitä erilaisen verrattuna muihin yleisiin tulkattaviin kieliin, kuten Pythoniin tai JavaScriptiin. Rubyn staattiset objektit kuten kokonaisluvut tai operaattorit sisältävät, jäsenfunktioita, ja näin ollen niitä voi käyttää ohjelmoinnissa kuten muitakin olioita.

Rubyn yksi vahvuuksista piilee sen ohjelmistokehyksissä. Rubystä on tehty C -kielelle oma versio CRuby tai MRI Ruby sekä Javalle JRuby toteutus. .NET -ympäristölle löytyy myös oma Iron Ruby -kehitysympäristö. Kuitenkin yksi Rubyn tunnetuimmista ohjelmistokehyksistä on Ruby on Rails. (11. s. 14 – 15.)

## 4.2 Rails

Ruby on Rails on siis Ruby -kielelle toteutettu ohjelmistokehys, ja se on tarkoitettu tuottamaan dynaamisia web-sovelluksi. Ruby on Rails -ohjelmistokehyksen perusperiaatteet on antaa web-kehittäjälle helpot työkalut web-sovelluksien kehittämiseen ja testaamiseen. Rails-kehys pohjautuukin seuraaviin kolmeen ajatukseen.

”Älä toista itseäsi” tarkoitetaan sitä, että kertaalleen määriteltyä toimintoa tulisi pystyä käyttämään monessa eri paikassa, jotta sitä ei tarvitsisi kopioida aina uudelleen. Tällä pyritään vähentämään virhetilanteita, jotka voivat aiheuta samantyyppisen toiminnon ristiriitaisuudesta.

Toinen ajatus on ”tavanmukaisuus ennen virittelyä”, minkä huomaa siinä, että Rails -kehys ympäristö määrittelee automaattisesti asiat sitä mukaa, kuin niitä määritellään järjestelmään, ilman että ohjelmoijan tulisi säätää jokainen parametri ja muuttuja käsin.

Kolmantena ajatuksena on käyttää REST-mallia. REST on käytännössä HTTP -tiedonsiirron tapa siirtyä tilasta toiseen. Railsin "hyvien tapojen" mukainen tapa on toteuttaa ohjaimet REST -mallilla. REST -malli pitää sisällään seuraavat HTTP -tiedonsiirto tavat: GET, POST, PUT, UPDATE ja DELETE. GET:iä käytetään pelkästään tiedonhakuun, eli esimerkiksi sivu haku. POST:lla siirretään käyttäjän lähettämä data sovellukselle. PUT komento välittää myös datan palvelimelle kun halutaan tallettaa käyttäjän lähettämä data palvelimelle. UPDATE toiminto on hyvin samankaltainen komento kuin PUT, mutta Rails sovelluksissa sitä käytetään nimensä mukaisesti päivittämään tietoa olemassa olevasta asiasta. DELETE -komennolla halutaan Rails -sovelluksissa poistaa annetuilla parametreilla tietoa sovelluksesta. Rails sovelluksessa voi siis käyttää koko ajan vaikka esimerkiksi POST -komentoa, mutta se ei ole Railsin hyvien tapojen mukaista. Mikäli REST-malli on toteutettu oikein on uuden kehittäjän helpompi lukea komentoja, koska silloin kehittäjä tietää, mitä komennon tulisi tehdä, jos se vaikka esimerkiksi käyttää komentoa UPDATE. (12. s. 3.)

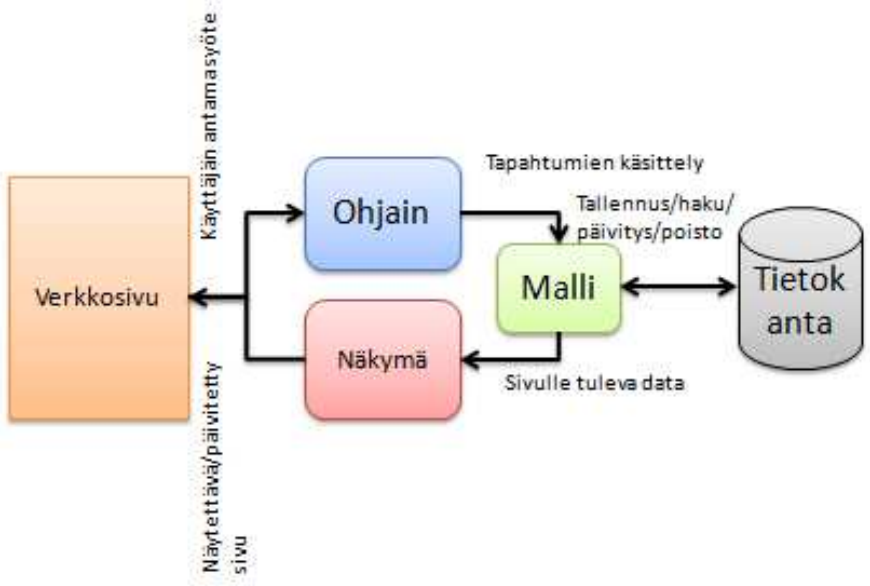
### Gem

Ruby laajennusosia kutsutaan gemeiksi. Nämä laajennukset sisältävät kirjastoja ja moduuleita. Esimerkiksi Rails gemin avulla voidaan luoda Ruby on Rails -sovelluksia.

Tämän vuoksi kun käyttäjä on asentanut Ruby -kielen koneelle, tulee asennuksen yhteydessä työkalu Rubygems, jonka tarkoituksena on hallita laajennuksia. Rubyn gemejä voi verrata esimerkiksi Javan Beanseihin.

## 4.3 MVC-suunnittelumalli

Suunnittelumalleilla tarkoitetaan yleensä samantyyppisten ongelmien ratkaisuja, jotka on muodostettu dokumentoimalla havainnot yhdeksi kokonaisuudeksi. Suunnittelumallin vakiinnuttua se muodostaa yhteisen arkkitehtuurin sovelluksen toiminnalle, joka helpottaa jatkokehitystä sekä uuden henkilön lisäämistä projektiin.(3. s. 365 – 370.)




Kuva 5. MVC -mallin arkkitehtuuri UML -kaaviona

Ruby on Rails pohjautuu MVC-suunnittelumalliin, kuten ASP.NETin ASP.NET MVC -viitekehykset tai PHP:n tunnetuin viitekehys Zend. MVC eli Model View Controller muodostuu siis kolmesta kokonaisuudesta: malli, näkymä ja ohjain kuvan 5 mukaisesti. Malli ilmentää käytettävän tiedon ja siihen liittyvän logiikan. Malliin toteutetaan siis kaikki muuttujien kelpuutusmetodit. Näkymässä määritetään käyttöliittymän ulkoasu ja mallin tietojen näyttäminen. Ohjain vastaanottaa sivulta tulevat komennot ja muuttaa mallia ja mallin tietojen avulla muuttaa myös näkymän oikeaksi, joko luomalla uuden näkymän tai päivittämällä jo olemassa olevaan mallia. (13. s. 528 – 531.)

Verkkopalvelun käyttäjä ei itse näe käytössä mitään muuta, kuin minkä näkymä antaa käyttäjälle. MVC:n toimintakierros alkaa sillä, että käyttäjä antaa syötteen näkymään, esimerkiksi käyttäjätunnuksen ja salasanan, minkä jälkeen ohjain saa näkymästä annetun tiedon ja luo tässä tapauksessa uuden käyttäjämallin. Käyttäjämallin tiedoilla tehdään tietokantakysely, ja mikäli arvot ovat oikein, ohjain luo uuden näkymän, johon käyttäjä ohjataan sisäänkirjautumisen jälkeen. Tämän tyyppisiä toimintakierroksia jatketaan niin kauan kuin käyttäjä käyttää palvelua.

# 5 MALTTI-työkalun toteutus

## 5.1 Työn vaiheistus

MALTTI-projektissa käytettäväksi kehitysmalliksi valittiin Scrum ja näin ollen myös työnvaiheistus noudattaa Scrumiin määriteltyjä ajanjaksoja.

Taulukko 1. Projektin aikataulutus

| | 2012 | 3 | 4 | | | 5 | | | | 6 | | 9 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Tapahtumat | | 29 | 4 | 20 | 3 | 14 | 16 | 31 | 4 | 11 | 21 | 19 |
| Ensitapaaminen | | | | | | | | | | | | |
| Julkaisun suunnittelukokous | | | | | | | | | | | | |
| Sisäinen aloituspalaveri | | | | | | | | | | | | |
| Tuotteen kehitysjonon kehittäminen | | | | | | | | | | | | |
| Sprint 1 suunnittelukokous | | | | | | | | | | | | |
| Sprint 1 sisäinen palaveri | | | | | | | | | | | | |
| Sprint 1 katselmointikokous | | | | | | | | | | | | |
| Sprint 1 | | | | | | | | | | | | |
| Sprint 2 suunnittelukokous | | | | | | | | | | | | |
| Sprint 2 katselmointikokous | | | | | | | | | | | | |
| Sprint 3 suunnittelukokous | | | | | | | | | | | | |
| Sprint 2 | | | | | | | | | | | | |
| Sprint 3 katselmointikokous | | | | | | | | | | | | |
| Sprint 3 | | | | | | | | | | | | |
| Projektin katselminointikokous | | | | | | | | | | | | |

Taulukon yksi mukaisesti ensitapaaminen pidettiin 29.3.2012, jonka yhteydessä tutustuttiin asiakkaan kanssa ja esiteltiin Eficoden toimintatavat, sekä käytiin tarjous

läpi. 4.4.2012 pidettiin Aalto -yliopiston tiloissa julkaisun suunnittelukokous, jossa käytiin läpi mitä projektissa tehdään.

Ennen ensimmäisen sprintin alkua oli vielä sisäinen aloituspalaveri, jossa käytiin läpi asiakkaan antama materiaali, sekä tuotteen kehitysjonon luomista varten oma palaveri, jossa luotiin alustavat käyttäjätarinat. Sprint 1 alkoi 14.5.2012 ja päättyi 31.5.2012 Ensimmäisen sprintin aikana asiakas halusi saada ensimmäiseksi käyttöliittymän.

Sprint 2 alkoi 4.6.2012 ja päättyi 11.6.2012. Toisen sprintin aikana toteutettiin joitain palvelinpuolen toimintoja sekä tehtiin muutoksia ensimmäisen sprintin aikana tehtyyn käyttöliittymään. Kolmannen sprintin aikana tehtiin tarvittavat hallintatyökalut, jotta järjestelmää voi käyttää palvelimella. Toiminnallisuudet toteutettiin 11.6 - 21.6.2012. Projektin kehitys jatkuu edelleen.

## 5.2 Käyttäjätarinat

Käyttäjätarinat ovat asiakkaan luomat ja priorisoimat. Käyttäjätarinoilla tarkoitetaan lyhyitä yleensä yhden tai muutaman lauseen mittaisia tarinoita, joissa kerrotaan, mitä käyttäjä haluaisi tehdä kyseisellä toiminnolla. Käyttäjätarinoita käytetään yleisesti ketterissä vaihejakomalleissa ja niillä pyritään määrittämään halutut toiminnot ja liiketoiminta logiikka. Tarkoituksena on tarjota vastaukset kysymyksiin kuka, mitä ja miksi lyhyellä ja helposti ymmärrettävällä tavalla. Käyttäjätarinat voidaan muuttaa käyttötapauksiksi ja käyttötapauksia voidaan kuvata käyttötapauskaavioilla. (14.)

Käyttötapauksilla on tarkoitus ilmentää järjestelmän toiminta. Jokainen käyttötapaus kuvaa yleensä vain yhtä toiminnallisuutta ja nämä käyttötapaukset määrittelevät koko järjestelmän toiminnan erillisinä tapahtumaketjuina. Käyttötapaukset kuvataan yleensä tekstillä, mutta tarvittaessa voidaan käyttää myös käyttötapauskaavioita ja muita kaavioita. Käyttötapauksissa määritellään roolit ja voidaan myös käyttää Unified Modelling Language (UML) -notaatioita kuvaamaan käyttötapauksien välisiä suhteita.

Käyttötapaukset alkavat aina jonkin käyttäjäroolin toiminnasta ja päättyvät siihen, että järjestelmä on toteuttanut halutun toiminnallisuuden. Hyvä käyttötapaus sisältää seuraavia ominaisuuksia: ymmärrettävä, asiakasvaatimuksia kuvaava, testattava, kooltaan ja sopiva tarkkuudeltaan. Ymmärrettävyydellä tarkoitetaan sitä, että asiakas ja

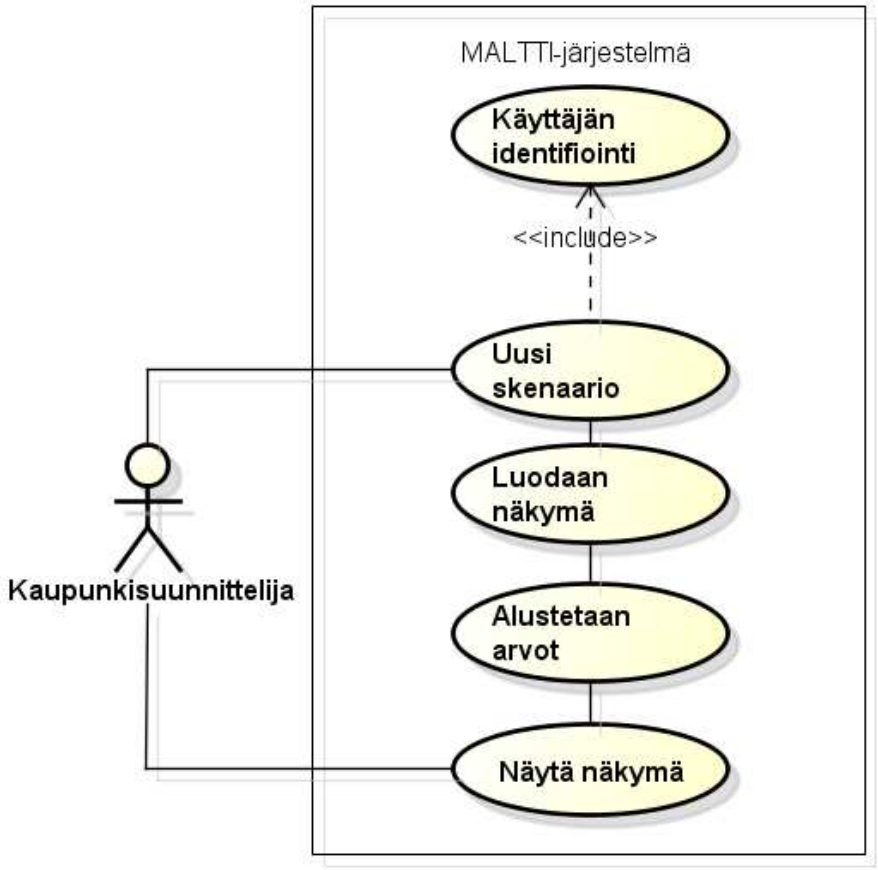
mahdolliset tulevat käyttäjät ymmärtävät käyttäjätarinan. Tähän taas vaikuttaa asiakasvaatimuksia kuvaava, sillä tarkoituksena ei ole perehtyä tekniseen toteutukseen, vaan antaa yleiskuva siitä, miten järjestelmän tulisi toimia. Käyttötapauksista tulisi voida huomata mahdolliset testitapaukset ja suorittaa testit niiden pohjalta. Käyttötapauksen fyysisen koon paperilla ei tulisi olla laaja. Yleisesti sopivana kokona pidetään yhtä A4-arkkia. Sopivalla tarkkuudella tarkoitetaan, että kaikkia yksityiskohtia ei tarvitse avata käyttötapauksissa, jotta ymmärrettävyys säilyisi. (3. s. 158 – 160.)

Maltti-projektissa toteutettiin asiakkaan kanssa yhdessä käyttäjätarinat ja kommunikointiin käytettiin JIRA -työkalua, jotta kaikki osapuolet pystyivät seuraamaan projektin kulkua ja asiakas pystyi myös lisäämään uusia käyttäjätarinoita.

Seuraavaksi esittelen projektin aikana syntyneet käyttäjätarinat, joiden kuvaamisessa olen käyttänyt käyttötapauskaavioita.

Sprint 1:n käyttäjätarina

”Kaupunkisuunnittelijana haluan nähdä mitä tietoja minun pitää syöttää ja millaisia valintoja voin tehdä työkalua käyttäessä, jotta näen suunnitteluvalintojeni vaikutuksen hiilipäästöihin.”

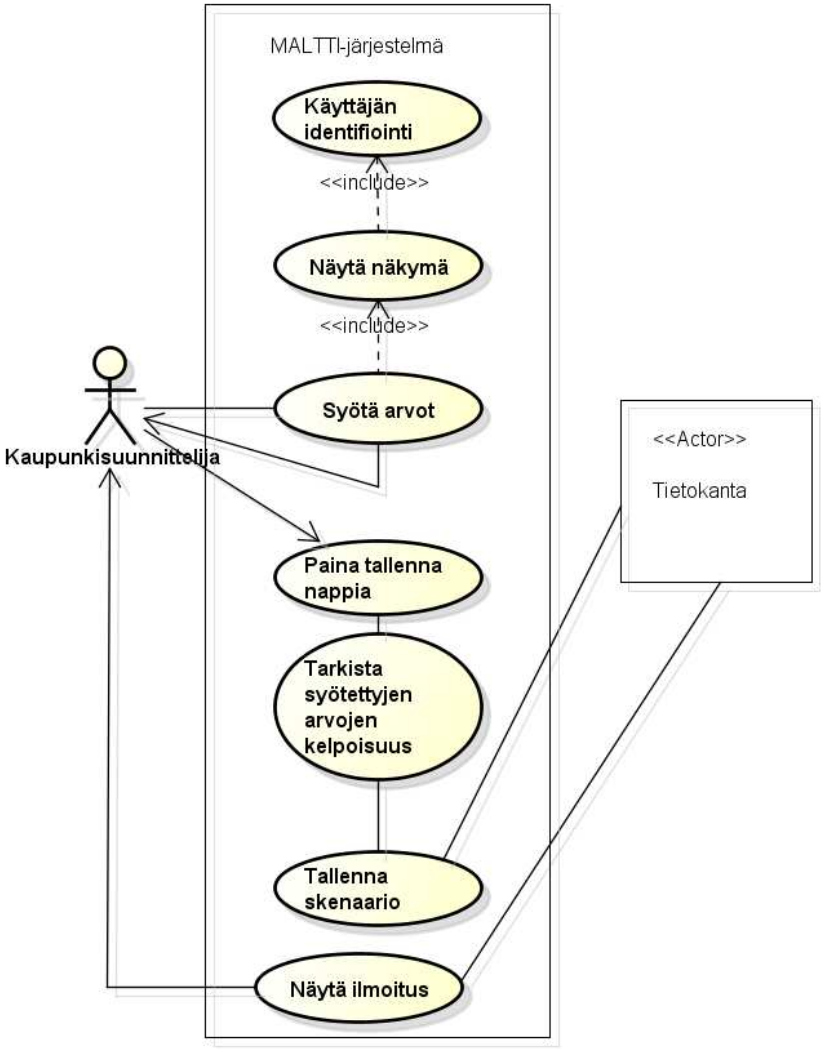



Kuva 6. Uuden skenaarion näkymän luomisen

Ensimmäisessä käyttötapauksessa (ks. kuva 6) kaupunkisuunnittelija kirjautuu aluksi ohjelmaan ja sen jälkeen valitsee uuden skenaarion luomisen. Järjestelmä luo uuden näkymän järjestelmään syötetyillä alkuarvoilla. Luotu näkymä uusien arvojen kanssa näytetään kaupunkisuunnittelijalle mahdollisilla aputeksteillä lisättynä. Käyttötapauksen alkuehto on että kaupunkisuunnittelijalla on käyttäjätili ja oikeus luoda uusia skenaarioita. Loppuehtona on, että järjestelmä palauttaa joko luodun näkymän tai virheviestin, mikäli virhetila tapahtui. Poikkeustilanne syntyy silloin, jos kaupunkisuunnittelijalla ei ole internetyhteyttä. Internetyhteys on tämän käyttötapauksen laadullinen vaatimus.

### 5.2.1 Sprint 2:n käyttäjätarinat

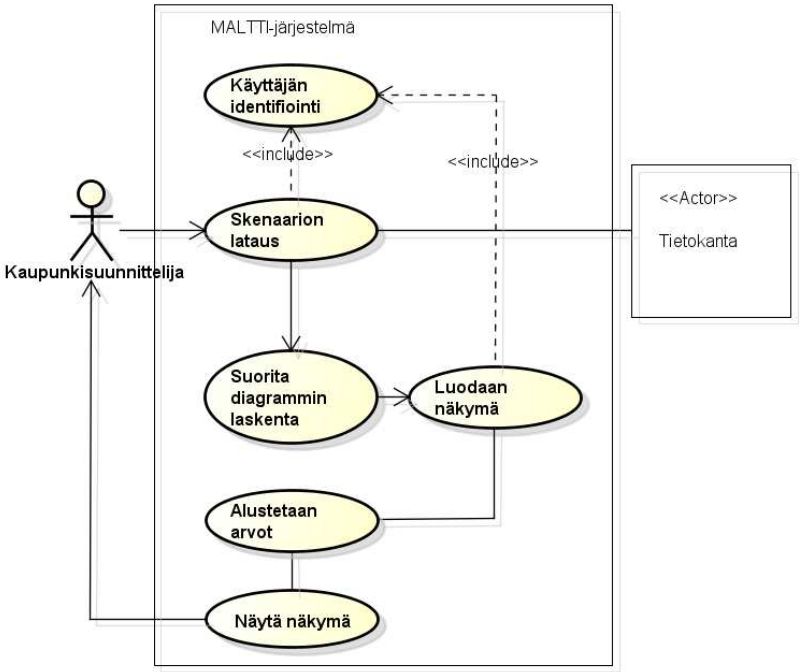
”Kaupunkisuunnittelijana haluan tallentaa muodostamani skenaarion, jotta pystyn vertailemaan sitä toiseen skenarioon tulevaisuudessa.” Tämä käyttötapauksen kulku on kuvattu kuvassa 7.




Kuva 7. Uuden skenaarion tallentaminen

Kaupunkisuunnittelija kirjautuu aluksi järjestelmään, minkä jälkeen hänelle näytetään uuden skenaarion näkymä. Kaupunkisuunnittelija syöttää arvot näkymään ja painaa tallenna -nappia, jonka jälkeen järjestelmä tarkistaa annettujen arvojen kelpoisuuden. Mikäli arvot eivät täyttäneet ehtoja, annetaan käyttäjälle vikatila -viesti. Arvojen ollessa oikeanlaisia ne tallennetaan tietokantaan, jonka jälkeen annetaan käyttäjälle viesti tallennuksen onnistumisesta. Alkuehtona on tälle käyttötapaukselle se, että kaupunkisuunnittelija syöttää tarvittavat arvot järjestelmään ja painaa tallennus -nappia. Käyttäjälle tulee loppuehtona näyttää viesti joko tiedon tallentumisesta tai tiedon tallentamisen epäonnistumisesta.

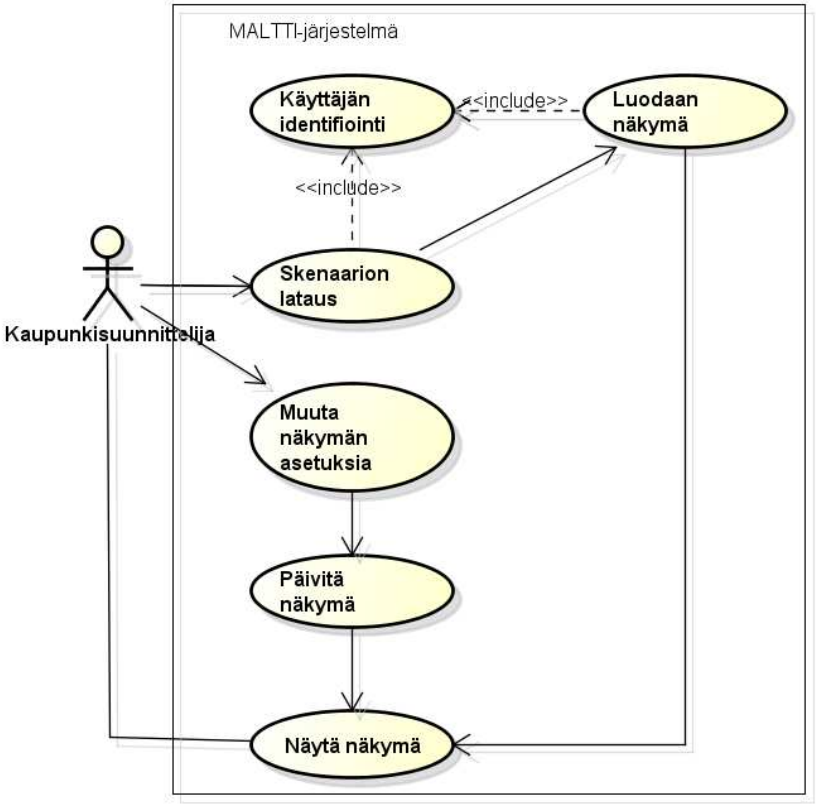
”Kaupunkisuunnittelijana haluan nähdä graafisena diagrammina asuinalueen elinkaaren hiilipäästöt, jotta näkisin suunnitteluvalintojeni vaikutukset helposti”. Tämä käyttötapaus on havainnollistettu kuvassa 8.




Kuva 8. Graafisen diagrammin lisääminen näkymään

Kirjauduttuaan hänen tulee valita tai tallentaa uusi skenaario ennen kuin järjestelmä voi suorittaan laskennan. Valittu tai luotu skenaario ladataan tietokannasta ja skenaarion arvojen avulla suoritetaan laskenta, joilla lisätään kuvaaja näkymään, minkä jälkeen näkymä näytetään käyttäjälle. Ennen kuin käyttötapausta voidaan lähteä suorittamaan, tulee alkuehtona olla jo olemassa skenaario, joka voidaan ladata tietokannasta. Lopuksi näytetään käyttäjälle saatu tulos, ja mikäli tapahtui virhe, ilmoitetaan myös siitä käyttäjälle. Laadullisina vaatimuksina onkin, että skenaario tulee olla olemassa, ennen kuin päästoarvot kertova diagrammi voidaan toteuttaa. Poikkeustilanteen voi aiheuttaa tietokantayhteys tai se, ettei skenaarioon ole tallentunut kaikkia arvoja ja tämän vuoksi diagrammin luominen ei onnistu.

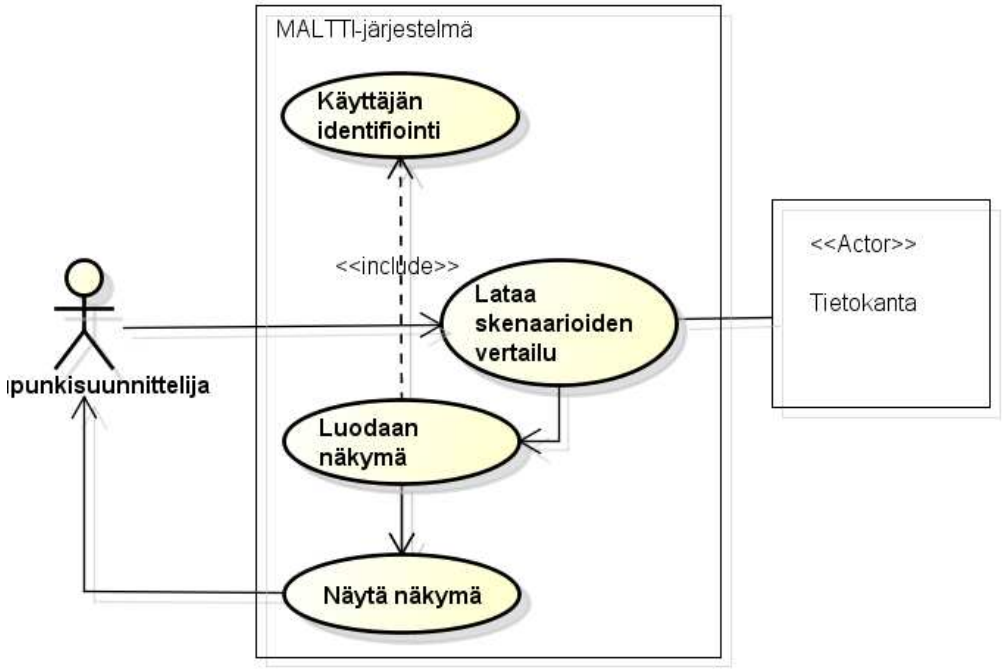
”Kaupunkisuunnittelijana haluan muokata skenaariota kattavasti, jotta voin vertailla rakentamisen ja/tai alueen käytön vaikutuksia hiilipäästöihin erilaisilla kuluttajaprofiilivalinnoilla sekä vuositasolla että pidemmällä aikavälillä.”




Kuva 9. Näkymän asetuksien muuttaminen

Kaupunkisuunnittelija lataa haluamansa skenaarion, minkä jälkeen hänelle luodaan ja näytetään näkymä kuvan 9 mukaisesti. Tämän jälkeen kaupunkisuunnittelija voi vaihtaa asetuksia, kuten tarkasteluaikavälin, näytettävien kategorioiden määrän, eri diagrammeissa. Alkuehtona on että käyttäjällä on olemassa skenaario, jotta näkymän asetuksia voi vaihtaa. Käyttäjälle näytetään lopuksi päivitetty näkymä tai virheilmoitus, mikäli järjestelmässä tuli suorituksen aikana häiriötä. Poikkeustilanne syntyy, jos käyttäjä ei anna näkymän asetuksia oikeanlaisena.

”Kaupunkisuunnittelijana haluan ladata aikaisemmin tallentamiani kahden skenaarion välisiä vertailuja, jotta minun ei tarvitse tehdä samoja skenaarioita uudestaan jatkaessani samojen skenaarioiden työstämistä.” Käyttötapauksen kulku on havainnollistettu kuvassa 10.

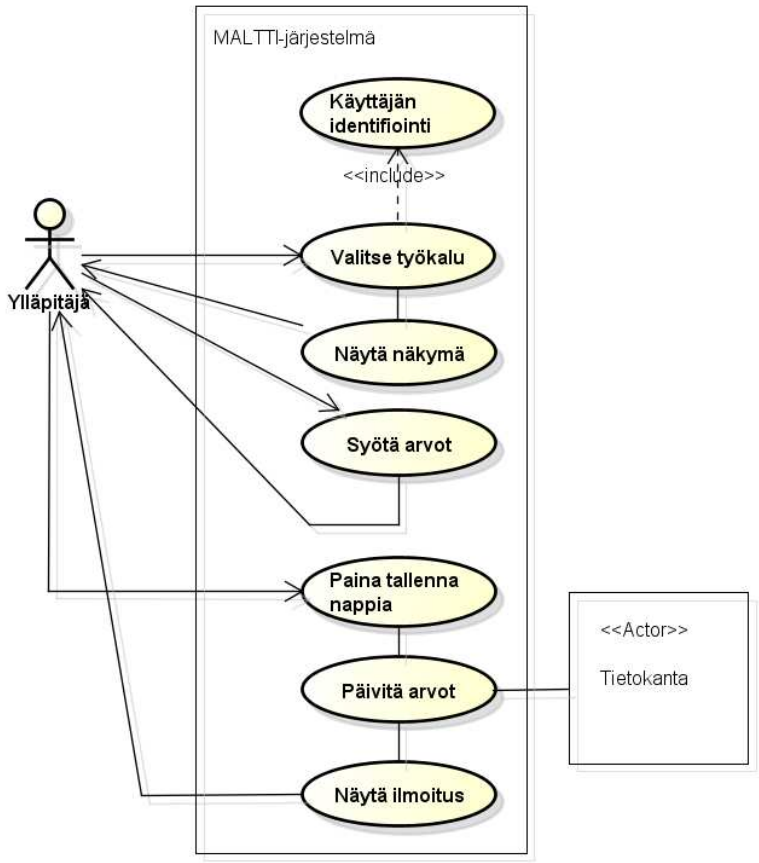



Kuva 10. Skenaarioiden vertailun lataaminen.

Kaupunkisuunnittelija kirjautuu sisään järjestelmään ja valitsee tallentamansa vertailun. Vertailu ladataan tietokannasta ja vertailuun tallennettujen skenaarioiden arvojen perusteella luodaan näkymä ja näytetään se käyttäjälle. Ennen kuin vertailun voi ladata, tulee kaupunkisuunnittelijalla olla olemassa kaksi eri skenaariota ja niistä tallennettuna vertailu tietokantaan. Järjestelmä näyttää lopuksi kaupunkisuunnittelijalle vertailunäkymän tai virheilmoituksen. Vertailun lataaminen voi epäonnistua, mikäli toinen tai molemmat skenaariot ovat tallentuneet tietokantaan väärin ja tarvittavia arvoja ei saada näkymälle.

Sprint 3:n käyttäjätarinat

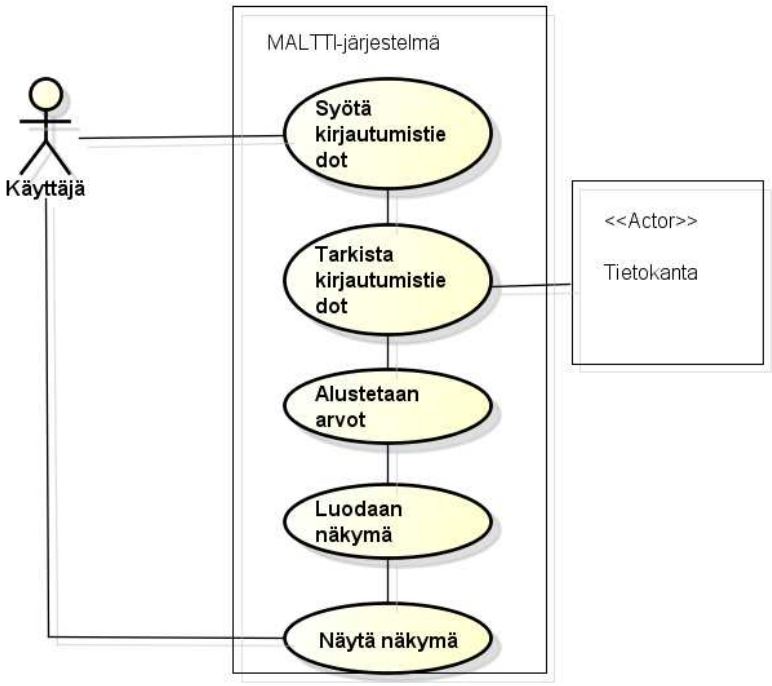
”Ylläpitäjänä haluan muokata päästölukuja, jotta hiilipäästöjen laskenta perustuu ajantasaiseen tietoon.” Käyttötapaus on havainnollistettu kuvassa 11.




Kuva 11. Päästölukujen arvojen päivittäminen.

Ylläpitäjä aluksi kirjautuu järjestelmään ja hän saa ylläpidon työkalut järjestelmään. Seuraavaksi hän valitsee haluamansa työkalun, jotta voi päivittää järjestelmään oikeat arvot. Ylläpitäjä syöttää uudet arvot järjestelmään ja painaa tallenna -nappia, jolloin tiedot päivitetään tietokantaan. Päivittämisen jälkeen ylläpitäjälle ilmoitetaan joko onnistumisesta tai epäonnistumisesta. Alkuehtoina käyttötapaukselle on, että käyttäjällä tulee olla ylläpitäjän oikeudet, jotta hän voi käyttää tarvittavia työkaluja. Loppuehtona järjestelmän tulee ilmoittaa käyttäjälle, oliko päivittäminen onnistunut vai ei.

”Ylläpitäjänä haluan, että kukaan muu ei pääse käsiksi ylläpitäjäroolin toiminnallisuuteen, jotta järjestelmään ei pääse tekemään ei-haluttuja muutoksia.”

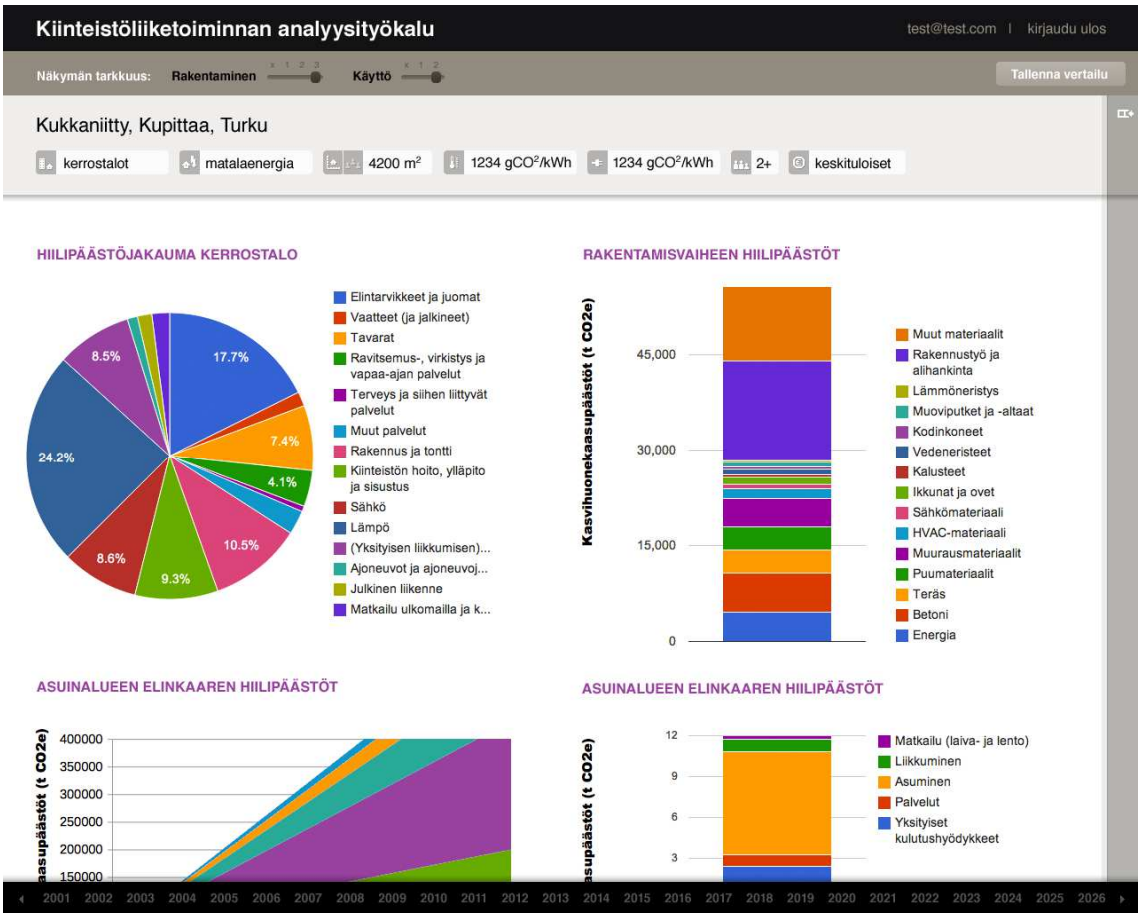



Kuva 12. Käyttäjän identifiointi.

Käyttäjä syöttää kirjautumistiedot, jolloin järjestelmä tarkastaa tietokannasta käyttäjän. Luodaan näkymä annettujen arvojen perusteella ja näytetään käyttäjälle. Käyttäjällä tulee olla käyttäjätili, ennen kuin voi käyttää sovellusta. Lopuksi käyttäjälle näytetään uusi näkymä kirjautumisen jälkeen, mikäli kirjautuminen onnistui. Muuten käyttäjä ohjataan takaisin kirjautumissivulle ja ilmoitetaan kirjautumisen epäonnistumisesta kuvan 12 mukaisesti. Mikäli käyttäjätiliä ei ole tai käyttäjä ei muista oikeita kirjautumistietoja järjestelmää, syntyy poikkeustilanne. Laadullisina vaatimuksina tälle käyttötapaukselle on, että käyttäjällä tulee olla käyttötili järjestelmässä ja että käyttäjä muistaa tilille oikeat tiedot järjestelmän.

## 5.3 Käyttöliittymä

MALTTI-projekti aloitettiin käyttöliittymän suunnittelulla ja toteutuksella. Käyttöliittymästä piti saada mahdollisimman yksinkertainen. Käyttöliittymän toteuttamiseen käytettiin jQueryä ja Haml Ruby gemiä. Haml on kevyt merkintäkieli ja muodostuu sanoista HTML Abstraktion Markup Language. Hamlia käytetään helpottamaan www-sivujen näkymien tekemistä. Hamlin tarkoitus on saada merkintä helpommaksi ja käyttäväystävällisemmäksi lukea. (15.)




Kuva 13. Luonnos uuden skenaarion luomisen näkymästä.

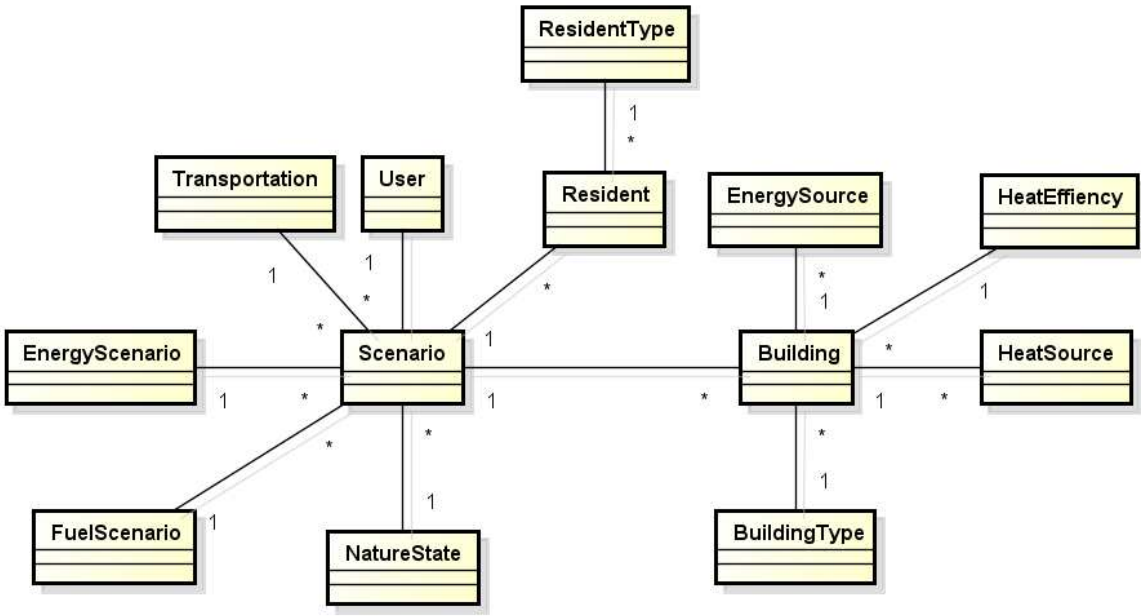
Hankalimpia asioita käyttöliittymän toteuttamisessa ja suunnittelussa oli saada mahtumaan kaikki tarvittavat muuttujat näkymään samalle näytölle. Lisäksi skenaarion toteuttaminen vaati dynaamisten listojen luomista, sillä asiakas halusi saada useita talo- ja asukastyyppejä määriteltyä samaan skenaarioon. Järjestelmän graafiset diagrammit päätettiin toteuttaa Googlen tarjoamilla Google Chart Tools -työkaluilla. Ruby on Railsiin löytyi oma gemi nimeltä GoogleVisualr, jonka avulla pystyy luomaan diagrammeja suoraan Rubylla.

## 5.4 Tekninen toteutus

### Mallit

Ruby -mallit voidaan linkittää toisiinsa assosiaatioilla. minkä takia niitä voi myös kuvata luokkakaaviolla. Mallit siis sisältävät tarvittavan tiedon, sekä liiketoiminta logiikan. Mallien pohjalta voikin siis toteuttaa suoraan tietokannan ja näin ROR -järjestelmässä

tietokannan luontikin toimii. ROR -malleja voi joko käsin kirjoittaa tai sitten käyttää komentoriviltä komentoa *rails –generate model*. Tälle komennolle annetaan parametreinä mallin nimi ja lopuksi kaikki tarvittavat arvot muotoa arvo:arvotyyppi. Esimerkiksi tässä projektissa käytetty malli luotiin komennolla *rails –generate model HeatSource value:float name:string*. Komentoa käyttämällä Rails loi db/migrate kansioon tiedoston jota käytetään komentoriviltä *db:migrate* komennolla ja täten tietokantaan luotiin taulu heat_sources ja kenttinä id, value, name, created_at ja updated_at.




Kuva 14. MALTTI –järjestelmän luokkakaavio.

ROR-järjestelmissä mallien väliset assosiaatiot luodaan siten, että mallin koodiin lisätään esimerkiksi rivi has_one :malli. Kuvassa 14 on kaikki MALTTI-järjestelmässä käytetyt mallit ja esimerkiksi User -ja Scenario -mallien välinen yhteys toteutettiin siten, että User malliin lisättiin rivi has_many :scenarios ja Scenario malliin belongs_to :user. Tämän vuoksi ROR ymmärtää että User -ja Scenario -mallin välillä on yhden suhde moneen -assosiaatio. Tämän vuoksi voidaankin ohjelmassa käyttää esimerkiksi komentoa user.scenarios.all, joka antaa kaikki käyttäjän luomat skenaariot listana.

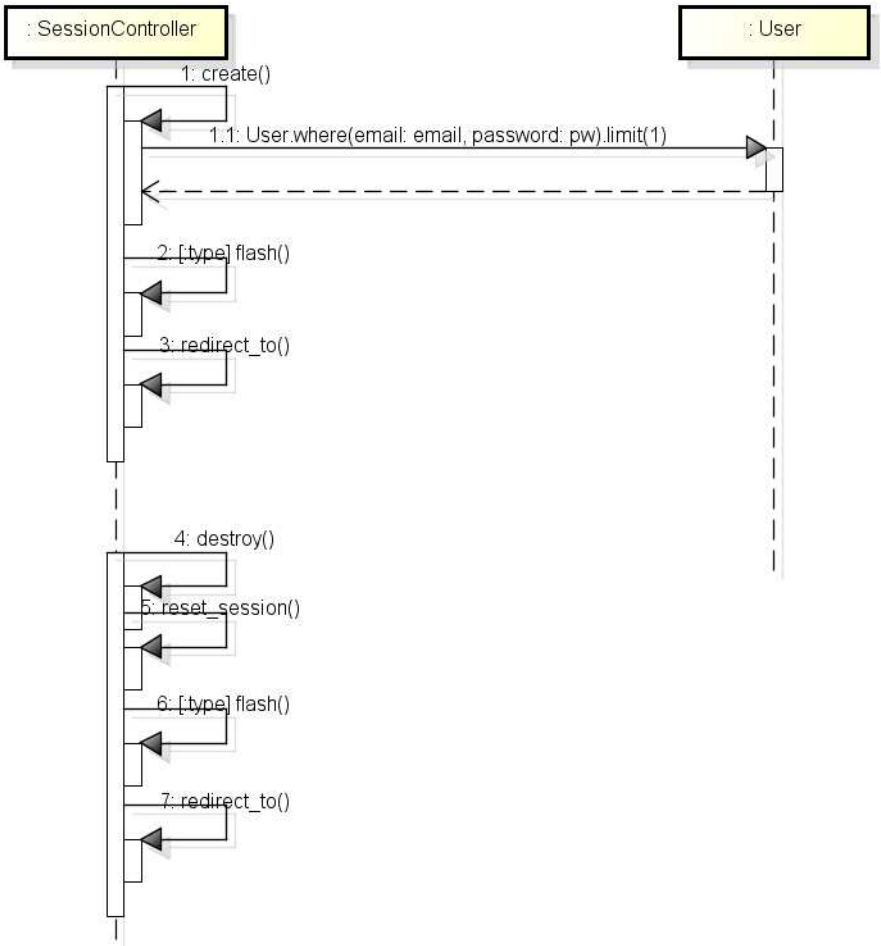
MALTTI -järjestelmän tarkoituksena on voida simuloida maksimissaan 50 vuoden välisenä aikana syntyvät hiilijalanjälkeen vaikuttavat päästöt rakentamisen ja käytön osalta. Tämän vuoksi Scenario -malli on koko järjestelmän ydin ja siihen on yhdistetty

seuraavat mallit: EnergyScenario, FuelScenario, NatureState, Transportation, HeatEffiency, Building ja Resident.

EnergyScenario, FuelScenario ja NatureState mallit sisältävät tarvittavat arvot, jotta mahdolliset paranemisskenaariot voidaan simuloida aluekaaviossa. Resident ja Building taas sisältävät tarvittavat arvot laskemaan alueen rakentamisen ja käytön aikana syntyneet päästöt. Resident sisältää vain viittuaksen ResidentType malliin, jotta järjestelmä osaa hakea oikeantyyppisten asukkaiden tiedot järjestelmään laskettavaksi. Building -malli oli hieman monimutkaisempi, sillä rakennuksiin vaikuttaa lämpötehokkuus, lämmitystapa, sähköntuotantotapa ja rakennuksen tyyppi.

Kontrollerit

Ruby On Rails käyttää MVC-arkkitehtuuria, joten sen mukaisesti jokaisella mallilla on myös oma kontrolleri eli ohjain. Esittelen tässä vain tärkeimmät kontrollerit, sillä muut kontrollerit sisältävät vain CRUD-metodit. Järjestelmän tärkein kontrolleri onkin charts_controller.rb, joka käsittelee skenaarioita. Toiseksi tärkein kontrolleri on session_controller.rb, jonka tehtävänä on luoda istunto järjestelmän käyttöä varten. Kontrollereiden toiminnan kuvaamiseen käytän sekvenssikaavioita.




Kuva 15. Session kontrollerin skenvenssikaavio.

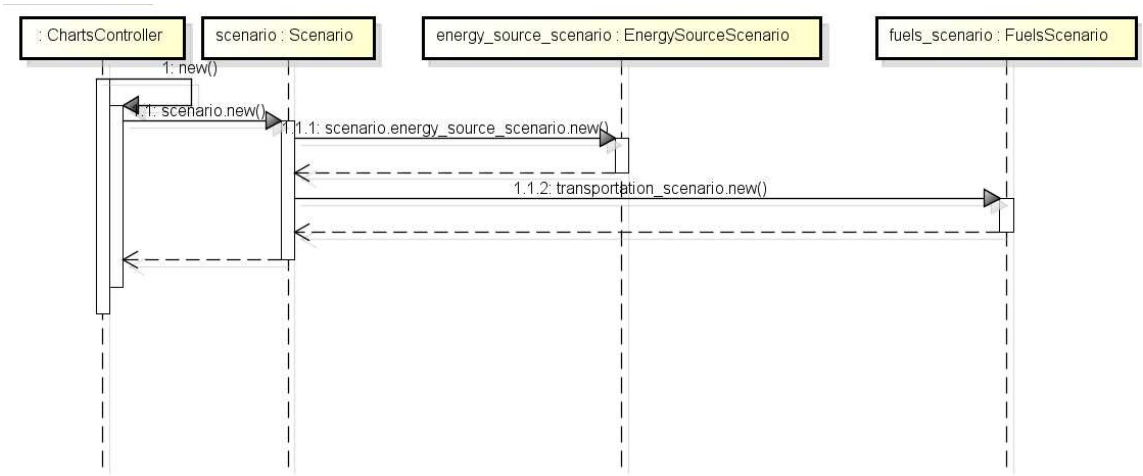
Session kontrorrelin toimintaperiaate on kuvattu kuvassa 15 sekvenssikaaviolla. Session kontrolleri sisältää kaksi metodia: create ja destroy. Molempia metodeita kutsutaan post http -metodilla, sillä esimerkiksi create -metodissa luodaan sisäänkirjautuminen. Create -metodia kutsuttaessa kontrolleri tallettaa email- ja pw-muuttujiin käyttäjän syöttämät arvot ja salasana muuttuja muutetaan md5 -muotoon.

Tarvittavien tietojen tallennuttua metodista kutsutaan User -mallin where -kutsua, joka muodostaa tietokantaan kyselyn. Kyselyyn annetaan parametreiksi sähköposti, salasana ja ehtona annetaan myös, että kysely hyväksyy vain yhden rivin. Tietokannan vastattua tarkistetaan käyttäjän tiedot ja se, että käyttäjä löytyi annetuilla tiedoilla. Mikäli käyttäjää ei löytynyt annetaan flash -metodille arvo, minkä takia sisäänkirjaus epäonnistui ja siirretään käyttäjä takaisin sisäänkirjautumissivulle.

Mikäli käyttäjä löytyi, annetaan flash metodille arvo sisäänkirjautumisen onnistumisesta ja siirretään käyttäjä etusivulle, mistä käyttäjä voi valita haluamansa toiminnon

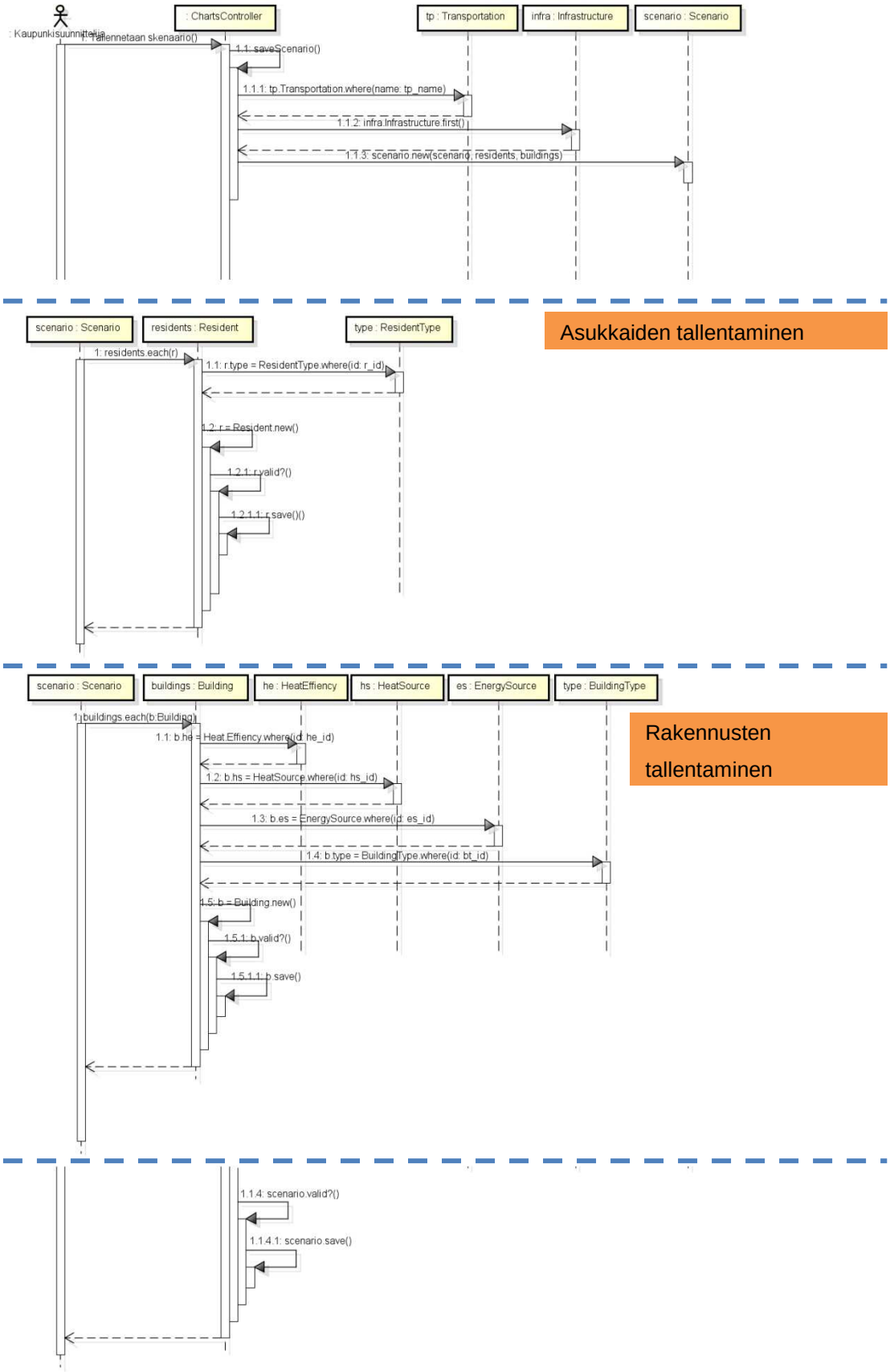
jatkaakseen järjestelmän käyttämistä. ROR:n flash -metodi näytetään vasta siirtymisen jälkeen ja sille voi asettaa oletuksena kolme erityyppistä ilmoitusta: error, alert ja notice. MALTTI-projektissa käytetään lähinnä error- ja notice-tyyppejä, joissa error sisältää virheviestin ja notice viestin onnistumisesta.

Skenaarioiden luomisen, päivittämiseen ja hakemiseen tarkoitettu chart_controller.rb on sovelluksen koko ydin. Tässä kontrollerissa suoritetaan laskentaan tarvittavat metodit ja keskustelu jQueryn kanssa, jotta tarvittavat diagrammit saadaan toteutettua näkymään. Kontrolleri sisältää seuraavat metodit: new, saveScenario, updateScenario, loadScenario, loadComparison, dataTable, saveComparison, updateComparison, deleteScenarios, deleteComparisons, printScenario, printComparison, areaChartLiving2, areaChartLiving1, areaChartLiving0, areaChartBuilding0, livingPie2, livingPie1, livingColumn2, livingColumn1, buildingPie3, buildingPie2, buildingPie1, buildingColumn3, buildingColumn2, buildingColumn1 ja buildingColumnBySquare.




Kuva 16. Uuden skenaarion luominen.

Käyttäjän luodessa uutta skenaariota (ks. kuva 16) kutsutaan osoitteen parametreilla /charts/ get -metodia, joka kutsuu charts_controller -tiedoston new -metodia, jossa luodaan scenario-olio ja scenario -muuttujan sisälle luodaan EnergySourceScenario- ja FuelsScenario- oliot. Tämän jälkeen käyttäjälle näytetään näkymä.

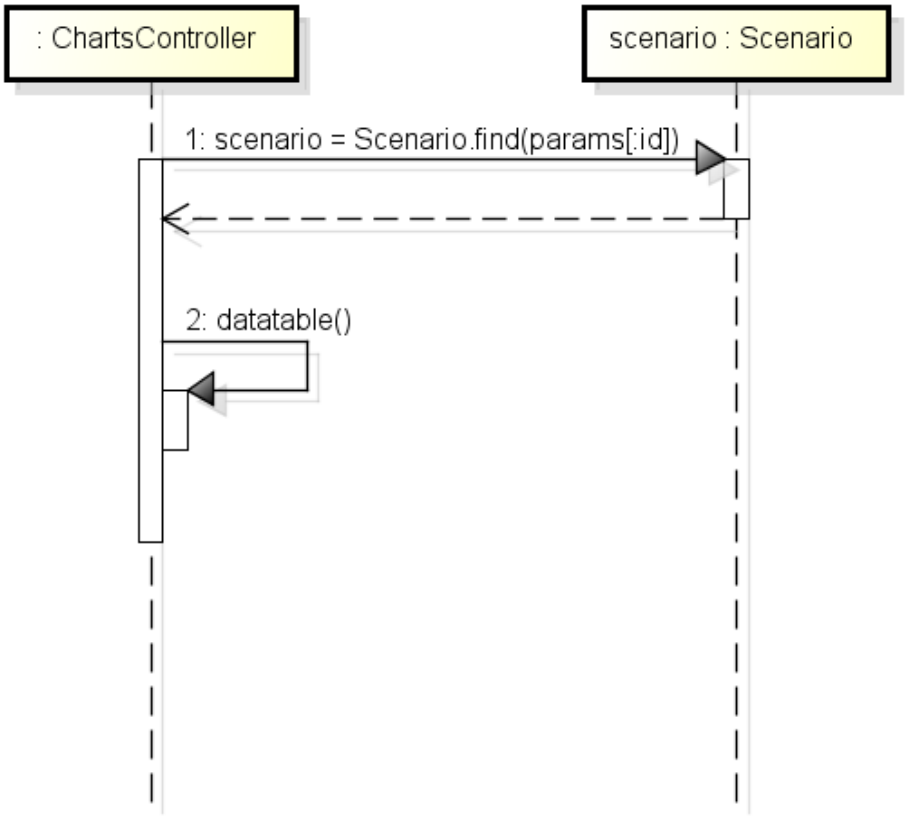



Kuva 17. Skenaarion tallentaminen järjestelmään.

Käyttäjän syötettyä kaikki tarvittavat arvot ja painettua tallenna -nappia, kutsutaan saveScenario metodia kuvan 17 mukaisesti. Asetettua alkuarvot käydään läpi kaikki skenaariota varten luodut asukkaat. Asukkaiden tallentamista varten luodaan ResidentType-olio. Alustettua asukkastyypistä kaikki tarvittavat tiedot, mukaan lukien asukastyypin indeksi arvo, tarkistetaan mallin valid? -nimisellä metodilla asukkastyypin kelpoisuus.

Arvojen ollessa oikein tallennetaan käyttäjä tietokantaan. Mikäli kaikki arvot eivät olleet oikein, siirrytään takaisin skenaarion luomisnäkymään ja näytetään käyttäjälle virheviesti. Kaikki skenaarion asukkaat käydään samalla tavalla läpi.

Rakennusten tallentaminen tapahtuu samalla tavalla, paitsi että joudutaan luomaan lämpötehokkuudesta, lämmöntuotantotavasta, sähköntuotantotavasta ja rakennuksen tyypistä omat oliot, jotta saadaan kaikki tarvittavat tiedot tallennettua rakennuksista. Kaikkien asukkaiden ja rakennusten ollessa kelpoisia järjestelmä lopuksi tallentaa skenaarion tietokantaan ja siirtää käyttäjän ladatun skenaarion näkymään. Skenaarion päivittäminen tapahtuu updateScenario -metodilla ja toimii samalla tavalla kuin saveScenario-metodi, mutta sitä kutsutaan PUT HTTP -metodilla ja kun saveScenariossa käytettiin mallien new- ja save -metodia, niin updateScenarion -malleissa käytetään find- ja update_attributes -metodeita.




Kuva 18. Skenaarion lataaminen.

Käyttäjän painettua tallenna –nappia, tai alkuvalikosta valittu tietty skenaario siirrytään ladatun skenaarion näkymään, jossa kutsutaan kontrollerin loadScenario -metodia (ks kuva 18). Metodissa ensimmäisenä haetaan tietokannasta find-metodin avulla oikea skenaario, minkä jälkeen tarkistetaan, onko käyttäjä sama henkilö, joka on luonut myös ladatun skenaarion. Käyttäjän ollessa sama henkilö kuin skenaarion luonut, suoritetaan kontrollerin datatable -metodi, jossa tarkistetaan näkymän asetukset ja kutsutaan sen mukaan seuraavia metodeja: areaChartLiving2, areaChartLiving1, areaChartLiving0, areaChartBuilding0, livingPie2, livingPie1, livingColumn2, livingColumn1, buildingPie3, buildingPie2, buildingPie1, buildingColumn3, buildingColumn2, buildingColumn1 ja buildingColumnBySquare. Näissä metodeissa luodaan tarvittavat taulukot diagrammeja varten ja annetaan ne jQuery -koodille suoritettavaksi näkymän puolella.

Vertailuja varten on omat näkymät, joissa ladataan kaksi skenaariota vierekkäin ja kun käyttäjä haluaa tallentaa vertailun, kutsutaan kontrollerin saveComparison -metodia, jossa aluksi molemmat skenaariot päivitetään siltä varalta, että käyttäjä on halunnut muokata skenaarioiden arvoja ja sen jälkeen tallentaa tietokantaan viitteet molemmista

skenaarioista. Näiden viitteiden avulla käyttäjä voi alkuvalikosta valita vain oikean linkin ja siirtyä suoraan tarkkailemaan tehtyjä skenaarioita. Tulostusnäkymässä haetaan vain tarvittavien skenaarioiden tiedot, mutta ei näytetä ollenkaan valikkoa, jolla voi muokata skenaarioita. Riippuen haluaako käyttäjä tulostaa vain yhden skenaarion tiedot vai tulostaa vertailun tiedot, kutsutaan kontrollerista printScenario- tai printComparison -metodeja.

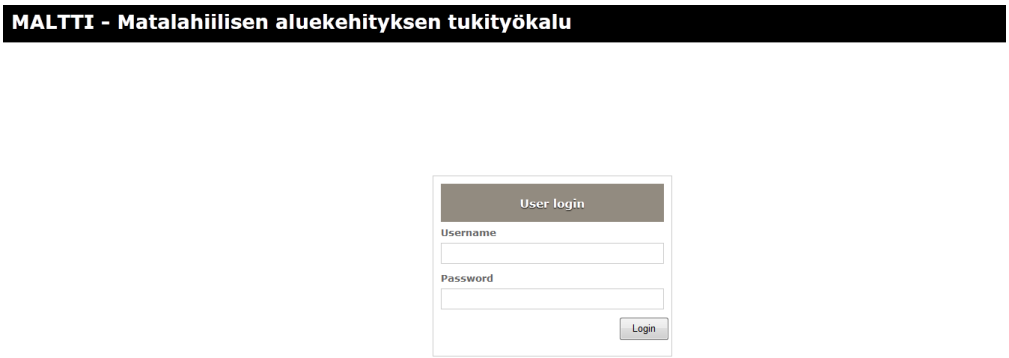
Näkymät

Ruby on Rails -järjestelmässä näkymät voidaan toteuttaa useilla eri tekniikoilla. Tässä kyseisessä projektissa näkymät toteutettiin Hamlin ja jQueryn avulla. Näkymien polut määritellään config -kansiossa olevaan routes.rb -tiedostoon. Kyseiseen tiedostoon määritellään myös, se mitä kontrolleria ja mitä kontrollerin metodia halutaan kyseisessä näkymässä suorittaa.

```
1  resources :nature_states
2  resources :ifranstructures
3  match 'login' => "sessions#create", :via => :post
```

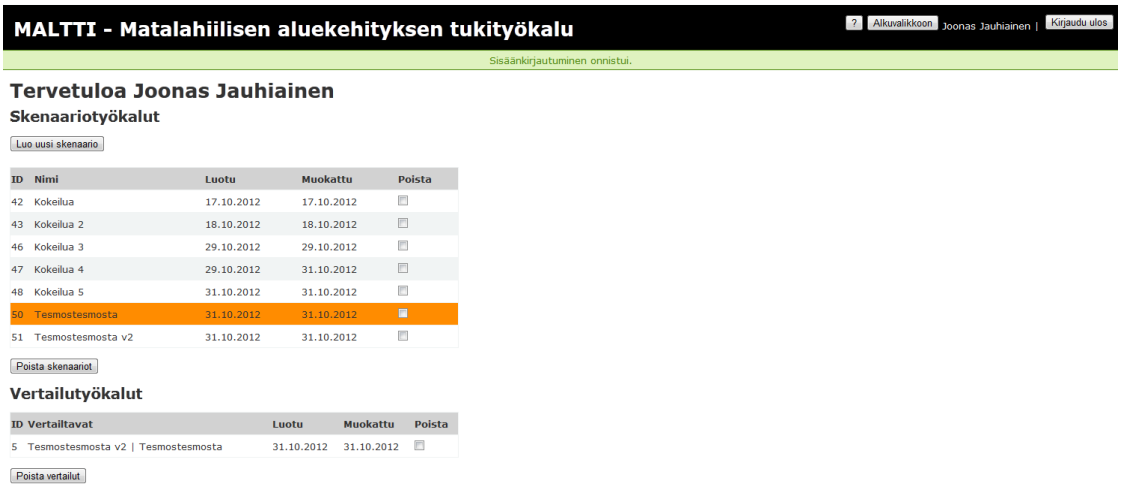
Kuva 19. Reititykset route.rb -tiedostoon.

Resources -komento luo valmiiksi tarvittavat metodikutsut komennolle annetun parametrin avulla. Parametriksi annetaan siis kyseinen kontrolleri, jota halutaan käyttää. Reittejä voi myös määrittää match -komennolla, esimerkiksi login -kutsu on MALTTI-järjestelmässä merkitty käyttämään session_controller -luokan create -metodia käyttäen HTTP POST -metodia. Reititykset voi nähdä komentoriviltä *rake:routes* -komennolla. MALTTI-järjestelmän "etusivuksi" on määritetty home kontrollerin index -metodi, jolloin käyttäjä ohjataan suoraan home -kontrollerin näkymään, jossa suoritetaan index -metodi. Kuvassa 19 näkyy käytössä oleva sisäänkirjautumisen näkymä.



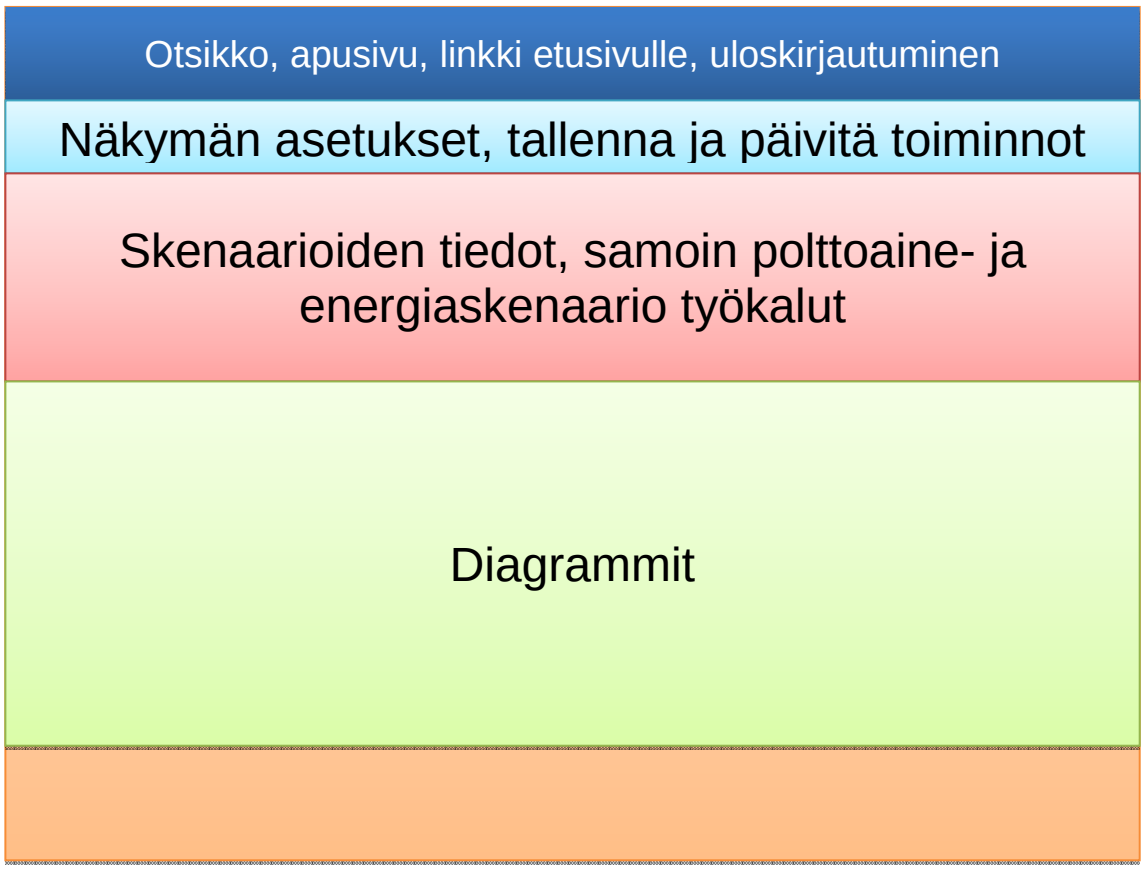

Kuva 20. MALTTI-järjestelmän sisäänkirjautuminen.

Järjestelmään kirjautuessaan käyttäjä ohjataan aloitusvalikkoon, jossa on käyttäjätasosta riippuen eri työkaluja käytettävissä (ks. kuva 20). Käyttäjälle myös näytetään kirjautumisen onnistuminen viesti, joka näkyy vihreällä pohjalla tummanvihreällä tekstillä.



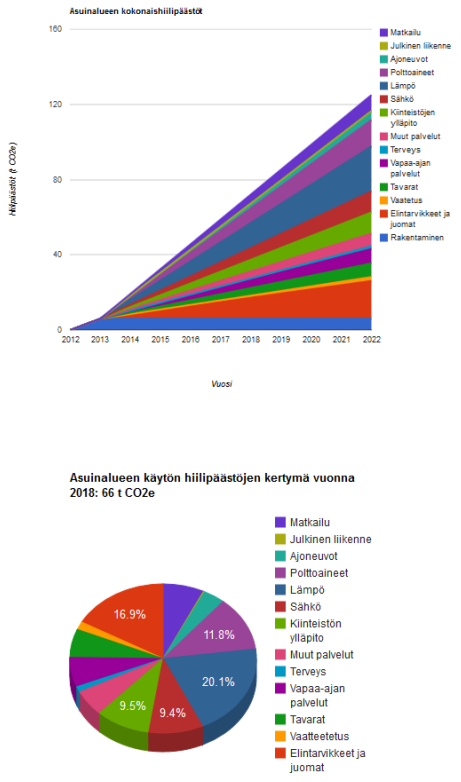

Kuva 21. MALTTI-järjestelmän alkuvalikko kaupunkisuunnittelijan roolissa.

Alkuvalikosta kaikilla käyttäjärooleilla on käytössä skenaariotyökalut ja vertailutyökalut kuvan 21 mukaisesti. Skenaariotyökaluilla käyttäjä voi luoda uuden skenaarion, poistaa yhden tai useamman skenaarion tai valita tietyn skenaarion tarkasteltavaksi. Mikäli käyttäjä haluaa poistaa skenaarion, jota käytetään vertailuissa poistetaan myös tallennettu vertailu, koska molemmat skenaariot tulee olla olemassa, kun skenaariota ladataan.



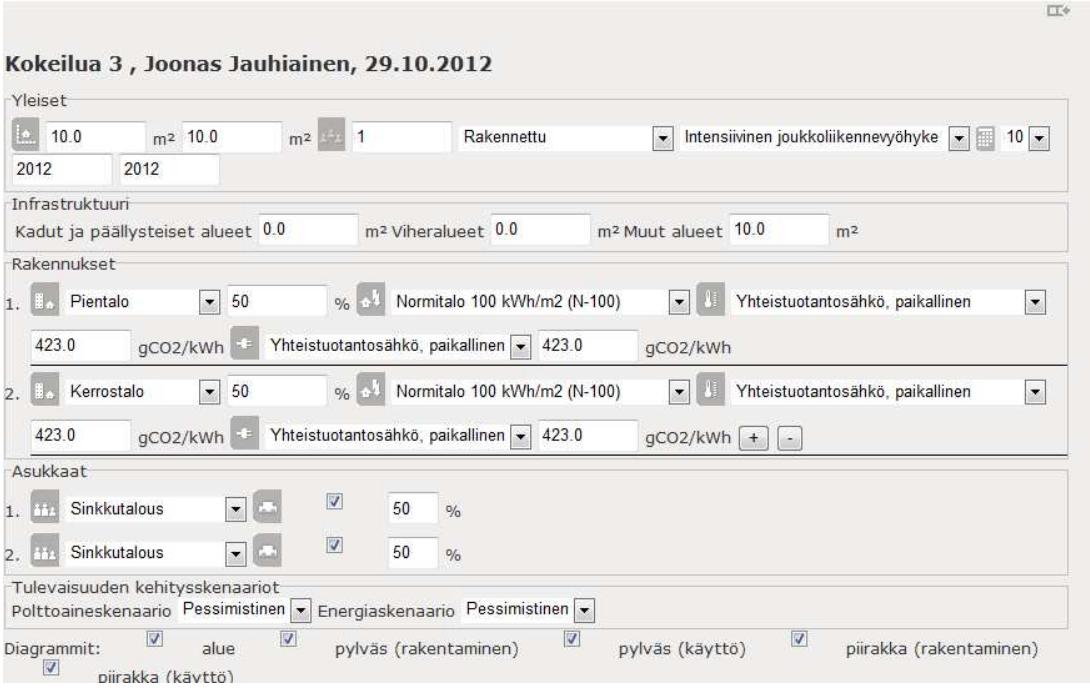

Kuva 22. Kaaviokuva skenaario näkymän elementtien asettelusta.

Skenaarion luonti, muokkaus ja tallennus on jaettu neljään osaan kuvan 22 mukaisesti. Ensimmäisenä on linkit takaisin alkuvalikkoon, käyttäjätiedot ja uloskirjautuminen. Toisena osiona on näkymän asetuksien muuttamiseen tarvittavat työkalut sekä tulosta-, tallenna- ja päivitä -toiminnot. Kolmas osio muodostuu skenaarion luomiseen ja muokkaamiseen tarvittavista kentätistä. Lopuksi käyttäjälle näytetään diagrammit, jotka ohjelma muodostaa näkymään käyttäjän tallennettua skenaarion.




Kuva 23. Esimerkki järjestelmän tuottamista diagrammeista.

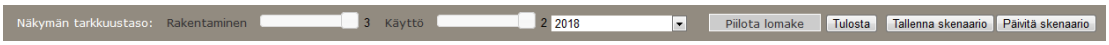
Diagrammit -osiossa ei näy muutakuin diagrammit ja niihin liittyvät tiedot. Kuvassa 23 on luotu näkymä aluekaavion diagrammista ja sen alapuolella piirakkadiagrammi käytön päästöjen jakautumisesta. Asiakkaan toiveina oli saada kolme pylväsdiagrammia, kaksi piirakkadiagrammia ja yksi aluekaavio. Alueen käyttäjien tuottamat hiilijalanjälkipäästöt ilmoitetaan aluekaaviossa sekä piirakka- ja pylväsdiagrammissa. Rakentamisen päästöillä on yksi pylväsdiagrammi, joka ilmaisee rakennustyyppikohtaisesti päästöt per neliömetri ja lisäksi kaikkien rakennustyyppien keskiarvo päästöt per neliömetri. Toinen pylväsdiagrammi ilmaisee tietyn tarkasteluvuoden päästöt rakennustyyppikohtaisesti. Piirakkadiagrammi ilmaisee myös tietyn tarkasteluvuoden päästökertymät, mutta piirakkakuviossa ei eritellä rakennustyyppejä toisistaan. Aluekaaviodiagrammi sisältää sekä rakentamisesta että käytöstä syntyneet päästöt, ja tarkasteluväli on enintään 50 vuotta ja vähintään 10 vuotta.




Kuva 24. Skenaarion muokkausosio.

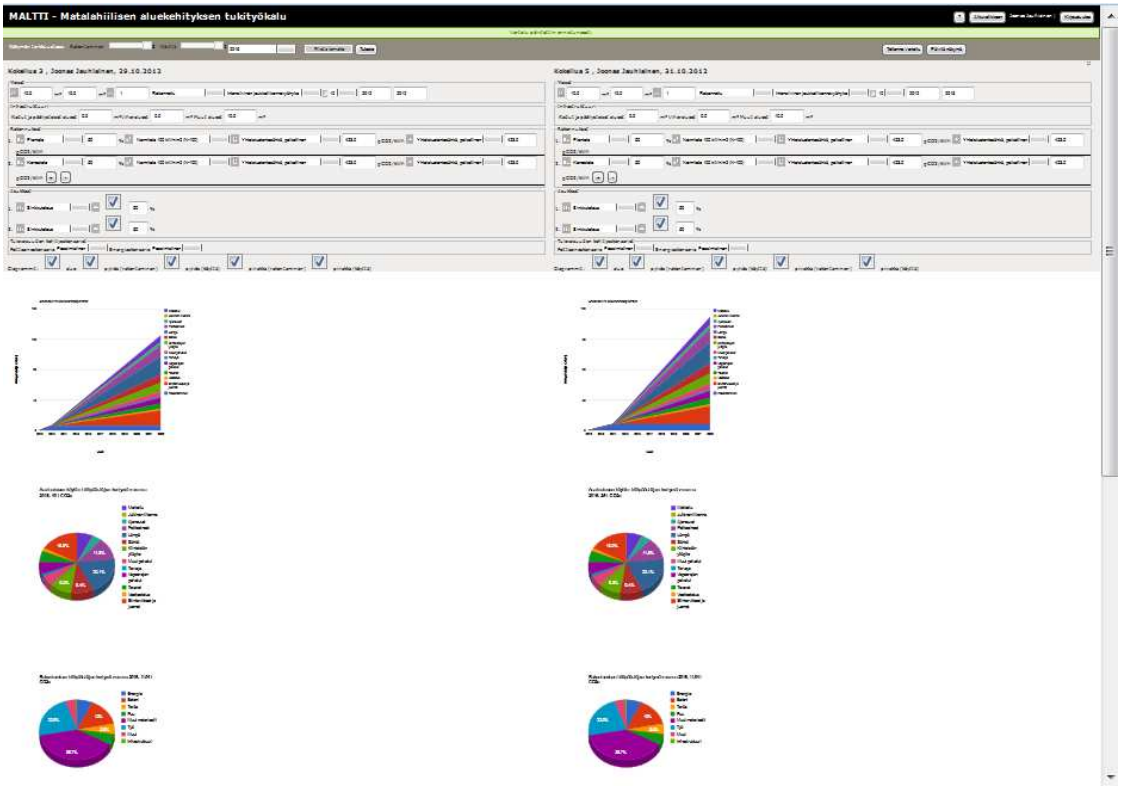
Skenaarion yleistiedot syötetään ensimmäisenä (ks. kuva 24). Ensimmäiseksi annetaan alueen nimi, jonka jälkeen syötetään alueen koko, rakennusoikeuden ala, väkiluku, alueen luonnontila, alueen liikennevyöhyketila, tarkasteluväli, rakentamisen aloitusvuosi ja lopetusvuosi. Alueen luonnontilalla tarkoitetaan onko alue jo raivattu vai tuleeko se raivata kasvustosta ja muusta ennen kuin rakentaminen voidaan aloitaa. Seuraavaksi tulee määrittää alueen infrastruktuuri, joka vaikuttavaa rakentamisen päästöihin.

Kolmantena osiona on rakennuksien määrittäminen. Rakennuksille tulee määrittää seuraavat arvot: rakennuksen tyyppi, prosenttiosuus alueen rakennusoikeudesta, lämpötehokkuus, lämmöntuotantotapa ja sähköntuotantotapa. Neljäntenä osiona määritellään alueen asukkaat, jotka ovat sidottuna aikaisemmin määritettyihin rakennuksiin. Asukkaille määritellään asukastyyppi, ja mikäli asukastyyppi ei ole keskimääräinen, asukkaille voi määritellä auton käytön ja lopuksi prosenttuaalinen osuus koko alueen väkiluvusta. Mikäli käyttäjä lisää rakennuksia alueeseen, muodostuu myös uusi asukasrivi, sillä jokaisella rakennukselle tulee määrittää oma asukastyyppi. Tulevaisuuden kehitysskenaariot -osiossa määritellään polttoaineskenaario ja energiaskenaario, joista muodostuvat vuosittaiset kertoimet.




Kuva 25. Skenaario näkmyän asetuksien muokkaustyökalu sekä muut toiminnot.

Asiakas halusi pystyä ryhmittelemään erikategorioiden arvoa sekä vaihtamaan tarkasteluvuotta, joten sitä varten piti tehdä oma työkalupalkki, josta viimeisin versio on kuvattuna kuvassa 25. Samasta työkalupalkista löytyy myös toiminnot lomakkeen piilottamiseen, skenaarion tallentamiseen ja päivittämiseen sekä tulostamiseen. Näkymän tarkkuustasot on toteutettu jQueryn omalla liukusäätimellä. Rakentamisen liukusäätimellä voidaan näyttää rakentamisen kategoriat kolme eri tasolla. Kolmas taso sisältää kahdeksan eri kategoriaa, toinen taso viisi kategoriaa ja ensimmäinen taso vain kaksi kategoriaa. Käytön liukusäätimen toisella tasolla näytetään 13 kategoriaa ja ensimmäisellä tasolla viisi kategoriaa. Tarkasteluvuoden voi valita alasvetovalikosta ja ”piiloita lomake” -napilla pystyy piilottamaan koko skenaarion muokkausosion. Asiakas halusi tämän toiminnon, jotta pienemmillä näytöillä on helpompi selata kaavioita.




Kuva 26. Vertailunäkymä.

Skenaarioiden vertailussa käytetään samaa näkymän osaa kuin yhden skenaarion tarkastelussa ja muokkaamisessa. Vertailunäkymässä molemmat skenaariot ovat vierekkäin, jotta vertailu olisi mahdollisimman helppoa (ks. kuva 26).

### 5.5 Testaus

MALTTI-järjestelmän testit on kirjoitettu käyttämällä RSpec -nimistä gemiä, joka tarjoaa paljon suuremman valikoiman määritellä testejä, kuin oletuksena tuleva gemi. Ruby On Rails -ohjelmistokehys tarjoaa valmiina testausympäristön. Oletustesteinä ROR-sovelluksissa voi ajaa vain toiminnallisia, integrointi- ja yksikkötestejä. RSpec gemin avulla testejä voi laajentaa reitityksiin, näkymiin ja avustajiin.

Testivetoista kehitystapaa käyttäen tulisi testit määrittää aina ennen kuin aletaan toimintoa kirjoittaa ohjelmallisesti. Tämän vuoksi käyttäjätarinat ovat hyväksi, sillä niiden perusteella on helppo määritellä, mitä tulisi testata, jotta järjestelmä toimisi, kuten on ajateltu. Esimerkiksi sisäänkirjautumisen tulisi aina ilmoittaa käyttäjälle onnistumisesta tai epäonnistumisesta.

```
1  describe BogusController, "handling GET to #index" do
2    it "sets flash.now[:message]" do
3      @controller.instance_eval { flash.extend(DisableFlashSweeping)
4      get :index
5      flash.now[:message].should_not be_nil
6    end
7  end
```

Kuva 27. Flash -toiminnon testaaminen RSpecillä.

Kuvassa 27 on lyhyt testi, jolla voidaan testata, lähettääkö käytetty kontrolleri flash komennon ja onko käytetty viesti jotain muuta kuin tyhjä.

## 6 Yhteenveto

Tämä työ on käsitellyt Ruby on Rails -tekniikan avulla toteutettua web-järjestelmää. Järjestelmän tarkoitus oli tuottaa Eficode Oy:n asiakkaalle Aalto -yliopistolle työkalu, joita sen asiakkaat voivat käyttää alueiden suunnittelussa ja simuloida mahdollisia hiilijalanjälkikertymiä.

Työ on sisältänyt uusiin menetelmiin ja teknologoihin perehtymistä, sekä järjestelmän toteuttamisen. Ruby on Rails -ohjelmistokehys valittiin koska se soveltuu hyvin nopeaan prototyyppien luomiseen, mikä taas soveltuu hyvin Scrum -vaihejakomallin kanssa. Projektissa haastavinta oli opetella kokonaan uusi ohjelmointikieli, ja lisäksi Ruby on Rails eroaa jonkin verran yleisimmistä www-sovelluksiin käytetyistä kielistä kuten PHP tai ASP.NET.

MALTTI-projekti oli myös ensimmäinen Scrum -vaihejakomallia hyödyntävä projekti, joka taas toi mukana asioita joihin piti ensin sopeutua. MALTTI-projektiin käytettiin kesän 2012 aikana noin kaksi kuukautta aikaa, jonka aikana suoritettiin kolme sprinttiä ja jokaisen sprintin jälkeen käytiin asiakkaan luona esittelemässä siihen mennessä tehdyt toiminnot. Kolmannen sprintin jälkeen asiakkaalle luovutettiin järjestelmä testattavaksi. MALTTI-projektin kehitys jatkuu edelleen ja asiakas on ollut tyytyväinen projektin edistymiseen.

## Lähteet


1 Kestävä asuminen ja ympäristö, K-EASY 2012. (Verkkodokumentti) Lahden Tiede- ja yrityspuisto Oy. <http://www.livingbusiness.fi/hankkeet/k-easy>. Päivitetty 7.8.2012. Luettu 11.8.2012.

2 Carbon Footprint 2012. (Verkkodokumentti) Global Footprint Network.<http://www.footprintnetwork.org/en/index.php/GFN/page/carbon_footprint/>. Päivitetty 17.7.12. Luettu 24.9.2012.

3 Haikala, I., Märijärvi J. Ohjelmistotuotanto. Jyväskylä: Gummerus Kirjapaino Oy, 2006.

4 Schwaber, K., Sutherland, J. Scrum. Scrum.org, 2011.

5 SixPagesAboutScrum(1).pdf 2012. (Verkkodokumentti) <http://wiki.metropolia.fi/download/attachments/50758582/SixPagesAboutScrum%281%29.pdf?version=1&modificationDate=1347522671000> Metropolia Ammattikorkeakoulu. Päivitetty 13.9.2012. Luettu 9.11.2012.

6 Crispin, L. Driving Software Quality: How Test-Drive Development Impacts Software Quality. Software, IEEE 6/2006, s. 70 − 71.

7 Aptana | Studio 2012. (Verkkodokumentti) Appcelerator Inc. <http://www.aptana.com/products/studio3>. Päivitetty 31.8.2012. Luettu 21.10.2012.

8 JIRA 2012. (Verkkodokumentti) Atlassian. <http://www.atlassian.com/software/jira/overview/>. Luettu 21.10.2012.

9 Deveo 2012. (Verkkodokumentti) Eficode. <https://deveo.com/>. Luettu 21.10.2012.

10 Chacon, S. Pro Git. New York: Springer-Verlag Inc, 2009.

11 Kasurinen J., Ruby on Rails -ohjelmointi. Jyväskylä: WSOYpro Oy, 2011.

12 Ibrahim, R., Razali, R. A performance-oriented interface design model of web applications. IICEI 10.1109/2011, s. 1 − 6.

13 Freeman E., Freeman E., Sierra K, Bates B. Head First Design Patterns. Yhdysvallat: O'Reilly Media Inc, 2004.

14 Project Advantages of User Stories as Requirements | Mountain Goat Software 2012. (Verkkodokumentti) <http://www.mountaingoatsoftware.com/articles/advantages-of-user-stories-for-requirements> Mountain Goat Software. Päivitetty 8.10.2004. Luettu 9.11.2012.

15 Haml 2012. (Verkkodokumentti) <http://www.haml.info> The Haml Team. Päivitetty 9.10.2012. Luettu 11.11.2012.