TAMPEREEN TEKNILLINEN YLIOPISTO

JUHA SIMOLA

HAJAUTETTU TYÖAJANSEURANTA DCI-ARKKITEHTUURIA KÄYTTÄEN

Diplomityö

Tarkastaja: Professori Tommi Mikkonen
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan
tiedekuntaneuvoston kokouksessa
3.10.2012

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
SIMOLA, JUHA: Hajautettu työajanseuranta DCI-arkkitehtuuria käyttäen
Diplomityö, 45 sivua
Toukokuu 2013
Pääaine: Sulautetut järjestelmät
Tarkastaja: Professori Tommi Mikkonen
Avainsanat: DCI, työajanseuranta, Qt, C++11, MVC, hajautettu järjestelmä, ohjelmistoarkkitehtuuri

Ohjelmistotuotannossa projektinhallinta, ajankäytön seuranta ja aikataulun pitävyys ovat tärkeitä laatutekijöitä. Kiinteähintaisista projekteista on siirrytty enenevässä määrin ketterään ohjelmistokehitykseen ja suoraan tuntiperusteiseen laskutukseen. Kehitys on lisännyt entisestään tarkan työajanseurannan merkitystä.

Ohjelmistojen koko kasvaa jatkuvasti, ja arkkitehtuurin hallinta monimutkaistuu. Yleisesti käytetyissä olioarkkitehtuureissa monivaiheisten käyttötapausten toiminta koostuu lukuisten olioiden yhteistoiminnasta. Kokonaisjärjestelmän toiminnallisuus hajoaa suuressa järjestelmässä moneen eri luokkaan, ja kokonaisuuden hahmottaminen vaikeutuu. Ratkaisuna ongelmaan on esitetty uutta DCI-arkkitehtuurimallia, jossa ohjelman rakenne suunnitellaan käyttäjälähtöisesti. DCI-arkkitehtuurissa käyttötapaukset ovat kokonaisjärjestelmän rakennusosia.

Tässä diplomityössä toteutetaan Qubit-työajanseurantasovellus DCI-arkkitehtuuria ja alustariippumatonta Qt-ohjelmistokehystä käyttäen. Toteutuksessa käytetään myös C++11-standardin tuomia uusia ominaisuuksia ja arvioidaan niiden hyödyllisyyttä. Työpöytäsovellus mittaa työaikaa automaattisesti taustalla ja integroituu yrityksen pilviympäristössä toimivaan projektinhallintajärjestelmään. Järjestelmä mahdollistaa työajanseurannan ja laskutuksen automatisoinnin.

Työssä selvitetään, miten akateemisessa ympäristössä tutkittu DCI-arkkitehtuuri soveltuu käytettäväksi oikeassa työelämän projektissa. Työssä tutkitaan arkkitehtuurin hyötyjä perinteiseen olioarkkitehtuuriin nähden ja arvioidaan, miten DCI soveltuu käytettäväksi ominaisuuksiltaan erilaisissa ja eri kokoisissa ohjelmistoissa. Qubit-sovelluksen toteutus oli kokonaisuutena onnistunut, ja DCI osoittautui toimivaksi arkkitehtuurimalliksi myös käytännössä. Arkkitehtuurin ansiosta järjestelmätason toiminta ja eri käyttötapaukset ovat selkeästi dokumentoituna koodissa. DCI helpottaa toimintojen automaattista testausta, koodin ylläpitoa ja ketterää kehitystä.

DCI soveltuu erityisesti käytettäväksi paljon eri käyttötapauksia sisältävissä järjestelmissä. Pieniin, yksinkertaisiin ja vähän käyttötapauksia sisältäviin sovelluksiin se ei kuitenkaan sovellu. Tällöin perinteisen oliosuunnittelun ja DCI-arkkitehtuurin välimuoto on järkevin vaihtoehto. Mitä suurempi ja käyttötapauksiltaan monimutkaisempi sovellus on, sitä enemmän hyötyä saavutetaan. Jatkossa arkkitehtuuria tulisikin testata ja soveltaa tätä projektia suuremmassa projektissa.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
SIMOLA, JUHA: Distributed work time tracking using DCI architecture
Master of Science Thesis, 45 pages
May 2013
Major: Embedded Systems
Examiner: Professor Tommi Mikkonen
Keywords: DCI, work time tracking, Qt, C++11, MVC, distributed system, software architecture

Project management, time tracking and firm schedule are all important quality factors in software production. Today, projects more often use agile work flow and hour-based invoicing and as a result accurate time tracking is increasingly important.

The size of software projects are increasing, this making architecture design more demanding and until recently object-orientation has been the mainstream architecture utilised. The overall system behaviour is distributed in several objects and the system functionality is created by complex co-operation of these objects.

It has been stated that DCI architecture is a solution for the complexity as it clarifies the system-wide functionality. DCI uses use cases as system functionality building blocks. It also uses the end-user's mental models for structure design.

In this thesis, Qubit work time tracking software is implemented using DCI architecture on Qt cross-platform application framework. The implementation also uses a few new features of the new C++11-standard; the usefulness and usability of these are also discussed. The desktop application measures work time automatically at background. The application also integrates to a project management system, which runs on cloud. The management system enables company-wide work time tracking and automatised customer invoicing.

This thesis concentrates on DCI architecture, which has recently been under academic research and studies how well the architecture can be utilised in a real work project. The thesis studies the differences, advantages and disadvantages between DCI and the traditional object-oriented design. Moreover, it evaluates how DCI applies in projects with different sizes and features. The thesis project was successful in general with DCI proving to be a good software architecture for practicality. Due to DCI, the system-wide functionality and use cases are clearly documented in source code. The architecture also simplifies automatic testing, code maintenance and agile development.

DCI applies particularly well to applications that include many use cases, however, it does not apply for small and simple applications that include only few real use cases. In such instances, a composite of the object-oriented design and DCI architecture is the best alternative. The larger the system and the more use cases it has, the more gain the DCI architecture obtains. Thus, future studies should be made using DCI on a larger scale.

ALKUSANAT

Tämä diplomityö tehtiin Tampereen teknillisen yliopiston Tietotekniikan laitokselle talven 2012–2013 aikana. Työssä toteutettiin Vincit Oy:lle automaattinen työajanseurantasovellus uutta DCI-arkkitehtuuria käyttäen. Työ on osa Vincitin sisäistä kehitysprojektia, jossa toteutettiin yrityksen käyttöön uusi projektien ja liiketoiminnan hallintajärjestelmä. Yrityksen puolesta diplomityön ohjaajana toimi DI Pekka Virtanen.

Vincit on tamperelainen, vuonna 2007 perustettu ohjelmistoyritys, joka tuottaa sulautettuja järjestelmiä, mobiilisovelluksia ja web-sovelluksia. Vincit on tulorahoitteinen, kannattava kasvuyritys, ja se työllistää tällä hetkellä noin 70 työntekijää.

Tietotekniikan laitokselta diplomityön tarkastajana toimi professori Tommi Mikkonen. Haluan kiittää häntä asiantuntevista kommenteista ja määrätietoisesta motivoivasta ohjauksesta työn aikana. Vincitiltä haluan kiittää työn ohjaajaa Pekka Virtasta rakentavista kommenteista sekä työtoveriani Janne Rönkköä yhteistyöstä projektin aikana.

Lisäksi haluan kiittää Mari Kuuselaa työn oikoluvusta ja tuesta kirjoitusprosessin aikana. Kiitokset myös ystävälleni Tanja Viirulle kommenteista diplomityöhön liittyen.

The answer to the ultimate question of life, the universe and everything is 42.
(Adams, Douglas. The Hitchhiker's Guide to the Galaxy. 1979.)

Tampereella, 26. maaliskuuta 2013.

Juha Simola

SISÄLLYS

1. JOHDANTO

Onnistunut projektinhallinta, ajankäytön seuranta ja aikataulujen pitävyys ovat tärkeitä menestystekijöitä ohjelmistoalan yrityksissä. Aiemmin ohjelmistoprojektit olivat pääosin kiinteähintaisia, mutta ketterien menetelmien myötä projekteissa on siirrytty yhä enemmän tuntiperusteiseen laskutukseen. Tuntiperusteinen laskutus lisää entisestään tuntikirjanpidon merkitystä.

Joissakin yrityksissä projektinhallintaan, työntekijöiden tuntiseurantaan ja asiakaslaskutukseen käytetään edelleen Excel-taulukkolaskentaohjelmaa. Projekteissa saatetaan myös käyttää useita toisistaan irrallisia järjestelmiä, ja hajanaisen järjestelmäkokonaisuuden hallinta ja ylläpito on monimutkaista. Projektien ja liiketoiminnan hallinnalle, tuntiseurannalle ja laskutuksen automatisoinnille onkin selkeä tarve. Monissa yrityksissä on kehitetty omia tuntikirjaus- ja laskutusjärjestelmiä tai otettu käyttöön valmiita projektinhallintasovelluksia.

Tässä diplomityössä kuvataan Vincit Oy:lle nykyisen tuntikirjausjärjestelmän pohjalta toteutettava uusi tuntikirjausjärjestelmä. Työssä kuvataan pilviympäristössä toimiva BIT (*Business Intelligence Toolkit*) -projektinhallintajärjestelmä ja toteutetaan sen kanssa kommunikoiva *Qubit*-työpöytäsovellus työajan automaattiseen seurantaan. Qubit-sovellus toteutetaan alustariippumatonta Qt-sovelluskehystä käyttäen, sillä työntekijöillä on käytössä eri käyttöjärjestelmiä. Qt:n ansiosta kaikki työntekijät voivat käyttää samaa tuntikirjaussovellusta.

Qubit-sovelluksen toteutuksessa käytetään uutta DCI (*Data-Context-Interaction*) -arkkitehtuuria. Arkkitehtuurin lähestymistapa ohjelmiston kehittämiseen on käyttäjälähtöinen, sillä käyttötapaukset ovat siinä keskeisessä osassa järjestelmän rakennusosina. Arkkitehtuuri tukee myös ketterää ohjelmistokehitystä ja *Lean*-ajattelumallia, jossa prosessista pyritään karsimaan pois kaikki turha työ ja loppukäyttäjälle tarpeettomat ominaisuudet. DCI erottaa järjestelmän toiminnan erilleen järjestelmän tietomallista ja rakenteesta. Koko järjestelmän toiminta on sen vuoksi selkeämmin dokumentoitu sovelluksen koodissa. Myös uusien ominaisuuksien lisäys on suoraviivaisempaa.

Työn tavoitteena on hyödyntää lähinnä akateemisessa ympäristössä tutkittua DCI-arkkitehtuuria ja selvittää, miten se soveltuu käytettäväksi käytännön työelämän projektissa. Työssä käytetään myös uuden C++11-standardin ominaisuuksia ja arvioidaan niiden hyödyllisyyttä.

Luvussa 2 esitetään Qubit-sovelluksen toteutuksessa käytetyt teknologiat. Tämän jälkeen luvussa 3 kuvataan pilviteknologiat, joita Qubit-sovellus tarvitsee kommunikoidessaan BIT-projektinhallintajärjestelmän kanssa. Luvussa 4 esitellään yleisesti nykyinen tuntikirjaus- ja projektinhallintajärjestelmä ja käsitellään syitä uuden järjestelmän kehittämiselle. Luvussa 5 kuvataan uuden tuntikirjaus- ja projektinhallintajärjestelmän arkkitehtuuri ja toteutusyksityiskohdat. Erityisesti huomioitavaa on esimerkki DCI-arkkitehtuurilla toteutetun käyttötapauksen toiminnasta. Luvussa 6 arvioidaan sovelluksen ja projektin onnistumista sekä työssä käytettyjen teknologioiden soveltuvuutta. Lopuksi esitetään projektin aikana syntyneet jatkokehitysajatukset.

2. TOTEUTUSTEKNOLOGIAT

Tässä luvussa kuvataan diplomityössä toteutettavassa Qubit-sovelluksessa käytetyt teknologiat. Aluksi esitetään yleisesti, mikä on DCI-arkkitehtuuri ja käsitellään sen eroja olio-ohjelmointiin ja yleisesti käyttöliittymäohjelmoinnissa käytettyyn MVC (*Model-View-Controller*) -arkkitehtuuriin nähden. Seuraavaksi kuvataan toteutuksessa käytetty alustariippumaton Qt-ohjelmistokehys. Lopuksi esitellään C++-ohjelmointikieli ja erityisesti äskettäin julkaistun C++11-standardin uudet ominaisuudet, joita hyödynnetään sovelluksen toteutuksessa.

2.1 DCI

DCI (*Data, Context, Interaction*) -arkkitehtuurin kehittäjä on norjalainen emeritusprofessori Trygve Reenskaug, joka oli merkittävä henkilö myös MVC-mallin kehityksessä. Reenskaug osallistui myös *UML* (*Unified Modelling Language*) -kuvauskielen kehittämiseen ja tutki koodin tuottamista korkeamman tason kuvauksesta. [1]

Reenskaug halusi kehittää ohjelmointitavan, jolla ohjelmakoodi olisi luettavampaa, helpommin katselmoitavaa ja vastaisi rakenteeltaan ja toiminnaltaan mahdollisimman täydellisesti loppukäyttäjän ajatusmalleja. Olio-ohjelmoinnin tuottama koodi ei Reenskaugin mielestä dokumentoi riittävästi järjestelmän toimintaa vaan ainoastaan sen sisäistä rakennetta. Keskeisenä ajatuksena on, että kokonaisjärjestelmän toiminnan tulisi olla selkeästi nähtävissä ohjelmakoodissa. [1; 2, s. 236–240]

DCI on jatketta vuonna 1995 esitellylle roolipohjaiselle ohjelmoinnille. Ensimmäiset ajatukset DCI-arkkitehtuurista syntyivät vuosien 2005 ja 2006 aikana. Reenskaug julkaisi toimivan DCI-prototyypin Squeak-ohjelmointiympäristölle vuonna 2008. Samoihin aikoihin James Coplien liittyi mukaan DCI-kehitystyöhön ja julkaisi prototyypin C++-ohjelmointikielelle. Vuoteen 2010 mennessä DCI-yhteisö jakoi esimerkkitoteutuksia jo useille eri ohjelmointikielille. [1; 2, s. 301]

Perinteinen olio-ohjelmointi kapseloi olion sisäisen datan ja siihen liittyvän toiminnan rajapintafunktioiden taakse. Järjestelmätason toiminta on oliosuunnittelussa ripoteltuna useaan eri luokkaan, jotka yhdessä muodostavat toimivan kokonaisuuden. [2, s. 236, 241]

Olio-ohjelmoinnissa kokonaisjärjestelmän rakennuspalikoita ovat yksittäiset luokat. Järjestelmän koon kasvaessa luokkien lukumäärä kasvaa hyvin suureksi vaikeuttaen kokonaisuuden hahmottamista. Monimutkaisissa järjestelmissä on hanka-

laa hahmottaa, miten luokkien yhteistoiminta aikaansaa järjestelmän toiminnan. Ohjelmistokehittäjän saattaa olla vaikeaa vakuuttua siitä, että tietty monivaiheinen käyttötapaus toimii oikein. [2, s. 241]

Korkeamman tason ohjelmointikielet on kehitetty kuvaamaan tietokoneessa suoritettavaa koodia, prosesseja ja resursseja korkeammalla abstraktiotasolla. Olio-ohjelmointi nostaa abstraktiotasoa edelleen kapseloimalla datan ja siihen liittyvät toiminnot. Nykyisin käytettyjen ohjelmointikielten ja olio-ohjelmoinnin kehitys on siis edennyt teknologialähtöisesti. [3]

DCI:n lähestymismalli ohjelmiston rakenteeseen on edellisen sijaan käyttäjälähtöinen: *Building software as if people mattered*. Coplienin mukaan DCI on olio-ohjelmoinnista seuraava askel kohti korkeampaa suunnittelun abstraktiotasoa. Arkkitehtuurissa käyttötapaukset ovat kokonaisjärjestelmän rakennuspalikoita ja järjestelmän toiminta on erotettu selkeästi järjestelmän rakenteesta. [2; 3]

DCI:n tavoitteena on kuvata ohjelmakoodissa yksittäisten luokkien rakenteen sijaan koko järjestelmätason toiminta. Tavoitteena on siis parantaa koodin luettavuutta perinteiseen olio-arkkitehtuuriin nähden. Arkkitehtuuri pyrkii myös mallintamaan luonnollista maailmaa ja ihmisten ajattelumalleja koodiksi mahdollisimman suoraviivaisesti. Ihmiset ja ihmisten vuorovaikutus käyttöliittymän kanssa on ajatuksena tärkeämpää kuin teknologialähtöinen suunnittelu, ohjelmaprosessit ja ohjelmointikielten työkalut. [2, s. 173, 245]

Koska DCI pyrkii rakentamaan koodin niin, että se kuvaa sovellusalueen käsitteitä ja loppukäyttäjän näkemystä järjestelmän toiminnasta, on sen etuna myös asiakaskommunikaation helpottuminen. Ohjelmistosuunnittelijat ja loppukäyttäjät näkevät järjestelmän rakenteen samanlaisena. Arkkitehtuuri helpottaa myös järjestelmän toimintojen testausta, koska DCI yhdistää koko käyttötapauksen toiminnan yhteen koodiluokkaan, ja toiminnan oikeellisuudesta on helpompi vakuuttua. [2, s. 294–295]

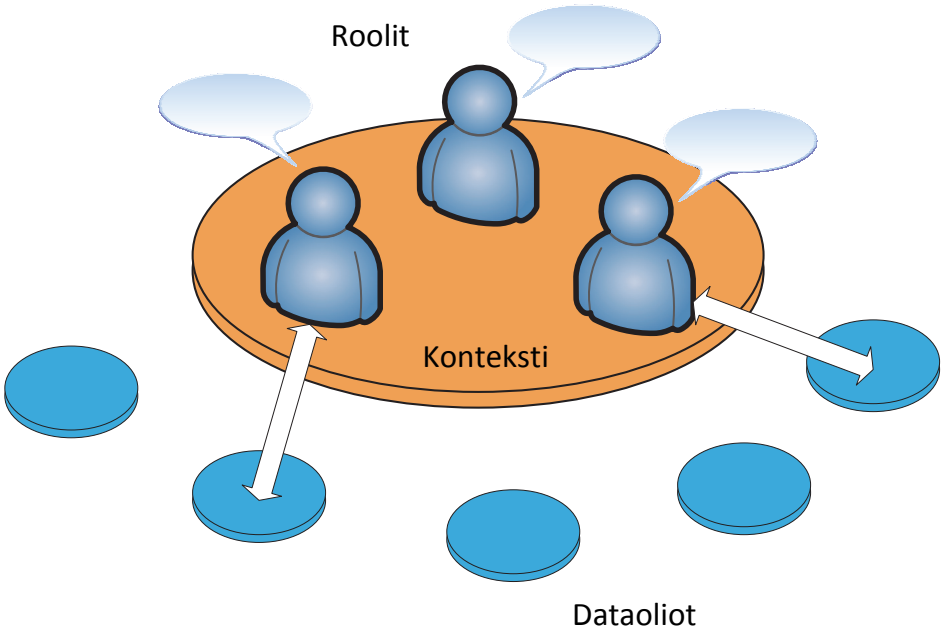
DCI tukee myös *Lean*-tuotantomallia. *Lean*-malli on alun perin Toyota-autotehtaan 1900-luvun alussa kehittämä tuotantomalli, jossa tuotetta tehdään vasta, kun sille on tilausta. Mallissa pyritään minimoimaan tuottamattoman työn ja syntyvän jätteen määrä. Kaiken tehtävän työn on tuotettava arvoa asiakkaalle. [2, s. 38–41]

Lean-malli ei historiastaan huolimatta liity itse autotuotantoon, vaan se on laajempi projektihallinnan ideologia. Malli on hyvin sovellettavissa myös ketterään ohjelmistotuotantoon. Järjestelmään ei saa syntyä *jätekoodia*, josta ei ole hyötyä loppukäyttäjälle. Järjestelmään kannattaa kehittää ominaisuuksia vasta, kun niille on tarvetta. Jokaisen uuden ominaisuuden ja käytetyn työtunnin on tuotettava lisäarvoa käyttäjälle. Korjaustyö ja kerran toteutettujen osien uudelleen tekeminen on myös *jätettä*, josta pitäisi tuotannossa päästä eroon. [2, s. 82–83, 288]

2.1.1 Yleiskuvaus

DCI-arkkitehtuuri pyrkii rakenteellaan erottamaan toisistaan erilleen järjestelmän olemuksen ja tiedon (*What system is*) ja järjestelmän toiminnallisuuden (*What system does*). DCI olettaa, että järjestelmän tietomalli pysyy suhteellisen muuttumattomana koko sen elinkaaren ajan. Järjestelmän olemus muuttuu hyvin hitaasti, mutta toiminnallisuus on järjestelmän nopeasti muuttuva ja kehittyvä osa. Ketterän kehityksen ja asiakasvaatimusten myötä uusia ominaisuuksia lisätään jatkuvasti. Vanhoja käyttämättömiä ominaisuuksia saatetaan myös karsia pois. [2, s. 27–33, 89, 245, 288]

DCI:n toimintaidea on esitetty kuvassa 2.1. Kun käyttötapaus alkaa, luodaan konteksti-olio (*Context*), joka on vastuussa käyttötapauksen suorittamisesta. Konteksti valitsee käyttötapaukseen osallistuvat dataoliot ja antaa niille roolit, joissa ne ovat käyttötapauksen aikana. Kun konteksti on luotu, kontekstin käyttötapaus aloitetaan, ja ensimmäinen käyttötapaukseen osallistuva rooli aloittaa osansa. Roolit voivat kommunikoida keskenään kontekstin välityksellä. Käyttötapaus etenee ja eri roolit ovat vuorollaan suorituksessa, kunnes käyttötapauksen toiminta on saatu valmiiksi ja käyttötapaus päättyy. [2]



Kuva 2.1. *DCI-arkkitehtuurin toiminta.*

Arkkitehtuurin toimintaa voidaan idean ymmärtämiseksi verrata teatteriin, jossa dataoliot ovat näyttelijöitä. Teatterianalogiassa käyttötapausta voidaan ajatella näyttämökohtauksena, jossa näyttelijät ovat tietyissä rooleissa. Kohtaus alkaa ja ensimmäinen näyttelijä näyttelee osansa, minkä jälkeen toinen rooli jatkaa ja lopulta kohtaus päättyy.

Sama näyttelijä voi olla useassa eri roolissa riippuen kohtauksesta. Myös yhden kohtauksen sisällä hän voi periaatteessa näytellä kahta tai useampaa roolia. Yksi näyttelijä ei kuitenkaan useinkaan voi näytellä kaikkia rooleja vaan sopii ominaisuuksiltaan vain tiettyihin rooleihin. Samaan tapaan dataolio voi olla eri käyttötapauksissa eri rooleissa ja käyttötapauksen aikana aluksi yhdessä ja myöhemmin toisessa roolissa. Dataoliot myös sopivat vain tiettyihin rooleihin.

Yksi rooleista on usein järjestelmän käyttäjä, eikä siihen liity näyttelevää dataoliota. Kuvassa 2.1 käyttäjän rooli on keskimmäisenä. Käyttäjän roolin suorituksen yhteydessä käyttöliittymässä näytetään esimerkiksi dialogeja, joilla saadaan käyttäjän toiminta mukaan käyttötapauksen suoritukseen. Käyttötapaus jatkuu eteenpäin sen mukaan, mitä käyttäjä roolinsa vuorolla tekee.

Vaikka näyttelijällä on useita rooleja, näyttelijä itse on silti aina sama henkilö. Samaten DCI-arkkitehtuurissa dataoliot sisältävät olion muuttumattoman identiteetin ja datan. Olion toiminta puolestaan liittyy rooliin, jota olio kulloinkin näyttelee.

Myös reaalimaailmassa sama henkilö voi toimia eri rooleissa. Työpaikalla henkilö voi työskennellä ohjelmistosuunnittelijana mutta olla samalla projektipäällikkönä ja luottamusmiehenä. Kotona sama henkilö voi edelleen olla isän tai äidin roolissa. [3]

Kontekstien ja roolien toimintaa havainnollistaa esimerkki, jossa suoritetaan rahansiirto tililtä toiselle. Tilit ovat dataolioita, joiden jäsenfunktioilla voidaan lisätä ja vähentää tilin saldoa. Rahansiirron suorittaa konteksti *TransferMoneyContext*. Kontekstissa ovat läsnä lähdetiliä ja kohdetiliä näyttelevät roolit *SourceAccount* ja *DestinationAccount*. Sama tili voi käyttötapauksesta riippuen olla joko lähde- tai kohdetilin roolissa. [2, s. 265–277]

Käyttötapauksessa käyttäjä valitsee rahasumman, lähdetilin ja kohdetilin ja painaa *Maksa*-painiketta, jonka painalluksesta käyttötapaus alkaa. Käyttötapauksen alussa luodaan *TransferMoneyContext*-olio, joka saa parametreinaan valitun rahasumman sekä lähdetiliä ja kohdetiliä näyttelevät oliot:

```
TransferMoneyContext(Currency amount, SourceAccount src, DestinationAccount destination);
```

Kontekstilla on lisäksi suoritusfunktio, joka tekee rahansiirron ja käsittelee virhetilanteet. Alla oleva esimerkki on yksinkertaistettu:

```
void execute()
{
  try
  {
    sourceAccount.decreaseBalance(amount);
    destinationAccount.increaseBalance(amount);
  }
  catch (...) { ... }
}
```

Seuraavissa alakohdissa käsitellään tarkemmin DCI-arkkitehtuurin osat. Tietomallista (*Data*), kontekstista (*Context*) ja toiminnasta (*Interaction*) kuvataan osien toteutus sekä niiden keskinäiset suhteet ja vastuualueet.

2.1.2 Tietomalli

DCI-arkkitehtuurin tietomalli (*Data*) sisältää järjestelmän hitaasti muuttuvan *What system is* -osan. Dataoliot toteutetaan perinteisen olio-ohjelmoinnin mukaisesti, ja tietomalli on hyvin samanlainen kuin MVC-arkkitehtuurissa. Erona MVC-arkkitehtuuriin on se, että DCI-suunnittelussa dataolioiden rajapinnat ovat yksinkertaisia ja minimaalisia, eivätkä ne sisällä toiminnallisuutta. [3]

Dataoliot kuvaavat mahdollisimman suoraviivaisesti loppukäyttäjän ajatusmallia järjestelmän sisältämästä tiedosta. Niiden tehtävänä on lähinnä abstrahoida käytetty tiedon tallennustapa. Dataoliolla on *get*- ja *set*-funktiot mutta ei datan käsittelyyn liittyvää toiminnallisuutta, joka liittyisi johonkin tiettyyn ohjelman toimintoon. [2, s. 237]

Tavoitteena on säilyttää arkkitehtuurin tietomallikerros mahdollisimman staattisena. Näin tietomalli säilyy puhtaasti tietomallina eikä siihen sekoitu useita loppukäyttäjän roolimalleja. Uuden toiminnon lisääminen tai tarpeettoman toiminnon poistaminen ei useinkaan aiheuta muutoksia tietomalliin. [2, s. 245, 288–289]

Yksinkertainen tietomalli tukee myös Lean-ajattelua, jossa koodia tuotetaan juuri, kun sille on tilausta. Tietomallin rajapinnat määritellään mahdollisimman täydellisesti jo suunnitteluvaiheessa, mutta malli voi projektin alussa sisältää toteutukset vain niille funktioille, joita järjestelmän olemassa olevat käyttötapaukset tarvitsevat. Funktioille voidaan kirjoittaa toteutusta inkrementaalisesti vasta siinä vaiheessa, kun niitä tarvitaan. [2, s. 288–289]

2.1.3 Konteksti

Konteksti (*Context*) vastaa DCI-arkkitehtuurissa yhtä käyttötapausta. Konteksti sisältää roolit ja algoritmit käyttötapauksen suorittamiseksi. Roolit voivat kommunikoida toisilleen kontekstin välityksellä. Konteksti ja sen sisältämät roolit määrittelevät järjestelmän *What system does* -toiminnallisuuden. [2, s. 243–246]

Kun käyttöliittymätapahtuma tapahtuu ja käyttötapaus alkaa, luodaan käyttötapauksen kontekstiolio. Luonnin yhteydessä sidotaan, mitkä dataoliot ovat käyttötapauksessa missäkin roolissa. Kun konteksti on luotu, kutsutaan sen suoritusfunktiota. Kontekstiin liittyvillä rooleilla on funktioita, jotka ovat pieni osa kokonaista käyttötapausta. Koko käyttötapaus on suoritettu, kun kaikki siihen osallistuvat roolit ovat suorittaneet oman tehtävänsä. [2, s. 243–246]

Järjestelmän toimintalogiikka on koodattu kontekstiluokkaan ja siihen liittyviin rooleihin. Kontekstien ja roolien koodi on geneeristä, ja esimerkiksi C++-kielellä ne on toteutettu luokka- ja funktiomalleja (*template*) käyttäen. [2, s. 257–259]

Suuremmassa järjestelmässä jotkin käyttötapaukset voivat olla osa suurempaa käyttötapausta. Voikin olla niin, että pienemmän käyttötapauksen suorittava kon-

teksti voi itse näytellä tiettyä roolia suuremmassa kontekstissa. Roolin näyttelijän ei siis välttämättä tarvitse olla dataolio. [2, s. 290–294]

Toisen roolin metodia kutsuva rooli ei tiedä, kuka kyseistä roolia näyttelee. Roolin identiteetillä ei kuitenkaan ole merkitystä, kunhan olio tai toinen konteksti pystyy näyttelemään roolin. [3]

2.1.4 Toiminta

DCI-arkkitehtuurin toimintaosuus (*Interaction*) koostuu rooleista, joita dataoliot näyttelevät. Roolit suorittavat järjestelmässä käyttötapausten pieniä osakokonaisuuksia, joista koko järjestelmän toiminta koostuu. Roolit yhdistävät toisiinsa dataoliot ja omat metodinsa. C++-kielellä roolit on kontekstien tapaan toteutettu luokka- ja funktiomalleja käyttäen. [2, s. 240–242, 257–259]

Roolit ovat tilattomia, sillä rooliluokat sisältävät vain roolin toimintaan liittyvän geneerisen koodin. Dataolioilla sen sijaan on tallennettuna tietosisältö ja olion tila – mutta ei toimintaa. [3]

Roolit voivat kommunikoida keskenään kontekstin välityksellä. Jokaisella roolilla on myös erityinen jäsenmuuttuja *self*, jonka avulla roolin koodi pystyy viittaamaan dataolioon, joka kyseistä roolia sillä hetkellä näyttelee. [2, s. 257]

Jokaisella roolilla on käyttötapauksessa vain yksi dataolio, joka roolia näyttelee. Näyttelijä voi kuitenkin vaihtua toiseen käyttötapauksen edetessä. Sen sijaan yksi dataolio voi samanaikaisesti toimia useassa eri roolissa. [4]

2.2 Qt

Qt on vuonna 1995 julkaistu norjalaisen Trolltechin kehittämä alustariippumaton ohjelmistokehys. Trolltechin tavoitteena oli kehittää helppokäyttöinen kirjasto C++-käyttöliittymäohjelmointiin, jolla samaa koodia voidaan käyttää käyttöjärjestelmästä riippumatta. Myöhemmin Qt on laajentunut käyttöliittymäkehyksestä soveltumaan myös tekstipohjaisten sovellusten, palvelinsovellusten ja mobiilisovellusten kehitysympäristöksi. Nokia osti Qt:n Trolltechiltä vuonna 2008, ja sitä käytettiin laajasti matkapuhelinten ohjelmistokehitysympäristönä. Diplomityön kirjoittamisen aikana Qt on myyty Digialle, joka jatkaa sen kehittämistä. [5; 6]

Qt abstrahoi eri käyttöjärjestelmäalustojen eroavaisuudet Qt-rajapintojen taakse ja käyttää sisäisesti alustojen omia rajapintoja. Käännetty Qt-sovellus näyttää siis samalta kuin kyseiselle alustalle tehty natiivisovellus. Qt:n tukemia alustoja ovat Windows, Linux/X11, Mac OS X ja Symbian. Diplomityöprojektissa on käytössä Qt-versio 4.8. [7]

Qt on jaettu käyttötarkoituksen mukaan moduuleihin, jotka voidaan ottaa käyttöön erikseen. Moduulit ja niiden sisältö on kuvattu taulukossa 2.1. Moduuleista

tärkeimpiä ovat *QtCore* ja *QtGui* -moduulit. *QtCore* sisältää Qt:n perustoiminnallisuuden ja *QtGui* graafisen käyttöliittymän. Työssä käytetään lisäksi *QtNetwork*-moduulia verkkokommunikointiin ja *QtSql*-moduulia paikallisen tietokannan käsittelyyn. [7]

Taulukko 2.1. *Qt:n moduulijako.* [8]

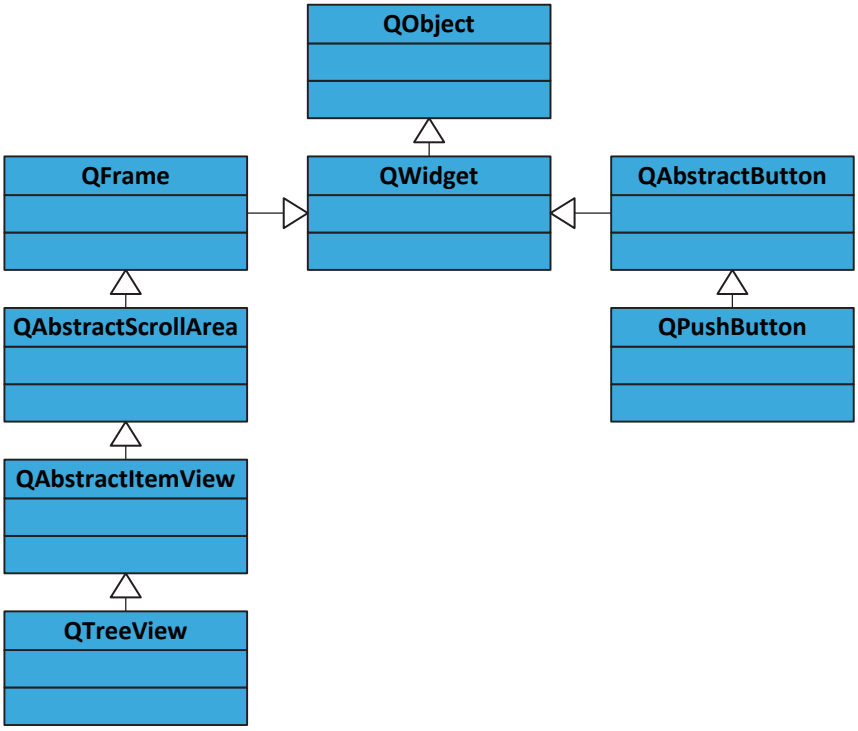
QtCore	Qt:n ei-graafinen ydintoiminnallisuus.
QtGui	Graafiset käyttöliittymäkomponentit.
QtMultimedia	Matalan tason multimediatoiminnallisuus.
QtNetwork	Verkkokommunikointi.
QtOpenGL	OpenGL-tuki.
QtOpenVG	OpenVG-tuki.
QtScript	Qt-skriptien suorittaminen.
QtScriptTools	Qt-skriptien lisäkomponentit.
QtSql	SQL-tietokantakommunikointi.
QtSvg	SVG-tiedostojen näyttäminen.
QtWebKit	Web-sisällön näyttäminen ja muokkaaminen.
QtXml	XML-tiedostojen käsittely.
QtXmlPatterns	XQuery & XPath toiminnallisuus XML-moduulille.
QtDeclarative	Deklaratiivinen käyttöliittymien rakentaminen.
Phonon	Multimediatoiminnallisuus.
Qt3Support	Qt3-yhteensopivuus.

Qt on toteutettu oliosuunnittelun periaatteiden mukaisesti, ja se käyttää vahvaa periytymishierarkiaa. Periytymishierarkian ansiosta toiminnallisuuden lisääminen onnistuu vähäisellä työmäärällä, koska luokille yhteinen jo olemassa oleva toteutus voidaan periyttää uuteen luokkaan. Qt:n luokkahierarkian suunnittelussa on pyritty esittämään luokkien väliset periytymissuhteet mahdollisimman luonnollisesti. Luokkien rajapinnat on pyritty suunnittelemaan intuitiivisiksi, ja niissä on käytetty johdonmukaisia nimeämiskäytäntöjä. [7]

Luokkahierarkiassa käytetään runsaasti kantaluokkia, joihin on koottu mahdollisimman paljon yhteen luokkien yhteistä toiminnallisuutta. Lähes kaikki Qt:n luokat periytyvät *QObject*-kantaluokasta. Käyttöliittymäkomponentit periytyvät edelleen *QWidget*-luokasta, joka sisältää käyttöliittymäkomponenteille yhteisen toiminnallisuuden. [9]

Kuvassa 2.2 on esimerkki periytymishierarkiasta. Kuvassa on esitetty käyttöliittymäpainikkeen toteuttavan *QPushButton*-luokan ja puurakenteisen listanäkymän toteuttavan *QTreeView*-luokan periytyminen. Esimerkiksi *QPushButton*-luokka periytyy kaikille painikkeille yhteisestä *QAbstractButton*-kantaluokasta. *QAbstractSc-*

rollArea puolestaan toteuttaa kaikille vieritettäville näkymille yhteisen toiminnallisuuden. [9]

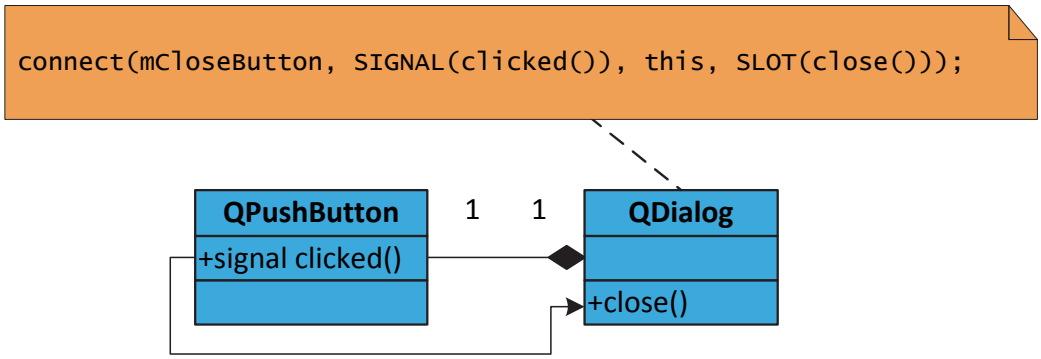


Kuva 2.2. *Esimerkki Qt:n luokkahierarkiasta.*

Qt:n käyttöliittymätapahtumat perustuvat *signals and slots* -tekniikkaan, jossa käyttöliittymän tapahtuma aiheuttaa signaalin lähettämisen. Tapahtumasta kiinnostuneet luokat voivat kytkeä signaalin porttiin (*slot*), joka on tavallinen luokan jäsenfunktio. Signaalin lähettäminen aiheuttaa kaikkien siihen liittyvien portti-funktioiden kutsumisen vuoron perään. Rinnakkaisuuteen ei tarvitse varautua, sillä Qt puskuroi sisäisesti signaalit siten, että vain yksi signaali on käsittelyssä kerrallaan ja vain yksi portti-funktio on kerrallaan suorituksessa. [10]

Signaalin ja portin ollessa samassa säikeessä signaalin lähetys tehdään normaalina funktiokutsuna. Signaalin lähetys kahden eri säikeen välillä toteutetaan Qt:ssa sisäisesti käyttämällä viestinvälitystä ja kutsumalla portti-funktiota Qt:n tapahtumasilmukasta (*Event Loop*). Signaalien yhdistäminen portteihin tapahtuu ajon aikana. Tästä seuraa huonona puolena se, ettei signaalikytkennän epäonnistuminen tai signaalin puuttuminen aiheuta käännösvirhettä vaan ainoastaan ajonaikaisen virheen. [10; 11]

Signaalien käyttö on varsin helppoa ja intuitiivista. Koodissa signaalit yhdistetään portteihin käyttämällä *QObject*-luokan *connect()*-funktiota sekä Qt:n makroja *SIGNAL()* ja *SLOT()*. Havainnollistava kuva ja signaalikytkennän koodiesimerkki on kuvassa 2.3.



Kuva 2.3. *Esimerkki Qt:n signaalikytkennästä.*

Koodiesimerkissä *QPushButton*-painikkeen (*mCloseButton*) painamisesta lähetettävä signaali *clicked()* yhdistetään *QDialog*-ikkunan (*this*) jäsenfunktioon *close()*. Painikkeen painaminen aiheuttaa ikkunan sulkemisen.

Qt:n laajennokset C++-ohjelmointikieleen eivät käänny C++-kääntäjällä suoraan, vaan koodi esikäännetään Qt:n *Meta-object compiler* -kääntäjää käyttäen. *Meta-object compiler* lukee luokkien otsikkotiedostot ja tuottaa Qt:n ominaisuuksien C++-koodin. Kääntäjää tarvitaan muun muassa *signals and slots* -mekanismin toteuttamiseen ja olioiden ajonaikaiseen tyyppitietoon. *Meta-object compiler* -kääntäjän tuottama koodi käännetään ja linkitetään normaalilla C++-kääntäjällä valmiiksi sovellukseksi. Käännös voidaan automatisoida *qmake*- tai *cmake* -käännöstyökaluja käyttäen. [12]

Qt:ta ja Qt-sovelluksia voidaan helposti laajentaa liitännäisillä (*plugin*). Liitännäiset helpottavat järjestelmän muunneltavuutta, sillä ne eristävät lisäosan toteutuksen erillisen rajapinnan taakse. Liitännäisiä voidaan lisätä, poistaa ja vaihtaa muuntaen näin sovelluksen toimintaa ja ominaisuuksia. Liitännäisarkkitehtuureissa sovelluksen toiminta ja ominaisuudet määräytyvät sovelluksen saatavilla olevista liitännäisistä. Liitännäisiä voidaan ladata ja vaihtaa dynaamisesti jopa ajon aikana. [13, s. 56, 199]

Esimerkiksi sovelluksen verkkokommunikointi tietyn palvelimen kanssa tai sovelluksen tietokantayhteys voidaan toteuttaa liitännäisillä. Tällöin palvelimen tai tietokannan muuttuessa voidaan sovelluksen toiminta pitää samana ja vaihtaa ainoastaan käytettävää liitännäistä.

Qt:ssa liitännäiset toteutetaan periyttämällä liitännäisluokka *QObject*-kantaluokasta ja julkaisemalla (*export*) liitännäisen toteuttama rajapinta. Liitännäinen käännetään jaetuksi objektitiedostoksi. [5, s. 491–508]

2.3 C++11

Bjarne Stroustrup kehitti C++-ohjelmointikielen, josta julkaistiin ensimmäinen kaupallinen versio vuonna 1985. Ensimmäinen virallinen ISO-standardi C++ ISO/IEC 14882:1998 julkaistiin vuonna 1998. Standardiin tehtiin parannuksia ja korjauksia, joiden tuloksena syntyi C++03-standardi. Samaan aikaan alettiin kehittää listaa kieleen lisättävistä uusista ominaisuuksista. Uuden standardin työnimenä käytettiin C++0X-lyhennettä, jonka lopullinen julkaisu kuitenkin viivästyi vuoteen 2011. Lopulliseksi standardiksi tuli C++11, joka sisältää vanhaan nähden lukuisia uusia ominaisuuksia. [14]

Kieleen otettiin mukaan monia ominaisuuksia avoimen lähdekoodin Boost-kirjastoprojektista [15]. Boost-projektilla olikin suuri vaikutus uuden standardin kehityksessä, sillä monet uudet hyväksi havaitut ominaisuudet otettiin sellaisenaan mukaan uuteen standardiin. Boostin myötä uusia kielen ominaisuuksia ovat säännölliset lausekkeet, uusi satunnaisjakaumakirjasto ja kalenteria ja kellonaikaa käsittelevä kirjasto. Kieleen lisättiin myös tuki monisäikeiseen ohjelmointiin lisäämällä atomiset operaatiot ja säikeet. Myös standardikirjastoon lisättiin uusia säiliötyyppejä. Syntaksiin tehtiin helpottavia muutoksia lisäämällä vaihtelevaparametriset mallit (*variadic templates*), lambda-funktiot, automaattinen muuttujan tyyppi, SI-järjestelmän yksiköt ja *for each* -silmukkarakenne. [14; 16]

Seuraavissa alakohdissa käsitellään tarkemmin diplomityön toteutuksessa hyödynnettyjä uusia C++11-standardin tuomia ominaisuuksia.

2.3.1 Vaihtelevaparametriset mallit

Vaihtelevaparametriset mallit (*variadic templates*) ovat laajennos C++03-standardin malleista (*template*). Mallit mahdollistavat käännösaikaisen metaohjelmoinnin, ja niiden avulla on mahdollista luoda geneerisiä tietorakenteita, funktioita ja luokkia. Funktio- tai luokkamalleissa kuvataan kyseisen luokan toiminta, mutta tietotyypit on jätetty avoimiksi. Samasta mallista voidaan luoda käännösaikana erilaisia instansseja kiinnittämällä avoimiksi jätetyt tyyppiparametrit. [17, s. 285–290]

Esimerkiksi C++-standardikirjaston *vector*-tietorakenne on toteutettu luokkamallina. Vectoriin tallennettujen alkioiden tyyppi määritetään ohjelmakoodissa ja luokkamallin toteutuksesta käännetään tyypille räätälöity toteutus. Seuraavassa esimerkissä luodaan kokonaislukuja sisältävä ja totuusarvoja sisältävä *vector*:

```
std::vector<int> ids;
std::vector<bool> ids;
```

Malleille voidaan määritellä yleisen toteutuksen lisäksi myös erikoistettuja toteutuksia tietyille tyyppiparametreille. Esimerkiksi standardikirjaston *vector<bool>*-tietorakenteelle on muistinkulutuksen optimoimiseksi kirjoitettu oma toteutuksen-

sa, koska *bool*-muuttujan arvo, tosi tai epätosi, on esitettävissä yhdellä bitillä. [17, s. 286]

C++11-standardissa mallit voivat kiinteän parametrien määrän sijasta saada määrittelemättömän määrän tyyppiparametreja. Vaihtelevaparametrinen malli määritellään seuraavasti:

```
template<typename... Arguments>
class VariadicTemplate;
```

Esimerkissä *VariadicTemplate* voi saada vaihtelevan määrän tyyppiparametreja mukaan lukien tilanteen, jossa parametreja ei anneta lainkaan. Syntaksin kolme pistettä kuvaavat vaihtelevaa parametrien määrää. [18]

Uuden standardin mukaisia tyyppimalleja käytettäessä esimerkiksi seuraavat alustukset ovat mahdollisia:

```
VariadicTemplate<double, float> t1;
VariadicTemplate<> t2;
```

Ylempi alustusrivi alustaa tyyppimallin kiinnittäen kaksi tyyppiparametria *double* ja *float*. Alempi rivi ei kiinnitä parametreja lainkaan. Alla olevassa esimerkkikoodissa vaihtelevaparametrisia malleja on käytetty SQL-kyselyn muuttujien sitomiseen:

```
template<typename Arg, typename... QueryParamsT>
std::shared_ptr<QSqlQuery> bind(std::shared_ptr<QSqlQuery> query,
    Arg param,
    QueryParamsT... params) const
{
  query->addBindValue(QVariant(param));
  return bind(query, params...);
}

std::shared_ptr<QSqlQuery> bind(std::shared_ptr<QSqlQuery> query) const
{
  return query;
}
```

Esimerkissä *bind*-funktio saa parametreinaan SQL-kyselyn ja vaihtelevan määrän kyselyyn sidottavia muuttujaparametreja. Funktio sitoo rekursiivisesti kaikki kyselyn parametrit.

2.3.2 Lambda-funktiot

Lambda-funktio on nimeämätön funktio, joka ei ole sidottu tunnisteeseen (*identifier*), ja se voidaan määritellä ohjelmakoodissa toisen funktion sisällä. Lambda-funktio määritellään C++11-kielessä seuraavasti:

```
[capture](parameters)->return-type{body}
```

Määrittelyssä *capture* on lista nykyisellä näkyvyysalueella olevista muuttujista, jotka halutaan näkyvän myös lambda-funktion sisällä, *parameters* on lista lambda-funktion parametreista, *return-type* on lambda-funktion paluuarvon tyyppi, ja *body* sisältää funktion rungon ohjelmakoodin. [19]

Lambda-funktioilla voidaan korvata luokkien jäsenfunktioita ja vanhantyylisiä C-kielen makroja. Lambda-funktio soveltuu käytettäväksi toimintoon, jota ei tarvitse nimetä. Niillä voidaan korvata pieniä funktioita, joita käytetään koodissa vain paikallisesti. Lambda-funktio käy koodissa lausekkeen paikalle. [20]

Seuraavassa listauksessa on esimerkki lambda-funktion käytöstä:

```
int sum(0);
for_each(v.begin(), v.end(), [&sum](int x) {sum += x;});
```

Esimerkissä lasketaan vectorin v alkioiden summa lambda-funktiota käyttäen. Esimerkissä käytetään myös C++11-standardin tuomaa uutta *for_ each*-silmukkarakennetta.

2.3.3 Auto-avainsana

C++11-standardin myötä kielen *auto*-avainsanan merkitystä muutettiin. Edellisissä versioissa *auto*-avainsanalla määritettiin muuttujan elinikä ja näkyvyysalue automaattiseksi. Avainsanaa ei kuitenkaan juurikaan käytetty, sillä paikallisen muuttujan elinikä on oletuksena automaattinen, eikä *auto*-sanan kirjoittaminen tai poisjättäminen muuta ohjelmakoodin toimintaa. [21]

C++11-standardissa *auto*-avainsana määrittelee muuttujan tyypin automaattiseksi. Muuttujan tyyppi päätellään käännösaikana siihen sijoitettavasta arvosta. *Auto*-avainsanan käyttö selkeyttääkin lähinnä syntaksia ja helpottaa koodin kirjoittamista. *Auto*-tyyppiä voidaan käyttää sekä muuttujien alustuksessa että funktion paluuarvon tyyppinä. [22]

Seuraavassa esimerkissä pitkä iteraattorin tyyppimäärittely on korvattu automaattisella tyypillä:

```
std::map<int, std::string> map;
std::map<int, std::string>::iterator iter = map.begin();
auto iter = map.begin();
```

Avainsanaa voidaan käyttää erityisesti C++:n mallien tai lambda-funktioiden kanssa helpottamaan muuttujan tyypin pitkää syntaksia.

3. PILVITEKNOLOGIAT

Tässä luvussa kuvataan teknologiat, joita diplomityössä toteutettava Qubit-sovellus tarvitsee kommunikoidakseen pilviympäristössä toimivan BIT-projektinhallintajärjestelmän kanssa. Aluksi esitellään lyhyesti työtehtävä- ja työtuntidatan siirtoon käytetty JSON-tiedonsiirtoformaatti. Seuraavaksi esitellään verkkosovelluksissa ja -rajapinnoissa yleisesti käytetty REST-arkkitehtuuri. Lopuksi kuvataan OpenID-autentikointiprosessi, jota Qubit-sovellus tarvitsee tunnistautuessaan BIT-palvelimelle.

3.1 JSON

JSON (*JavaScript Object Notation*) on yksinkertainen tiedonsiirtoformaatti, joka perustuu JavaScript-ohjelmointikielen standardiin. JSON on erityisesti käytössä moderneissa web-sovelluksissa dataolioiden välittämisessä verkon yli. Formaatti on kuitenkin ohjelmointikieliriippumaton, ja sen jäsentämiseen on olemassa valmiita kirjastoja kaikille yleisimmille ohjelmointikielille. JSON on tekstimuotoinen, ja sen syntaksissa olioiden jäsenmuuttujat esitetään (nimi, arvo) -pareina. Myös listoja voidaan esittää. [23]

Seuraava yksinkertaistettu esimerkki esittää listan projektiin kuuluvista työtehtävistä (*task*):

```
[
    {"id": 1, "name": "task1", "projectId": 20},
    {"id": 2, "name": "task2", "projectId": 20}
]
```

Jokaisella tehtävällä on tunniste (*id*), nimi (*name*) ja projekti (*projectId*), jolle tehtävä kuuluu.

3.2 REST

Representational State Transfer (*REST*) on verkkosovelluksissa vallitsevasti käytetty arkkitehtuuri. Arkkitehtuuri on kehitetty HTTP1.1-protokollan rinnalla, ja se esiteltiin vuonna 2000. REST on asiakas-palvelin-arkkitehtuuri, joka on verkkosovelluksissa kaikkein yleisimmin käytetty arkkitehtuuriratkaisu. Järjestelmä koostuu yhdestä tai useammasta palvelimesta ja suuresta määrästä asiakkaita. Asiakkaat lähettävät palvelimelle resurssipyyntöjä, joihin palvelin vastaa. [13, s. 136–138]

REST-arkkitehtuuri on asiakas-palvelin-arkkitehtuurin tapaan tilaton. Arkkitehtuuri on yleiskäyttöinen, ja sitä voidaan soveltaa verkkosovelluksissa datan välitykseen asiakkaan ja palvelimen välillä sovellusalueesta riippumatta. Arkkitehtuuri määrittelee viisi eri rajoitetta, jotka järjestelmän on toteutettava. Rajoitteita ovat edellä mainittujen asiakas-palvelin-arkkitehtuurin ja tilattomuuden lisäksi soveltuvuus käyttöön välimuistin kanssa, järjestelmän hierarkkisuus ja yhtenäinen rajapinta. Lisäksi vapaaehtoisena kuudentena rajoitteena on mahdollisuus ladata ohjelmakoodia suoritettavaksi palvelimen sijasta asiakkaalla. [24, s. 76–85]

Palvelimen tarjoaman rajapinnan yhtenäisyyttä ei määritellä tarkemmin, mutta käytännössä REST-rajapinta on joukko resursseja kuvaavia URL-osoitteita. HTTP-pyynnön vastauksessa saadaan resurssin esittämä data. Rajapinnan resurssit voidaan identifioida URL-osoitteilla, ja resursseja voidaan pyytää, muokata ja poistaa. Rajapinta on suunniteltu ohjelmalliseen käyttöön, ja vastaus palautetaan usein helposti jäsennettävässä JSON-muodossa tai XML-dokumenttina. [24, s. 76–106]

URL-osoitteet eivät osoita tiettyyn resurssitiedostoon, vaan osoitteet kuvaavat datan loogista rakennetta. Rajapinta kapseloi palvelimen sisäisen toteutuksen – paluuarvona palautettu data voi olla peräisin staattisesta tiedostosta, tietokannasta tai se voidaan generoida dynaamisesti pyynnön yhteydessä. [24, s. 110–112]

Esimerkki REST-rajapinnan URL-osoitteista on taulukossa 3.1. Rajapinnasta voi HTTP:n *GET*-pyynnöllä pyytää listan palvelimella olevista projekteista tai tietyn projektin tiedot. Myös muita HTTP-metodeja *POST*, *PUT* ja *DELETE* voidaan käyttää lisäämään, muokkaamaan ja poistamaan dataa.

Taulukko 3.1. *REST-esimerkkirajapinta.*

URL	**Kuvaus**
/resource/project	Kaikkien projektien tiedot.
/resource/project/{id}	Projektin {id} tiedot.

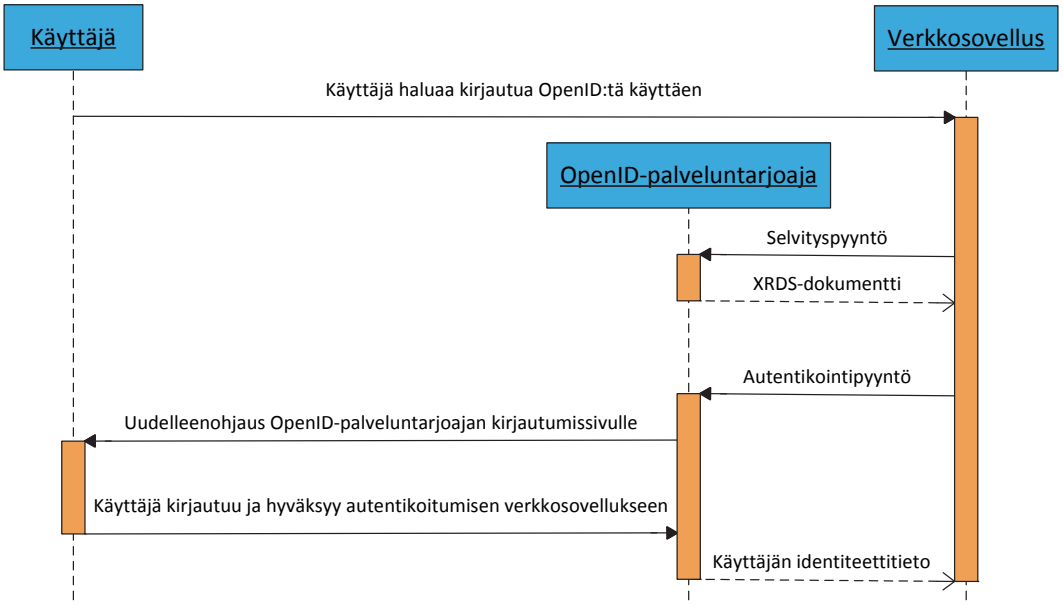
Esimerkiksi *POST*-metodilla voisi ylempää rajapintafunktiota käyttäen lisätä uuden projektin. Alempi funktio puolestaan mahdollistaisi tietyn projektin muokkaamisen ja poiston *PUT*- ja *DELETE*-metodeilla.

REST-arkkitehtuurissa URL-osoitteiden muodostama hierarkia vastaa funktiorajapintaa. Funktioiden parametrit välitetään joko HTTP-pyynnön parametreina tai ne ovat osa URL-osoitetta, kuten esimerkkirajapinnan tapauksessa. Funktion paluuarvo on palvelimen palauttama HTTP-vastaus, joka on jäsennettävissä esimerkiksi JSON-muotoa käytettäessä suoraan ohjelmointikielen olioksi. [25]

3.3 OpenID

OpenID on verkkopalveluissa käytetty avoin standardi käyttäjän tunnistautumiseen eli autentikointiin ja identiteetin varmistamiseen. Alkuperäinen standardi julkaistiin vuonna 2005, ja nykyinen versio 2.0 on julkaistu vuonna 2007. Standardi mahdollistaa käyttäjän autentikoinnin samalla OpenID-tunnuksella useampaan verkkopalveluun, eikä niihin tarvita erillistä rekisteröitymistä tai omaa käyttäjätunnusta ja salasanaa. OpenID-tunnusten myöntäminen on hajautettu, ja mikään yksittäinen taho ei ylläpidä tai tarjoa autentikointipalvelua. OpenID-tunnuksen voi luoda haluamalleen palveluntarjoajalle, joita ovat esimerkiksi verkkosivustot Google, Yahoo, Blogger ja MySpace. [26; 27]

Autentikointiprosessi etenee kuvan 3.1 mukaisesti. Verkkosovellus lähettää selvityspyynnön (*Service Discovery Request*) OpenID-autentikointipalvelun URL-osoitteeseen. Vastaus saapuu *XRDS*-dokumenttina. *XRDS*(*Extensible Resource Descriptor Sequence*) on verkkoresurssin metatiedon kuvaukseen käytetty kieli. Dokumentti sisältää URL-osoitteen, johon lähetetään käyttäjän autentikointipyyntö. [26; 28]



Kuva 3.1. *OpenID-autentikointiprosessin eteneminen.* [28]

Autentikoinnissa käyttäjä uudelleenohjataan verkkosovelluksesta OpenID-palveluntarjoajan sivustolle. Sivusto voi olla esimerkiksi Googlen kirjautumissivu. Kirjautumissivulla käyttäjä kirjautuu sisään kirjoittamalla käyttäjätunnuksensa ja salasanansa. Tämän jälkeen OpenID-palveluntarjoaja varmistaa, että käyttäjä haluaa tunnistautua autentikointia pyytäneeseen verkkosovellukseen ja jakaa tietojaan sen kanssa. Lopuksi käyttäjä uudelleenohjataan alkuperäiseen verkkosovellukseen sisäänkirjautuneena käyttäjänä. [26; 27; 28]

Alkuperäinen verkkosovellus saa OpenID-palveluntarjoajalta tietoja käyttäjän identiteetistä, esimerkiksi koko nimen ja sähköpostiosoitteen. Verkkosovellus ei autentikoinnin yhteydessä saa tietoonsa käyttäjän salasanaa, vaan ainoastaan vakuutuksen siitä, että käyttäjä on autentikoitu. [26; 27; 28]

Käytäntö vaikeuttaa käyttäjätunnuksen ja salasanan väärinkäyttöä ja lisää tietoturvaa. Kuitenkin tunnistautumisen helppous aiheuttaa myös riskejä: koska samoilla tunnuksilla voidaan kirjautua useaan eri palveluun, on OpenID-tunnuksen vuotaminen ulkopuoliselle vakavampaa kuin yksittäisen palvelun käyttäjätunnuksen väärinkäyttö.

Käyttäjä myös totutetaan siihen, että kirjautumisen yhteydessä selain uudelleenohjataan OpenID-palveluntarjoajan kirjautumissivulle. Uudelleenohjaus esimerkiksi täysin Googlen kirjautumissivun näköiselle väärennetylle sivulle voisi saada käyttäjän antamaan epähuomiossa Google-tunnuksensa ulkopuoliselle.

4. NYKYINEN TUNTIKIRJAUSJÄRJESTELMÄ

Tässä luvussa kuvataan Vincit Oy:ssä käytetty nykyinen projektinhallinta- ja tuntikirjausjärjestelmä. Aluksi esitellään projektien hallintaan käytetty verkossa toimiva Jira-projektinhallintajärjestelmä. Seuraavaksi kuvataan nykyisin työtuntien kirjaukseen käytetty *Vtt* (*Vincit Time Tracker*)-tuntikirjaussovellus ja sen ominaisuudet. Lopuksi käsitellään nykyisen järjestelmän vajavaisuuksia ja syitä uuden projektinhallinta- ja tuntikirjausjärjestelmän kehittämiselle.

4.1 Yleiskuvaus

Ennen nykyisen järjestelmän kehittämistä juuri toimintansa aloittaneen yrityksen työntekijöiden tuntikirjanpitoa pidettiin Excel-taulukkolaskentapohjaa käyttäen. Nopeasti syntyi kuitenkin laskutusta varten tarve työntekijä- ja projektikohtaisille tuntiraporteille. Nykyinen järjestelmä perustuu valmiin Jira-projektinhallintajärjestelmän [29] käyttöön.

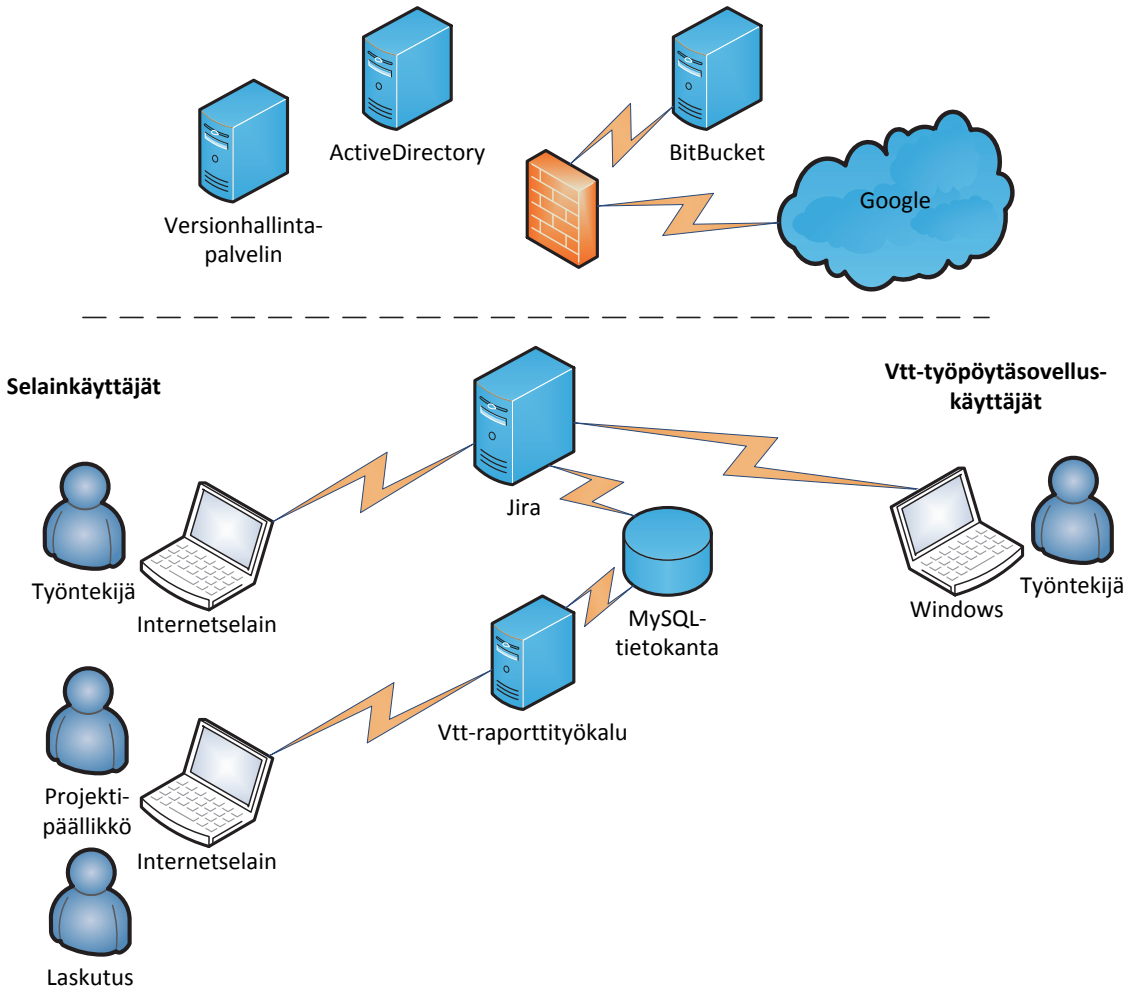
Järjestelmä valittiin aikanaan käyttöön, koska siinä oli valmiina tarvittavat käsitteet projekteille, projektien sisältämille työtehtäville ja mahdollisuus tuntien kirjaamiselle. Toimiva järjestelmä haluttiin saada aikaan mahdollisimman pienellä työmäärällä, sillä tuottavaa laskutettavaa työtä haluttiin tehdä mahdollisimman tehokkaasti. Kaikki silloiset työntekijät olivat sulautettujen järjestelmien erikoisosaajia, ja yrityksen sisällä ei tuolloin ollut resursseja oman liiketoimintatietojärjestelmän kehittämiseen.

Jiran rinnalle toteutettiin palvelinsovelluksena Vtt-raporttityökalu, jonka avulla on mahdollista luoda raportteja projektien ja työntekijöiden kokonaistunneista laskutusta varten. Jiraan toteutettiin myös liitännäinen, joka mahdollisti työntekijän projektien ja työtehtävien hakemisen sekä tuntien syöttämisen ohjelmallisen rajapinnan kautta. Tuntikirjaukseen kehitettiin Vtt-työpöytäsovellus tuolloin ainoana käytetylle Windows-käyttöjärjestelmälle.

Nykyisen järjestelmän yleiskuva on esitetty kuvassa 4.1. Kuvan keskellä on valmiina käyttöönotettu Jira-projektinhallintajärjestelmä, joka käyttää MySQL-tietokantaa [30] tietojen tallentamiseen. Raportointiin kehitetty Vtt-raporttityökalu käyttää Jiran tietokantaa tuntikirjausten lukemiseen ja tuottaa halutut raportit, joista tuntimäärät voidaan katsoa laskutusta varten.

Vtt-työpöytäsovellusta voidaan käyttää tuntien automaattiseen kirjaamiseen vain

Windows-käyttöjärjestelmällä. Muilla käyttöjärjestelmillä tuntikirjaus on tehtävä Jiran selainkäyttöliittymän kautta. Projektipäälliköt ja laskutus käyttävät työssään Vtt-raporttityökalun tuottamia tuntiraportteja.



Kuva 4.1. *Nykyisen järjestelmän yleiskuva.*

Projektinhallinta- ja tuntikirjausjärjestelmän lisäksi yrityksen päivittäiseen toimintaan liittyy kuitenkin monia muitakin sisäisiä ja ulkoisia järjestelmiä, joista osa on esitetty kuvassa. Uuden projektin alkaessa projektipäällikkö tai IT-tuki luo projektille versionhallinnan ja lisää projektin jäsenille tarvittavat oikeudet projektin jaettuihin *Active Directory* -kansioihin. Osa projekteista pitää tietovarastoa, tehtävänhallintaa ja virheiden raportointia myös ulkoisessa BitBucket-palvelussa. Myös Googlen pilvipalveluita käytetään dokumenttien ja projektikansioiden jakamiseen. Uuden projektin perustaminen, tarvittavien oikeuksien myöntäminen ja asetusten tekeminen on pitkälti käsityötä.

Seuraavissa alakohdissa kuvataan tarkemmin Jira-projektinhallintajärjestelmä ja erityisesti tuntikirjauksiin käytetty Vtt-työpöytäsovellus, jonka pohjalta diplomityössä toteutettavaa Qubit-sovellusta kehitetään.

4.2 Jira

Projektien hallintaan on Vincit Oy:ssä aiemmin käytetty web-pohjaista Jira-projektinhallintajärjestelmää. Jira on Atlassianin kehittämä ja vuonna 2002 julkaistu tehtävänhallintaohjelmisto. Jira on maksullinen ohjelmisto, ja sitä käytetään laajalti ohjelmistoalan yrityksissä tehtävien seurantaan, projektien hallintaan ja ohjelmistovirheiden raportointiin. [29]

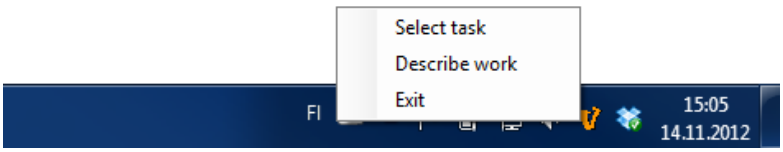
Jira mahdollistaa projektien luonnin ja projektien jakamisen osiin. Projektille voidaan lisätä uusia tehtäviä, uusia lisättäviä ominaisuuksia ja kehitysehdotuksia. Projektille voidaan myös raportoida korjausta vaativia ohjelmistovirheitä. Tehtävät voidaan määrätä tietylle henkilölle ja kirjata käytetyt työtunnit. Yksittäisten tehtävien seurannan lisäksi voidaan seurata koko projektin etenemistä, ajankäyttöä ja jäljellä olevaa työmäärää. [29]

4.3 Vtt

Vtt on Vincit Oy:n käyttämä Windows-työpöytäsovellus työajan automaattiseen kirjaamiseen. Sovellus on kehitetty yksittäisen työntekijän projektina, ja se käyttää .NET-teknologiaa. Sovellus on käynnissä taustalla, ja se mittaa eri tehtäville käytettyä työaikaa. Työpäivän tai -viikon päätteeksi sovelluksesta voidaan lähettää tehdyt työtunnit automaattisesti Jira-projektinhallintajärjestelmään.

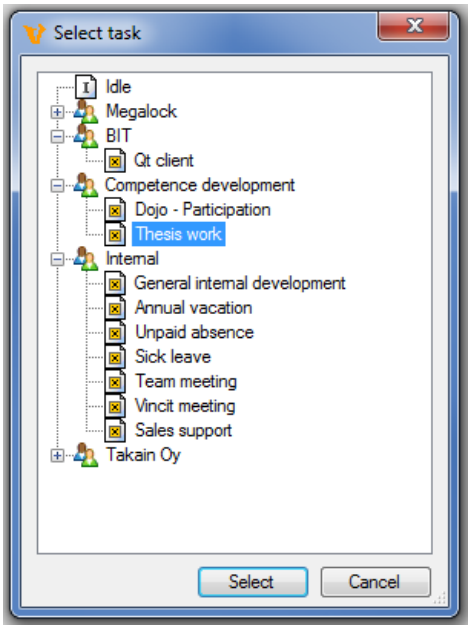
Sovelluksesta valitaan nykyinen projekti ja tehtävä, minkä jälkeen ajan mittaaminen käynnistyy. Vtt tarkkailee myös tietokoneen tilaa ja keskeyttää työajan kirjauksen, kun tietokoneen työpöytä lukitaan. Sovellus säilyttää työaikatietoa paikallisessa tietokannassa. Automaattisesti kirjattuja tunteja voi jälkikäteen jakaa pienempiin osiin, ja myös kirjausten lisääminen on mahdollista. Muokkausten jälkeen tunnit lähetetään Jira-projektinhallintajärjestelmään, jolloin ne päätyvät laskutukseen.

Kuvassa 4.2 on Windowsin tehtäväpalkissa tausta-ajossa oleva Vtt-sovellus. Sovelluksesta voidaan valita nykyinen työtehtävä valitsemalla *Select task*. Tehtyjä työtunteja voi muokata valitsemalla valikosta *Describe work*.



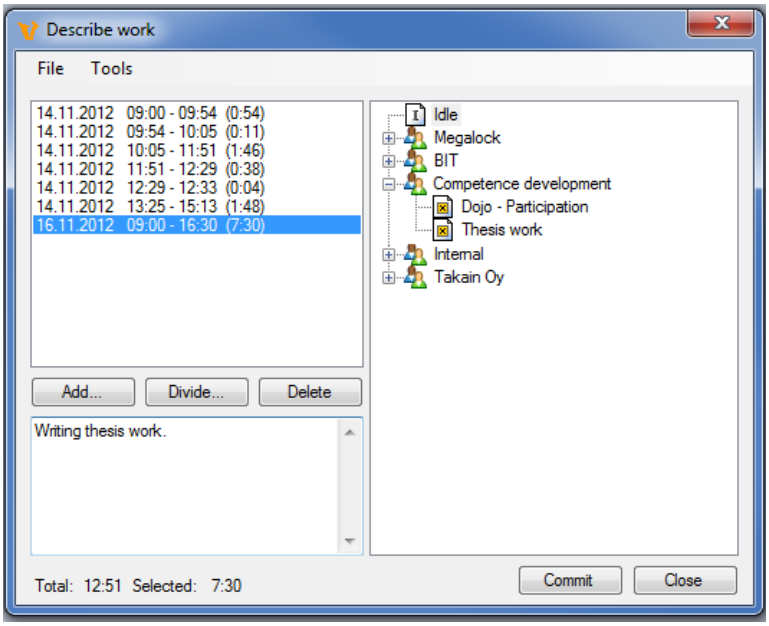
Kuva 4.2. *Vtt:n käynnistys taustalta.*

Nykyisen työtehtävän valintaikkuna on esitetty kuvassa 4.3. Ikkunassa on puumainen projektirakenne, josta valitaan tietyn projektin tehtävä parhaillaan suoritettavaksi. Työtehtävän tekeminen alkaa, ja ajan mittaaminen käynnistyy.



Kuva 4.3. *Vtt:n näkymä nykyisen työtehtävän valitsemiseen.*

Tuntikirjausten muokkausnäkymässä kuvassa 4.4 näkyvät tehdyt työtunnit, joita ei ole vielä lähetetty palvelimelle. Näkymässä on mahdollista kirjoittaa tuntikirjaukselle selite, poistaa ja lisätä tuntikirjauksia sekä jakaa olemassa olevia kirjauksia osiin.



Kuva 4.4. *Vtt:n näkymä työtuntien muokkaukseen ja lähetykseen.*

Muokkausten jälkeen työtunnit lähetetään Jira-projektinhallintajärjestelmään työtuntien muokkausnäkymästä. Tuntien lähettämisen jälkeen tuntikirjaukset poistuvat käyttäjän tietokoneelta.

4.4 Kohti uutta järjestelmää

Yrityksen kasvaessa ja projektien määrän lisääntyessä tarvitaan uusi aiempaa käytettävämpi ja automatisoidumpi järjestelmä. Vtt-tuntikirjaussovellus on olemassa ainoastaan Windows-tietokoneille, mutta suuri osa työntekijöistä käyttää Linux- tai Mac-tietokonetta. Merkittävä osa yrityksen työntekijöistä ei siis voi käyttää nykyistä Vtt-sovellusta.

Tuntikirjausten tekeminen ja muokkaaminen Jiran selainkäyttöliittymän kautta on kankeaa. Myöskään työaikaa ei ilman Vtt-sovellusta voida automaattisesti mitata taustalla. Jira ei myöskään mahdollista laskutuksen automatisointia, vaan laskutus hoidetaan edelleen Vtt-raporttityökalun raportoimien tuntien pohjalta taulukkolaskentaohjelmassa.

Projektien määrän kasvaessa myös uuden projektin luomiseen liittyvät toimet aiheuttavat enemmän työtä. Jokaista projektia varten on luotava versionhallinta, jaettu kansio projektin dokumentaatiolle ja projekti Jira-projektinhallintajärjestelmään. Lisäksi on annettava tarvittavat käyttöoikeudet projektin työntekijöille ja mahdolliset oikeudet asiakkaan kanssa jaettuun julkiseen tietovarastoon tai versionhallintaan.

Edellä kuvattu käsin tehtävä laskutus ja uuden projektin perustaminen halutaan automatisoida niin pitkälle kuin mahdollista. Tämän vuoksi Vincit Oy:ssä on sisäisenä projektina kehitetty pilviympäristössä toimiva BIT (*Business Intelligence Toolkit*) -projektinhallintajärjestelmä. Järjestelmän alustava versio on otettu käyttöön, ja siihen kehitetään uusia ominaisuuksia projektien luonnin ja laskutuksen automatisoimiseksi.

Nykyisessä järjestelmässä tuntikirjauksia tehdään projektien laskutusta varten. Siksi Vtt:n ja Jiran työaikakirjaukset pyöristetään ylöspäin lähimpään laskutettavaan yksikköön. Tämän vuoksi Jira ja Vtt eivät mahdollista työntekijöiden todellisten työtuntien seurantaa. Vincit Oy:n työaikamalli on vapaa eikä työaikaa valvova. Liukuman tai ylitöiden käsitteitä ei pääsääntöisesti ole. Tarve todellisen työajan seuraamiselle tuleekin työntekijöiltä itseltään.

BIT-järjestelmän kehittämisen yhteydessä myös Vtt-työpöytäsovellus halutaan uudistaa tekemällä uusi tässä diplomityössä toteutettava Qubit-sovellus. Uusi sovellus halutaan käytettäväksi Windows-käyttöjärjestelmän lisäksi yhtenevästi myös Linux- ja Mac OS X -käyttöjärjestelmillä.

Lisäksi nykyisen sovelluksen pohjalta tehdään uuteen sovellukseen joitakin käytettävyysparannuksia. Edellisen Vtt:n kahden eri ikkunan – nykyisen työtehtävän valinnan ja tehtyjen työtuntien muokkauksen – sijaan toteutetaan yksi ikkuna, jossa on samassa näkymässä painikkeet uuden tehtävän aloittamiselle ja lopettamiselle sekä mahdollisuus muokata tehtyjä työtunteja.

Uudessa sovelluksessa halutaan myös aiempaa selkeämmin näkyviin, mikä on parhaillaan valittuna oleva työtehtävä. On myös hyödyllistä nähdä, paljonko aikaa on kulunut nykyisessä työjaksossa ja yhteensä koko päivän aikana.

Nykyisestä työaikalokista ei myöskään näe selkeästi, mille tehtävälle työaika on kirjattu. Käytettävyysparannuksena näytetään selkeämmin korostettuna valittu työtehtävä. Nykyisessä näkymässä on myös työtehtävän selitteelle vain tyhjä tekstikenttä, jonka merkitys on osalle työntekijöistä epäselvä. Uudessa versiossa vapaamuotoista selitettä ei ainakaan alkuvaiheessa näytetä lainkaan.

Työaikakirjausten muokkaus tekstimuotoisena listana on hankalaa. Listasta on vaikea nähdä, menevätkö tehtävät ajallisesti päällekkäin tai onko niiden välissä ylimääräisiä taukoja. BIT:iin tuleekin tätä varten kalenterimainen graafinen käyttöliittymä. Listaa parannetaan myös Qubit-sovellukseen jaottelemalla eri päivinä tehdyt tunnit selkeästi erilleen toisistaan. Lisäksi toistensa kanssa päällekkäin menevät tuntikirjaukset näytetään korostettuna.

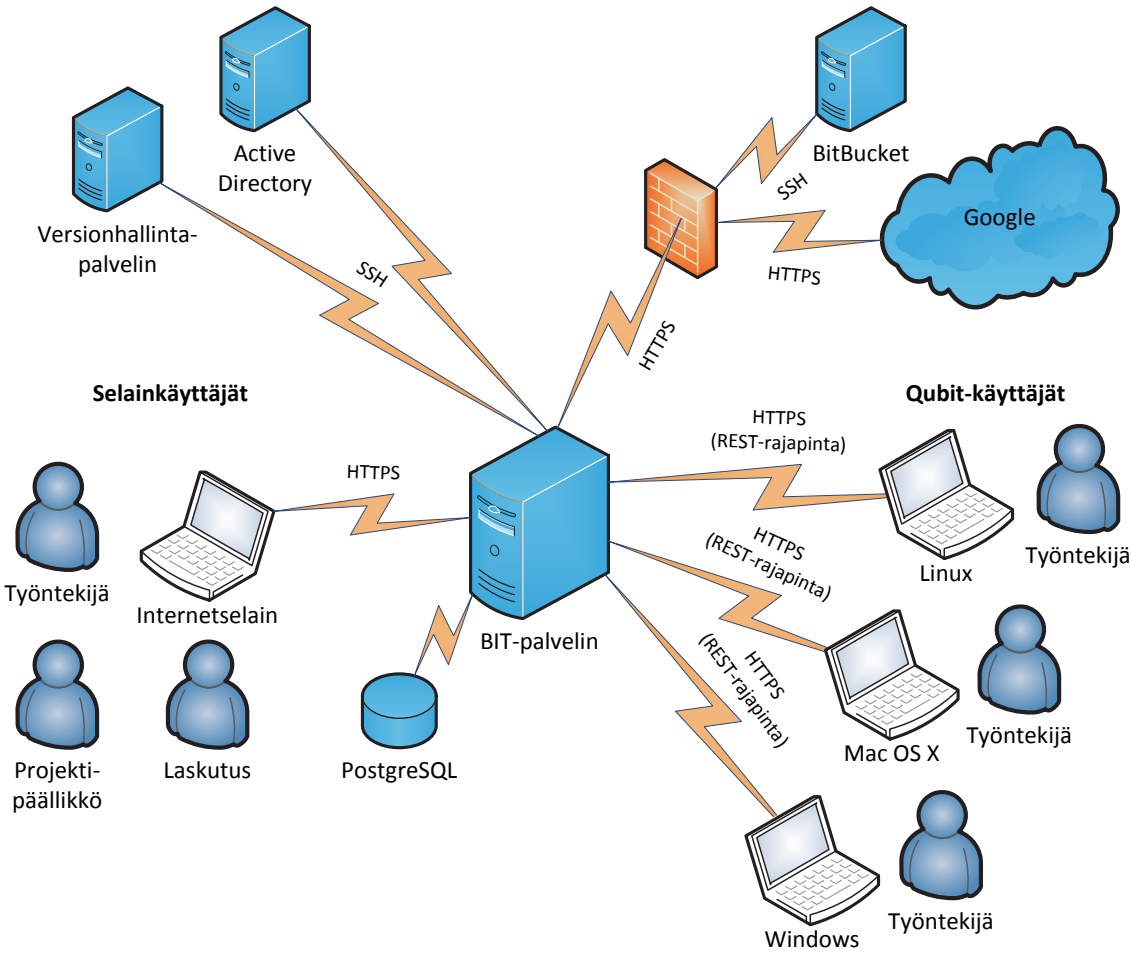
Vaikka yrityksessä siirrytään uuden BIT-järjestelmän käyttöön, on Jira kuitenkin toimiva tehtävänhallintaohjelmisto. Osa projekteista jatkaa Jiran käyttämistä ohjelmistovirheiden ja uusien ominaisuusvaatimusten kirjaamiseen. Jira ei kuitenkaan ole kokonaisvaltainen projektinhallinta- ja liiketoimintatietojärjestelmä, ja BIT korvaa sen siinä käytössä. BIT pyrkii integroimaan yrityksen eri järjestelmät yhteen ja vähentämään projektinhallintaan liittyvän käsin tehtävän työn määrää.

5. UUSI TUNTIKIRJAUSJÄRJESTELMÄ

Tässä luvussa kuvataan yrityksen uusi projektinhallinta- ja tuntikirjausjärjestelmä. Aluksi kuvataan koko järjestelmän toiminta yleisellä tasolla. Seuraavaksi esitellään lyhyesti verkossa toimiva BIT-projektinhallintajärjestelmä. Järjestelmästä kuvataan sen Qubit-sovellukselle tarjoama rajapinta ja autentikointi. Lopuksi kuvataan uuden Qubit-sovelluksen arkkitehtuuri ja toteutusyksityiskohdat.

5.1 Yleisarkkitehtuuri

Uusi projektinhallintajärjestelmä yhdistää yrityksessä käytetyt erilliset palvelut yhtenäiseksi kokonaisuudeksi. Uuden järjestelmän yleiskuva on esitetty kuvassa 5.1.



Kuva 5.1. *Uuden järjestelmän yleiskuva.*

BIT-palvelin sisältää tiedot yrityksen asiakkaista, projekteista ja työntekijöistä. BIT-palvelinsovellukseen on pyritty integroimaan kaikki projektinhallintaan, työajan raportointiin ja laskutukseen liittyvät toiminnot.

Palvelinsovellus tarjoaa selainkäyttöliittymän, jonka kautta järjestelmää hallitaan. Selainkäyttöliittymän lisäksi palvelin tarjoaa ohjelmallisen REST-rajapinnan, jota diplomityössä toteutettava Qubit-sovellus käyttää kommunikoidessaan BIT-projektinhallintajärjestelmän kanssa.

BIT-palvelin käyttää käyttäjien tunnistautumiseen Googlen OpenID-palvelua. Uuden projektin luonnin yhteydessä BIT-sovellus luo Googleen projektin jäsenille jaetun projektikansion. Sovellus luo lisäksi projektille versionhallintajärjestelmän ja antaa sen käyttöoikeudet projektiin kuuluville työntekijöille.

Työntekijät voivat kirjata työtuntejaan sekä selainkäyttöliittymästä että käyttämällä Qubit-sovellusta. Kukin työntekijä käyttää tuntien kirjaamiseen helpoimmaksi katsomaansa tapaa. Yleinen käyttötapa on käyttää Qubit-sovellusta toimistolla tehtyjen työtuntien automaattiseen kirjaamiseen ja selainkäyttöliittymää esimerkiksi työmatkojen, asiakaspalavereiden tai muulla kuin tietokoneella tehtävän työn yhteydessä. Qubit-sovellusta käyttävillä työntekijöillä voi olla Windows-, Linux- tai Mac OS X -käyttöjärjestelmä.

5.2 BIT-palvelin

BIT-palvelinsovelluksen käyttöliittymä mahdollistaa tuntiraporttien näyttämisen ja automaattisen laskujen luonnin. Käyttöliittymän kautta on mahdollista myös kirjata työtunteja helpommin kuin aiemmassa Jira-projektinhallintajärjestelmässä. Projektinhallintapuolella voidaan sijoittaa työntekijöitä projekteihin tietyillä viikkotuntimäärillä ja aikatauluttaa projekteja. Uusia tulevia projekteja sekä työntekijäresurssien käyttöastetta ja sijoittelua voidaan suunnitella tulevaisuuteen.

Työntekijäkohtaisesti on mahdollista hakea työntekijän projektit ja työtehtävät, kirjata työtunteja ja hakea palvelimelle kirjatut tunnit. Lisäksi on mahdollista tarkastella, mikä on työntekijän nykyinen ja suunniteltu työmäärä eri projekteissa.

Yritysorganisaatiossa ja palvelinsovelluksen tietomallissa työntekijät on jaettu tiimeihin. Tiimeittäin voidaan tarkastella tiimin käyttöastetta, työtuntiraportteja, tiimin jäsenten tietoja ja työntekijöiden sijoittelua eri projekteihin. Projektinhallintapuolella voidaan luoda uusia projekteja ja muokata projektien työtehtäviä. Lisäksi voidaan muokata työntekijöiden projektikiinnityksiä sekä tarkastella projektin budjettia ja projektille kirjattuja työtunteja.

Palvelinsovellus tarjoaa web-käyttöliittymän lisäksi ohjelmallisen REST-rajapinnan, joka mahdollistaa edellä kuvatut toiminnot. Diplomityössä toteutettava Qubit-sovellus käyttää REST-rajapintaa kommunikoidessaan BIT-palvelimen kanssa. Rajapinta mahdollistaa sovellukselle työntekijän tietojen, näytettävien projektien ja

työtehtävien hakemisen sekä työtuntien kirjaamisen.

Palvelin on toteutettu Java Spring MVC -ohjelmistokehystä [31] käyttäen. Tietojen tallentamiseen käytetään PostgreSQL-tietokantaa [32]. Käyttöliittymän näkymät on toteutettu siten, että palvelimelta ladataan sivun runko ja Javascript-koodia selaimessa suoritettavaksi. Javascript-koodi käyttää palvelimen REST-rajapintaa sivun varsinaisen tietosisällön hakemiseen. Sivun osia voidaan näin päivittää palvelimelta dynaamisesti, eikä pienen muutoksen vuoksi ole tarvetta koko sivun uudelleenlataamiselle.

5.2.1 REST-rajapinta

Palvelimen REST-rajapinnan Qubit-sovelluksen kannalta oleelliset funktiot ovat työntekijöiden projektien ja tehtävien kysely ja tuntikirjauksen lähetys. Myös työntekijän omien tietojen kyselyä käytetään sovelluksen avaamisen yhteydessä testaamaan, että kirjautuminen onnistuu. Rajapinnassa funktioiden parametrit ja paluuarvot lähetetään JSON-muodossa.

Työntekijän tietojen kysely suoritetaan taulukossa 5.1 kuvatulla funktiolla. Taulukossa kuvattuun resurssiin lähetetty HTTP GET -pyyntö palauttaa JSON-muodossa työntekijän nimen, sähköpostiosoitteen ja muita työntekijän tietoja.

Taulukko 5.1. *Työntekijän omien tietojen haku palvelimelta.*

URL	/resource/user/me
Metodi	GET
Parametrit	-
Paluuarvo	Työntekijän tiedot.

Työntekijälle näytettävät projektit ja työtehtävät voidaan kysyä taulukossa 5.2 kuvatulla funktiolla. Taulukossa kuvattuun resurssiin lähetetty HTTP GET -pyyntö palauttaa JSON-muodossa listan näytettävistä projekteista ja projektin sisällä listan sen sisältämistä työtehtävistä.

Taulukko 5.2. *Työntekijän projektien ja tehtävien haku palvelimelta.*

URL	/resource/user/me/tasks
Metodi	GET
Parametrit	-
Paluuarvo	Lista projekteista ja tehtävistä.

Työtuntien kirjaus tehdään taulukossa 5.3 kuvatulla funktiolla. Taulukossa kuvattuun resurssiin lähetetty HTTP POST -pyyntö lisää työntekijän työaikalokiin

uuden kirjauksen. Pyynnön parametreina ovat tehty työaika, työtehtävän tunniste ja työn alkamisajankohta.

Taulukko 5.3. *Työntekijän tuntikirjauksen lähetys palvelimelle.*

URL	/resource/user/me/worklogs
Metodi	POST
Parametri	minutesWorked – työaika minuutteina.
Parametri	taskId – tehtävän tunniste.
Parametri	workStarted – aloitushetken aikaleima.
Paluuarvo	HTTP statuskoodi.

Alla on esimerkki työtuntikirjauksen lähetyksessä käytettävästä JSON-viestistä:

```
{ "minutesWorked": 60, "taskId": 9, "workStarted": 1360502254152 }
```

Viestissä työtehtävälle 9 kirjataan 10.2.2013 klo 13:17 alkaen 60 minuuttia työaikaa. Työn alkamisajankohta on esitetty epoch-aikaleimana.

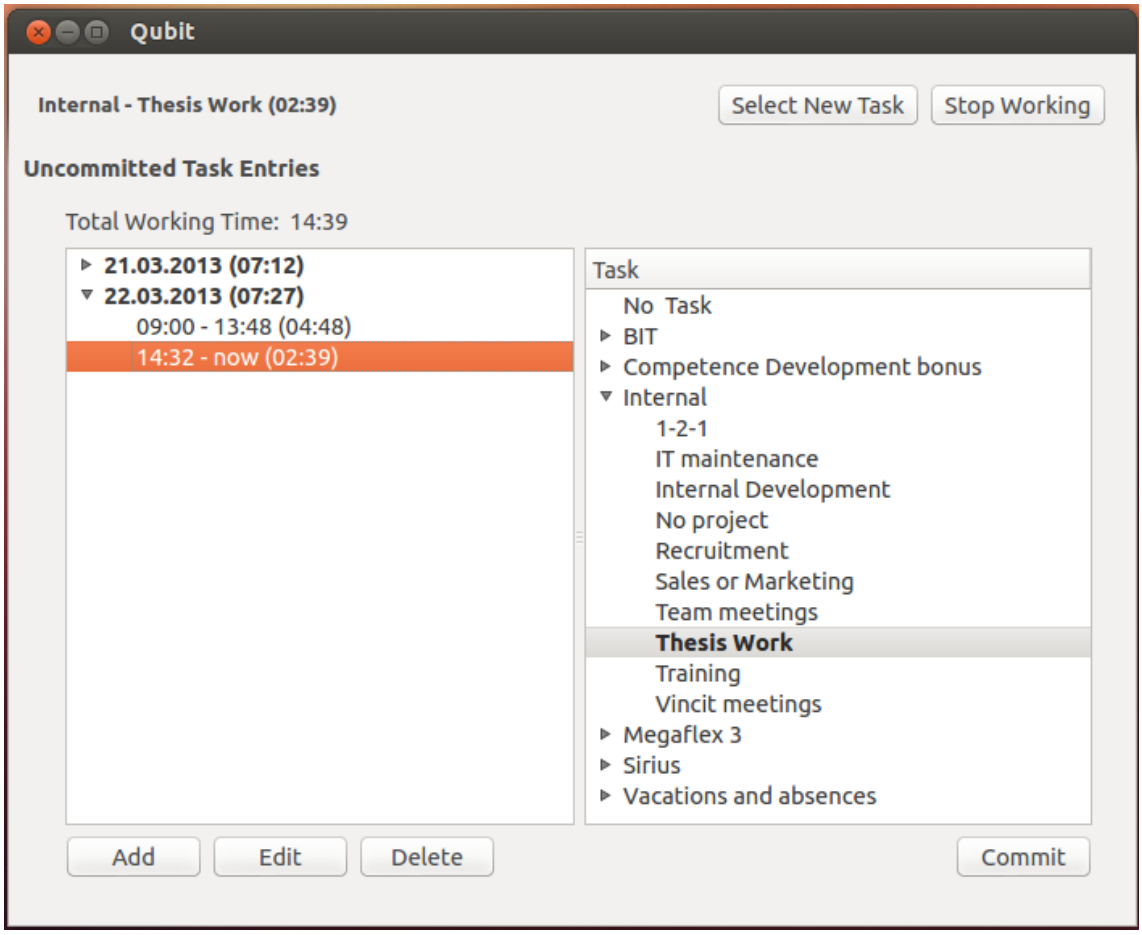
5.2.2 Autentikointi

BIT-palvelinsovellus käyttää käyttäjien autentikointiin Googlen OpenID-palvelua. Selainkäyttöliittymään kirjautuminen onnistuu siis työntekijän Google-tunnuksilla. REST-rajapinta käyttää HTTP Basic -autentikointia, mutta ensimmäisellä kirjautumiskerralla on tunnistauduttava manuaalisesti Googlen OpenID-palvelussa.

Tunnistautuminen tapahtuu kirjautumalla sisään käyttäjän Google-tunnuksilla. Tämän jälkeen rajapinnan käyttö onnistuu HTTP Basic -autentikoinnilla käyttämällä ensimmäisen kirjautumiskerran yhteydessä saatua käyttäjätunnusta ja salasanaa. Rajapinnan automaattinen käyttö onnistuu samoilla tunnuksilla niin kauan, kunnes palvelimen asettama tunnuksen voimassaoloaika päättyy.

5.3 Qubit-asiakassovellus

Qubit-sovellus on Qt:lla toteutettu alustariippumaton työpöytäsovellus työajan mittaamiseen ja työtuntien kirjaamiseen. Sovellus kommunikoi erillisellä liitännäisellä BIT-palvelimen kanssa. Palvelimelta haetaan näytettävät projektit ja työtehtävät ja lähetetään palvelimelle sovelluksella kirjatut työtunnit. Sovelluksen käyttöliittymä on esitelty kuvassa 5.2. Käyttöliittymässä on pyritty siihen, että kaikki useimmiten tarvittava tieto olisi näkyvissä samassa näkymässä.



Kuva 5.2. *Qubit-sovelluksen käyttöliittymä.*

Parhaillaan käynnissä oleva työtehtävä ja sen kesto näkyvät sovelluksen yläreunassa. Työtehtävää voi vaihtaa painamalla oikean yläreunan painiketta *Select New Task*. Painike avaa puunäkymän projekteista ja työtehtävistä, joista voidaan kaksoisnapsauttamalla valita uusi työtehtävä.

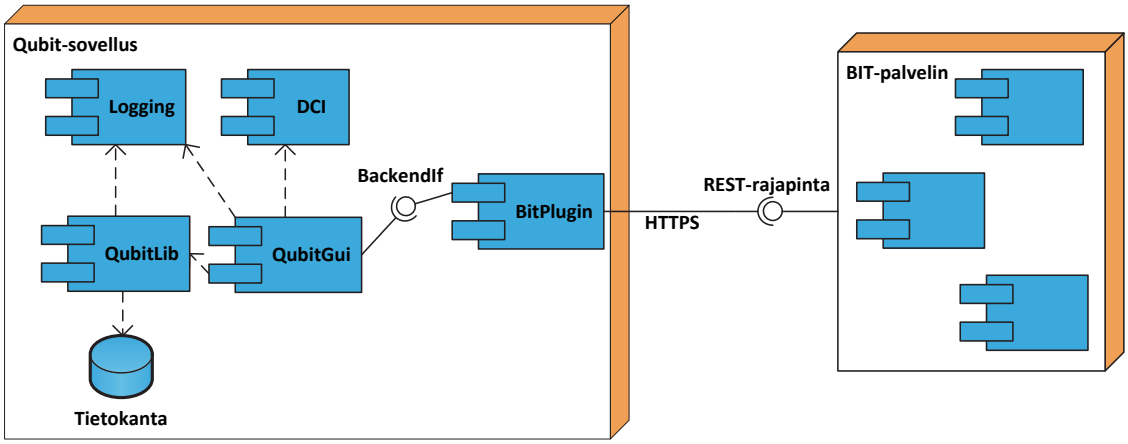
Sovellus tunnistaa, milloin tietokonetta ei käytetä, ja päättelee työntekijän olevan poissa koneelta. Työntekijän palatessa sovellus kysyy, oliko työntekijä tauolla, jatkamassa edellistä työtehtävää tai tekemässä toista tehtävää.

Ikkunan vasemmalla puolella on lista tehdyistä työtunneista, joita ei ole vielä lähetetty palvelimelle. Lista on selkeyden vuoksi ryhmitelty päiväkohtaisesti. Valittuna olevan tuntikirjauksen työtehtävää voi vaihtaa valitsemalla oikean reunan työtehtäväpuusta toisen työtehtävän.

Kirjauksen alku- ja loppuaikaa voi muokata kaksoisnapsauttamalla kyseistä tuntikirjausta. Tuntikirjauksien kokonaisaika näkyy listan yläpuolella. Lisäksi jokaisen päivän kohdalla näkyy suluissa kyseisen päivän työaika. Tehdyt työtunnit voi lähettää palvelimelle painamalla oikeassa alareunassa olevaa *Commit*-painiketta.

5.3.1 Yleisarkkitehtuuri

Sovelluksen yleisarkkitehtuuri on esitetty kuvassa 5.3. Sovelluksen käyttöliittymä ja tietomalli on toteutettu DCI-arkkitehtuuria käyttäen. Sovelluksen kommunikointi taustajärjestelmän kanssa on toteutettu erillään muusta sovelluksesta käyttäen liitännäisarkkitehtuuria. *BackendIf*-rajapinta yhdistää liitännäisen muuhun sovellukseen. Diplomityössä toteutettu BitPlugin-liitännäinen kommunikoi BIT-palvelimen tarjoaman REST-rajapinnan kanssa. Liitännäisen toiminta on kuvattu tarkemmin alakohdassa 5.3.3.



Kuva 5.3. *Qubit-sovelluksen yleisarkkitehtuuri.*

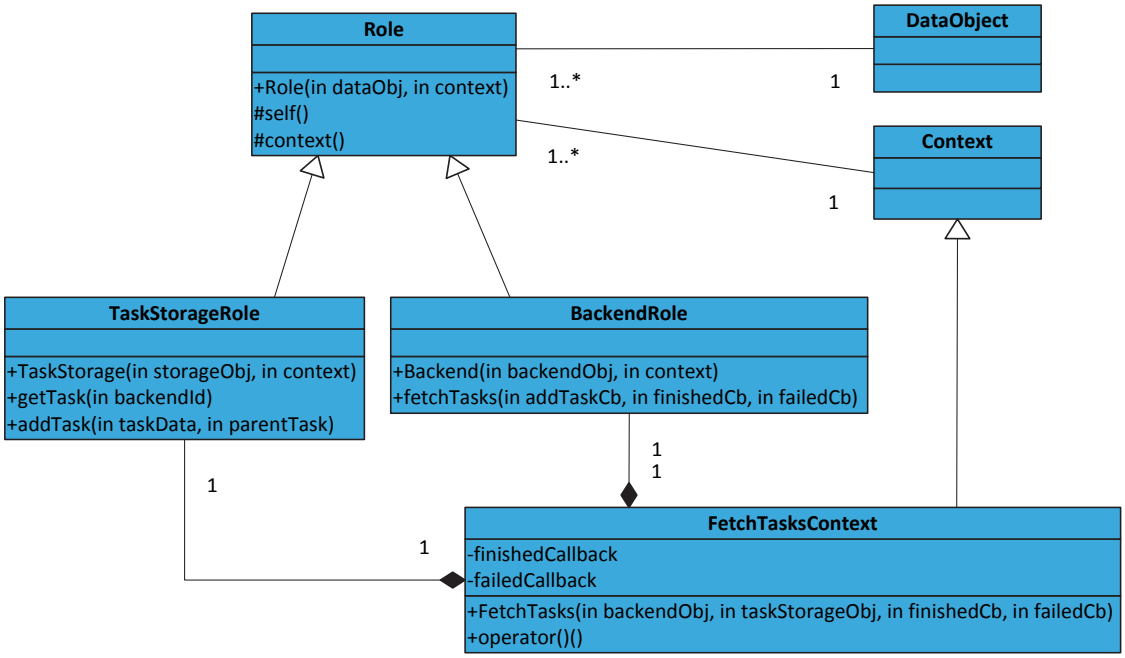
Qubit-sovellus on sisäisesti jaettu useampaan moduuliin. *QubitGui*-moduuli sisältää Qt-käyttöliittymäkoodin. *DCI*-moduulissa on kantaluokkatoteutukset DCI-arkkitehtuurimallin rooleille ja konteksteille.

Sovelluksen toimintalogiikka on koottu *QubitLib*-moduuliin. Moduuli sisältää siten myös toteutukset DCI-arkkitehtuurin rooleille ja konteksteille. *Logging*-moduuli sisältää lähinnä debug-tarkoitukseen kirjoitettavaan tapahtumalokiin liittyvät funktiot. Lisäksi sovelluksessa on paikallinen tietokanta, johon palvelimelta haetut työtehtävät ja tehdyt työtunnit tallennetaan, kunnes ne on lähetetty palvelimelle.

5.3.2 DCI-käyttöliittymän toiminta

Tässä alakohdassa esitetään DCI-arkkitehtuurin toiminnasta esimerkkinä työtehtävien hakuun liittyvä käyttötapaus ja kuvataan sen avulla arkkitehtuurin toimintaa. Kuvassa 5.4 on esitettynä esimerkkikäyttötapaukseen liittyvä luokkakaavio. *FetchTasks*-konteksti on vastuussa työtehtävien haku -käyttötapauksen suorittamisesta. Kontekstissa ovat läsnä *Backend*- ja *TaskStorage*-roolit. *Backend*-rooli kommunikoi taustajärjestelmäliitännäisen kanssa. *BitPlugin*-liitännäinen kommunikoi edelleen BIT-palvelimen REST-rajapinnan kanssa. *TaskStorage*-rooli tallentaa työteh-

tävät Qubitin paikalliseen tietokantaan ja Qt:n tietomalliluokkaan (*QAbstractItemModel*), josta työtehtävät näytetään käyttöliittymässä puurakenteena.



Kuva 5.4. *Työtehtävien haku -kontekstiin liittyvä DCI-luokkakaavio.*

Kun käyttötapaus alkaa, luodaan *FetchTasks*-konteksti. Kontekstin rakentajassa sidotaan *Backend*- ja *TaskStorage*-rooleja näyttelevät objektit. Lisäksi rakentajaparametrina on takaisinkutsufunktiot, joita kutsutaan, kun työtehtävien haku on suoritettu onnistuneesti loppuun tai se epäonnistuu. Epäonnistumisen sattuessa näytetään käyttäjälle virheilmoitus. Kun konteksti on luotu, kutsutaan kontekstin funktio-operaattoria (C++:n *operator()*), joka aloittaa käyttötapauksen.

Roolin *Role*-kantaluokan rakentaja saa parametreinaan viitteen roolia näyttelevään dataolioon ja kontekstiin, johon rooli kuuluu. Roolilla on sisäinen jäsenfunktio *self*, jonka avulla rooli voi viitata siihen dataolioon, joka roolia näyttelee. Roolilla on myös jäsenfunktio *context*, joka viittaa parhaillaan suoritettavaan kontekstiin eli käyttötapaukseen.

Rooliin liittyy tasan yksi dataolio, joka roolia näyttelee. Yksi dataolio voi kuitenkin toimia useammassa roolissa. Samaten rooliin liittyy kerrallaan tasan yksi konteksti, mutta kontekstiin liittyy useampi käyttötapaukseen osallistuva rooli.

Backend-roolilla on yksi ainoa julkinen jäsenfunktio, jonka avulla pyydetään taustajärjestelmäliitännäiseltä työntekijän työtehtävät. Funktion *fetchTasks* funktiokutsu on asynkroninen, ja siten taustajärjestelmäliitännäiseltä saadaan työtehtävät takaisinkutsufunktiota käyttäen.

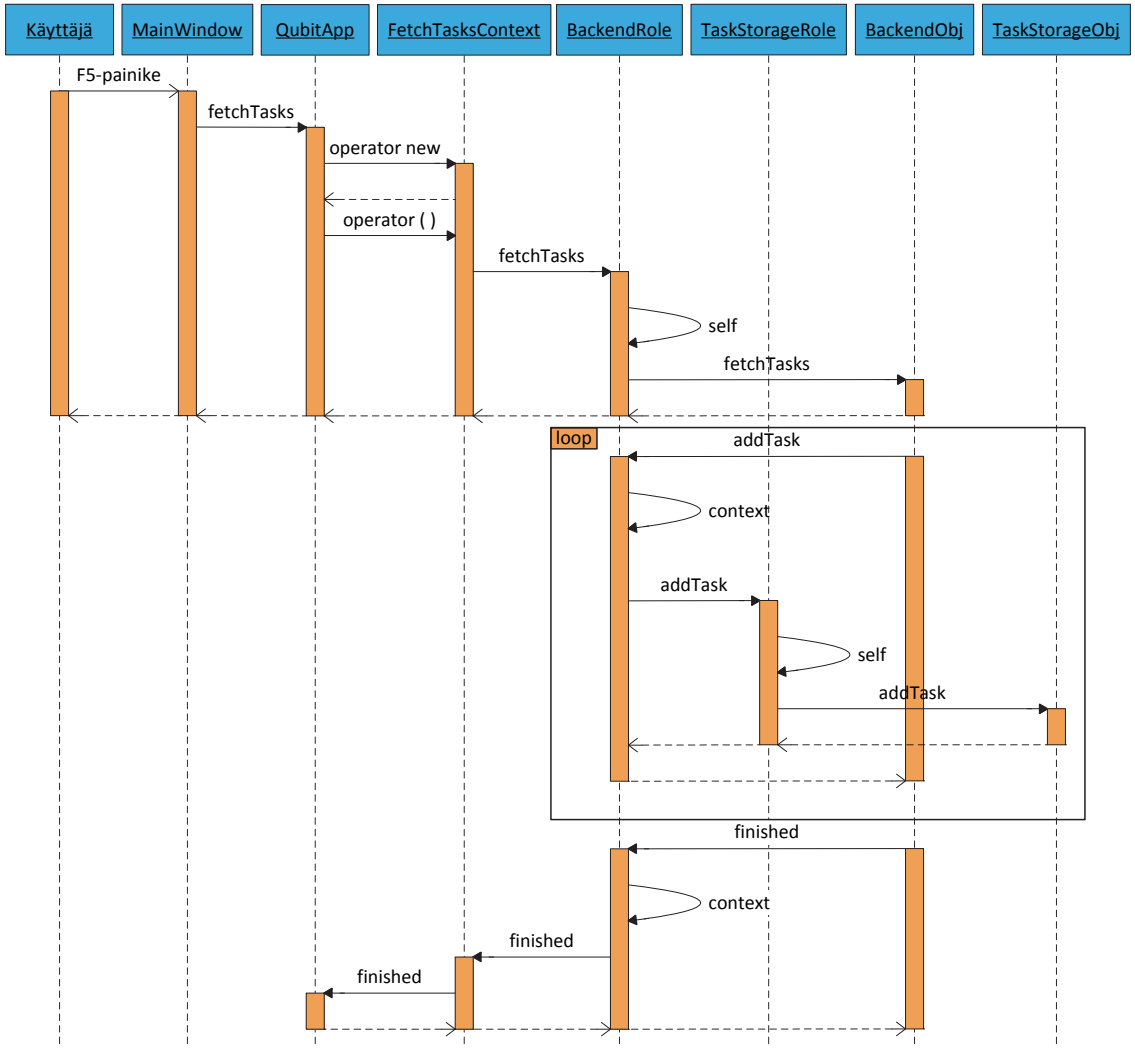
TaskStorage-roolilla on jäsenfunktiot työtehtävien lisäämiseen ja muokkaamiseen. Funktio *getTask* palauttaa työtehtävän tunnisteen (*id*) perusteella osoittimen työ-

tehtäväolioon. Tämän jälkeen työtehtävää voidaan muokata. Funktio *addTask* lisää uuden tehtävän käyttöliittymän työtehtäväpuun tietomalliin.

Käyttötapauksen tapahtumasekvenssikaavio on esitetty kuvassa 5.5. Käyttäjä painaa F5-painiketta päivittääkseen työtehtäväpuun tehtävät ja käyttötapaus alkaa. Painikkeen painallus käsitellään pääikkunassa, joka kutsuu *QubitApplication*-luokan *fetchTasks*-jäsenfunktiota, jonka toiminta on kuvattu alla olevassa esimerkissä:

```
auto finished = [this]() -> void { ... };
auto failed = [this](QString msg) -> void { ... };
mFetchTasksContext.reset(new FetchTasksContext(mBackend, mData, finished, failed));
(*mFetchTasksContext)();
```

Funktio luo *FetchTasks*-kontekstin ja käynnistää sen kutsumalla kontekstin funktio-operaattoria. Kontekstin parametreja ovat rooleihin sidottavat dataoliot ja lambda-funktiot, joita kutsutaan onnistuneen ja epäonnistuneen suorituksen jälkeen.



Kuva 5.5. *Työtehtävien haun eteneminen DCI-arkkitehtuurissa.*

Käyttötapaukseen osallistuvaa *Backend*-roolia pyydetään ensimmäisenä hakemaan työtehtävät. Rooli viittaa omaan dataolioonsa (*self*) ja pyytää sitä ja edelleen

taustajärjestelmäliitännäistä noutamaan työtehtävät. Kutsu on asynkroninen, ja F5-painikkeen aiheuttaman tapahtumaketjun suoritus päättyy. Käyttöliittymäsäikeen suoritus jatkuu, ja käyttöliittymä pysyy aktiivisena työtehtävien hakemisen aikana.

Jonkin ajan kuluttua taustajärjestelmäliitännäinen on saanut haettua työtehtävät ja kutsuu jokaiselle tehtävälle *Backend*-roolin takaisinkutsufunktiota. *Backend*-rooli viittaa kontekstin kautta käyttötapaukseen osallistuvaan toiseen rooliin, joka on työtehtävien paikallisesta tallennuksesta ja käyttöliittymän tietomallista vastaava *TaskStorage*-rooli. Roolin *addTask*-funktio lisää työtehtävän roolia näyttelevään *TaskStorage*-dataolioon, ja yksittäisen tehtävän lisäys päättyy.

Kun kaikki tehtävät on lisätty/päivitetty, kutsuu taustajärjestelmäliitännäinen haun päättymisen merkiksi *finished*-takaisinkutsufunktiota. *Backend*-rooli ilmoittaa kontekstille, että se on näytellyt oman osansa loppuun, ja kontekstin suoritus päättyy. Käyttötapaus on saatu onnistuneesti suoritettua. Epäonnistumisen tapauksessa konteksti kutsuisi käyttöliittymän *failed*-takaisinkutsufunktiota, joka näyttäisi käyttäjälle virheilmoituksen.

Osa kontekstiluokan sisällöstä on esitetty alla olevassa koodiesimerkissä:

```
template<typename BackendT, typename TaskStorageT, typename TaskPtrT, typename TaskDataT>
class FetchTasks: public ::Context
{
  public:
    typedef FetchTasks< BackendT, TaskStorageT, TaskPtrT, TaskDataT > ContextType;

    class Backend: public Role< BackendT, ContextType > { ... }
    class TaskStorage: public Role< TaskStorageT, ContextType > { ... }

    FetchTasks(BackendT backend, TaskStorageT storage, FinishCb finished, FailCb failed):
        backend(backend, *this),
        taskStorage(storage, *this),
        finished(finished),
        failed(failed)
    {
    }

    void operator()()
    {
      backend.fetchTasks();
    }

  private:
    Backend backend;
    TaskStorage taskStorage;
}
```

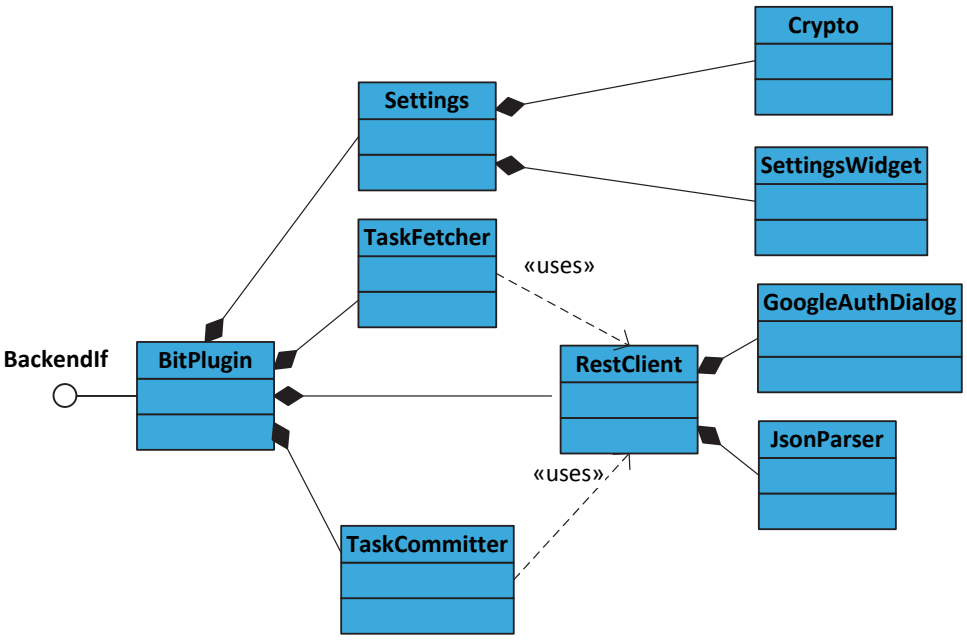
Kontekstiluokka sisältää *Backend*- ja *TaskStorage*-roolit. Kontekstin funktio-operaattori kutsuu *Backend*-roolin *fetchTasks*-jäsenfunktiota. Roolit on esimerkissä kirjoitettu kontekstin sisäluokkina, mutta ne voitaisiin toteuttaa myös omissa otsikkotiedostoissaan. Esimerkki havainnollistaa luokkamallien käyttöä. Yksityiskohtainen toteutus on kuitenkin jätetty esimerkin ulkopuolelle.

Käyttötapauksessa esitellyt roolit ovat geneerisiä C++:n luokkamalleja, ja niitä voidaan uudelleenkäyttää myös muissa käyttötapauksissa. Mitä suurempi järjestelmä, sitä enemmän käyttötapauksia ja rooleja voidaan uudelleenkäyttää järjestelmän sisällä. Qubit on ominaisuuksiltaan melko yksinkertainen sovellus, mutta jo pienessäkin sovelluksessa komponenttien uudelleenkäyttö on mahdollista. DCI:stä saadaan kuitenkin enemmän hyötyä, kun sovelluksen koko kasvaa.

5.3.3 Liitännäinen taustajärjestelmään

BitPlugin-liitännäinen hoitaa Qubit-sovelluksen kommunikoinnin BIT-palvelimen kanssa. Liitännäisen ja Qubit-sovelluksen välinen rajapinta on yleiskäyttöinen. Taustajärjestelmäliitännäistä vaihtamalla saadaan Qubit keskustelemaan toisen palvelimen tai paikallisen tietokannan kanssa. Toisaalta samaa BitPlugin-liitännäistä käyttäen voidaan tehdä sovellukselle Vincit Oy:n sisällä myös muita käyttöliittymiä – esimerkiksi komentorivikäyttöliittymä. Lisäksi liitännäisarkkitehtuuri mahdollistaa Qubit-sovelluksen käyttämisen myös muissa yrityksissä työajan mittaamiseen, sillä yritys voi toteuttaa ohjelmaan oman liitännäisensä.

BitPlugin-liitännäinen on toteutettu Qt-liitännäisenä Qt4-ohjelmistokehystä käyttäen. Sen luokkakaavio on esitetty kuvassa 5.6. Liitännäinen käyttää OpenSSL-kirjastoa HTTPS-liikennöintiin ja käyttäjän tunnusten salaamiseen. OpenID-tunnistautumisen yhteydessä saadut tunnukset tallennetaan paikalliseen tietokantaan, ja samoja tunnuksia käytetään niin kauan kuin ne ovat voimassa. REST-rajapintakyselyiden parametrien ja paluuarvojen JSON-muodon jäsentämiseen käytetään QJson-kirjastoa.

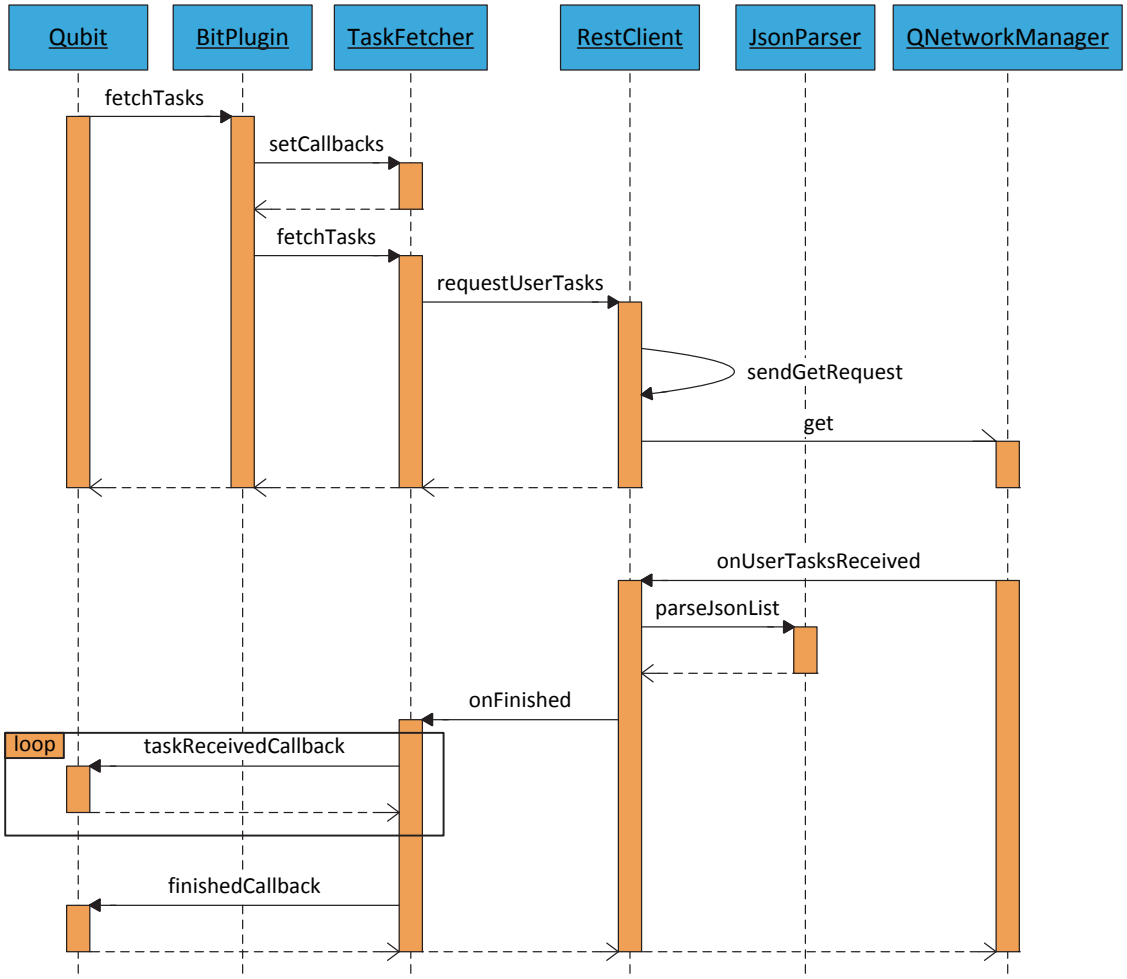


Kuva 5.6. *BitPlugin-liitännäisen luokkakaavio.*

BitPlugin-luokka toteuttaa *BackendIf*-rajapinnan. *Settings*-luokka toteuttaa liitännäisen pysyvien asetusten tallentamisen ja lukemisen. Näitä ovat liitännäisen käyttämä palvelimen osoite ja viimeksi käytetyt REST-rajapinnan käyttäjätunnukset. Tunnukset salataan tallentamisen yhteydessä *Crypto*-luokkaa käyttäen. Liitännäisen asetuksia voi muokata käyttöliittymästä *SettingsWidget*-asetusnäkymää käyttäen. Varsinainen palvelinkommunikointi tapahtuu *RestClient*-luokassa. Luokka käyttää JSON-muodon jäsentämiseen ja tuottamiseen *JsonParser*-luokkaa, joka käyttää sisäisesti *QJson*-kirjastoa.

Ensimmäisellä kirjautumiskerralla vaadittu Google OpenID -autentikointi tehdään *GoogleAuthDialog*-dialogia käyttäen. Dialogi näyttää Googlen kirjautumissivun ja jäsentää vastauksesta onnistuneen autentikoinnin jälkeen REST-rajapinnassa käytettävät tunnukset. *TaskFetcher*-luokka kysyy työtehtävät ja käsittelee saapuneen vastauksen. *TaskCommitter* puolestaan tekee työtuntikirjauksen ja tarkistaa, että tuntikirjaus palvelimelle onnistui.

Kuvassa 5.7 on tapahtumasekvenssikaavio tehtävien haun etenemisestä.



Kuva 5.7. *Työtehtävien haun eteneminen liitännäisen sisällä.*

Qubit-sovellus pyytää työtehtäviä *BackendIf*-rajapinnan funktiolla *fetchTasks*. Funktion parametreina saadaan takaisinkutsufunktiot yksittäisen työtehtävän lisäämiselle ja koko kyselyn valmistumiselle ja epäonnistumiselle. Takaisinkutsufunktiot asetetaan *TaskFetcher*-oliolle ja kutsutaan sen *fetchTasks*-funktiota. *TaskFetcher* pyytää *RestClient*-oliolta työtehtäväkyselyn lähettämistä *requestUserTasks*-funktiolla, joka lähettää HTTP GET -pyynnön BIT-palvelimelle.

HTTP-vastauksen saavuttua sen sisältämä JSON-viesti jäsennetään ja palautetaan *TaskFetcher*-luokalle työtehtävät sisältävä dataolio. *TaskFetcher* käsittelee työtehtävät ja kutsuu alussa määritettyä *taskReceived*-takaisinkutsufunktiota erikseen jokaiselle työtehtävälle. Kun kaikki työtehtävät on käsitelty, kutsutaan vielä alussa määritettyä *finished*-takaisinkutsufunktiota sen merkiksi, että koko työtehtävien haku on valmis.

5.3.4 Testaus

Sekä liitännäiselle että Qubit-sovellukselle on automaattiset testitapaukset. Testaukseen käytetään *cmake*-käännöstyökalun *ctest*-työkalua. Testit käyttävät Qt:n *QTest*-testauskehystä.

Liitännäisestä testattiin sen jokaisen julkisen jäsenfunktion toiminta. Qubit-sovellusta puolestaan testattiin DCI-arkkitehtuurin mukaisesti käyttötapauskohtaisesti. DCI-arkkitehtuurin käyttö helpotti käyttötapausten testaamista, koska arkkitehtuurissa on jokaiselle käyttötapaukselle oma *Context*-olio, joka voidaan suorittaa.

Kun tietokone on ollut käyttämättä jonkin aikaa ja työntekijä palaa koneelle, esittää ohjelma *Missä olit* -dialogin. Työntekijä voi valita dialogista, mitä tehtävää oli tekemässä, oliko hän tauolla tai jatkaa keskeytyksettä edellistä käynnissä ollutta tehtävää. Tämän käyttötapauksen automaattinen testaaminen on DCI-arkkitehtuurin ansiosta erityisen yksinkertaista ja olisi perinteisellä MVC-arkkitehtuurilla luultavasti vaatinut useita testitapauksia.

6. ARVIOINTI

Tässä luvussa arvioidaan toteutetun Qubit-sovelluksen ja projektin onnistumista. Aluksi arvioidaan projektin onnistumista kokonaisuutena sekä sovelluksen käytettävyysparannuksien ja eri käyttöjärjestelmäversioiden toteutus. Seuraavaksi pohditaan, miten DCI-arkkitehtuuri soveltui kyseiseen projektiin, ja onko sitä kannattavaa soveltaa tulevaisuudessa vastaavissa tai isommissa projekteissa. Lisäksi arvioidaan toteutuksessa käytettyjen C++11-standardin ominaisuuksien tuomaa hyötyä ja käytettävyyttä. Lopuksi esitetään Qubit-sovellukseen liittyen esiin nousseet jatkokehitysajatukset.

6.1 Sovelluksen ja projektin onnistuminen

Qubit-sovelluksen kehitys aloitettiin syksyllä 2012. Sovellusta kehitettiin työajan ulkopuolella, ja sen valmistumiselle ei asetettu tiukkaa aikataulua. Projekti saatiin kuitenkin vietyä alusta loppuun tasaisella aikataululla, ja se valmistui kohtuullisessa ajassa. BIT-palvelimen ensimmäinen versio otettiin käyttöön helmikuusta 2013 alkaen, jonka jälkeen Jiraan ei enää kirjattu tunteja. Samaan aikaan myös Qubitista julkaistiin rajoitetulle käyttäjäryhmälle ensimmäinen testiversio. Versio 1.0 otettiin käyttöön maaliskuun aikana.

Uuden DCI-arkkitehtuurin opettelu vei oletettua enemmän aikaa, ja Qubitia kehitettiin iteroiden lopulliseen muotoonsa. DCI:stä johtuen sovelluksessa on myös perinteistä ratkaisua enemmän koodirivejä.

Qubitin käytettävyyttä parannettiin onnistuneesti aikaisempaan Vtt-sovellukseen nähden. Ensimmäistä testiversiota koekäyttäneet työntekijät antoivat diplomityössä esitettyjen parannusten lisäksi myös uusia ideoita ja palautetta käytettävyyteen liittyen. Lopputulos koettiin testiryhmän mielestä käytettäväksi.

Qt osoittautui odotetusti hyväksi ohjelmistokehykseksi alustariippumattoman sovelluksen kehittämiseen. Alustariippumaton *cmake*-käännöstyökalu mahdollisti helpon kääntämisen käyttöjärjestelmästä riippumatta, ja myös eri alustojen asennuspakettien teko onnistui pienellä työmäärällä. Itselleni uuden *cmake*-työkalun opettelu vei kuitenkin paljon aikaa diplomityöprojektin alkuvaiheessa. Kokonaisuutena projekti oli onnistunut.

6.2 DCI-arkkitehtuurin soveltuvuus

DCI soveltui hyvin käytettäväksi MVC-mallia noudattavan Qt-ohjelmistokehyksen kanssa. DCI vaati verrattain pienikokoisessa sovelluksessa perinteiseen MVC-ratkaisuun nähden enemmän koodirivejä. Arkkitehtuurin rakentaminen siis vaatii enemmän työtä ja koodirivejä, mutta DCI muuttuu tehokkaaksi sovelluksen koon kasvaessa. Tehokkuus kasvaa sen mukaan, mitä enemmän sovelluksella on käyttötapauksia.

Jo pienelläkin sovelluksen koolla DCI dokumentoi kooditasolla selkeästi monimutkaisten käyttötapausten toiminnan, joten arkkitehtuuri parantaa koodin luettavuutta ja koko järjestelmän toiminnan hahmottamista. Toisaalta C++-kielen luokkamallien (*template*) käyttö hankaloittaa jonkin verran syntaksia. Tämä on kuitenkin ohjelmointikielen eikä DCI-arkkitehtuurin ominaisuus.

DCI-ajattelumallin mukainen *Mikä järjestelmä on - mitä järjestelmä tekee* -erottelu vaikuttaa toimivalta varsinkin suuremmissa järjestelmissä. Uuden ominaisuuden lisäys vaatii käytännössä uuden *context*-luokan kirjoittamisen. Luokkaan liittyy yksi tai useampi rooli, joita voidaan uudelleenkäyttää olemassa olevista käyttötapauksista. Varsinainen data vaatii muutoksia DCI-ideologian mukaisesti suuressa järjestelmässä hyvin harvoin. Järjestelmän toiminta on altis muutoksille ja lisäyksille, mutta data on pysyvää.

Oppimiskynnys uuden arkkitehtuurin omaksumiseen on kuitenkin korkea. Lean Architecture -kirjan [2] esimerkit ovat yksinkertaistettuja, ja niiden soveltaminen oikean järjestelmän toteuttamiseen on vaikeaa. Uuden ajattelumallin opettelu on kuitenkin aina haastavaa, kuten aikanaan esimerkiksi funktionaalisesta ohjelmoinnista olio-ohjelmointiin siirryttäessä.

Qubit on melko pieni sovellus, ja DCI ei tässä kokoluokassa tuonut ylivoimaista etua, koska sovellukseen ei ole odotettavissa suurta määrää uusia käyttötapauksia. Sovelluksen testaus oli kuitenkin helpompaa DCI-arkkitehtuurin ansiosta, koska testauksessa testataan käyttötapauksia, ja niille on arkkitehtuurin ansiosta suora vastine ohjelmakoodissa. Käyttötapauksien toiminta on myös suoraviivaisesti nähtävissä kontekstiluokkien koodista, eli koodi dokumentoi hyvin järjestelmän toiminnan.

Kuten edellä todettiin, DCI-arkkitehtuurin soveltuvuus riippuu järjestelmän luonteesta. Jos järjestelmä on käyttäjäkeskeinen tai sisältää paljon toiminnallisuutta ja erilaisia käyttötapauksia, on arkkitehtuurin käytöstä merkittävää hyötyä koko järjestelmän hallinnassa. Kuitenkin perinteinen oliolähestymistapa on DCI:tä parempi, jos järjestelmä on melko yksinkertainen, sisältää vain muutaman oikean käyttötapauksen ja koostuu atomisista operaatioista. Myös näiden lähestymistapojen yhdistäminen on mahdollista: yksinkertaiset operaatiot voi toteuttaa DCI-arkkitehtuurin dataolioihin, mutta rooleja ja kontekseja kannattaa silti käyttää käyttötapauksien toteuttamiseen, jolloin niiden toiminta on selkeästi dokumentoitu. [2, s. 287]

DCI:tä voi kokemuksen perusteella suositella käytettäväksi myös muissa projekteissa – erityisesti ketterissä asiakasprojekteissa, joissa ensimmäinen ohjelmaversio on suppea demo, ja sitä on tarkoitus laajentaa, kun vaatimukset ja käyttötapaukset selkenevät. Arkkitehtuurin käyttö on suositeltavaa myös, jos järjestelmä on laaja ja sisältää paljon käyttötapauksia. DCI on askel kohti järjestelmätason toiminnan kuvausta ja korkeampaa suunnittelun abstraktiotasoa.

6.3 C++11:n ominaisuuksien hyödyntäminen

Esitellyistä C++11:n ominaisuuksista hyödynnettiin koodissa erityisesti luokkamalleja (*template*) DCI-arkkitehtuurin toteuttamiseen. Syntaksia helpottavista uudistuksista käytettiin uutta *for_ each*-silmukkarakennetta lähes poikkeuksetta tietorakenteiden läpikäymiseen ja *auto*-avainsanaa korvaamaan DCI-toteutuksen aiheuttamia pitkiä *template*-tyyppimäärittelyjä. Nämä ominaisuudet helpottivat huomattavasti koodin kirjoitusta ja luettavuutta.

Lisäksi taustajärjestelmäliitännäisen kanssa kommunikoinnissa käytettiin takaisinkutsufunktioiden toteutukseen lambdafunktioita. Lambdafunktiot toimivat hyvin tässä käyttötarkoituksessa, sillä varsinaisen funktiokutsun yhteyteen saatiin koodissa dokumentoivasti näkyviin, mitä tapahtuu onnistuneen ja epäonnistuneen suorituksen jälkeen. Lambdafunktioita käytettiin onnistuneesti myös paikallisten muutaman rivin apufunktioiden toteutuksessa ja yksikkötestauksessa.

6.4 Kehitysideat

Qubitin ensimmäinen versio sisältää ohjelman minimitoiminnallisuuden. Mukana on lähes samat ominaisuudet kuin aikaisemmassa Vtt-sovelluksessa – käytettävyyttä on kuitenkin onnistuneesti parannettu. Ajatuksena on pitää Qubit-sovelluksessa vain päivittäin tarvittavat ominaisuudet ja pitää sen käyttö mahdollisimman yksinkertaisena ja nopeana. Harvemmin tarvittavat uudet ominaisuudet lisätään BIT-palvelimen selainkäyttöliittymään.

Tällä hetkellä Qubit-sovelluksessa voidaan lähettää työtunnit palvelimelle, minkä jälkeen ne häviävät Qubitista. Kehitysajatuksena on mahdollistaa työtuntikirjausten synkronointi molempiin suuntiin, jolloin palvelimella tehdyt muutokset näkyisivät myös Qubitissa, ja Qubitista olisi mahdollista muokata lähetettyjä työtunteja.

Tekstimuotoisesta työtuntilokista on vaikea hahmottaa taukojen ja työtehtävien kestoja tai tehtävien päällekkäisyyttä. Käytettävyysparannuksena työtuntilokiin voitaisiinkin tehdä myös viikkonäkymä, jossa tuntikirjaukset näkyisivät kalenterimaisesti jokainen päivä omassa sarakkeessaan.

Jos työntekijä on mukana monessa eri projektissa, projektipuurakenteesta tulee helposti isokokoinen. Oikean tehtävän löytäminen ja valitseminen hiirellä on hanka-

laa. Kehitysajatuksena voitaisiin erikseen näyttää useimmin käytetyt työtehtävät. Puurakenteen yläpuolelle voitaisiin myös laittaa tekstikenttä, jolla puunäkymässä näkyviä projekteja ja tehtäviä voisi suodattaa kirjoittamalla osan projektin tai tehtävän nimestä.

BitPlugin-liitännäinen mahdollistaa myös kokonaan vaihtoehtoisen käyttöliittymän toteutuksen samaa liitännäistä käyttäen. Jatkokehitysajatuksena on suunniteltu vaihtoehdoksi myös komentorivikäyttöliittymää, jossa komennot muistuttaisivat yrityksen käyttämästä *git*-versionhallintajärjestelmästä tuttuja komentoja.

7. YHTEENVETO

Kokonaisuutena uusi BIT-projektinhallintajärjestelmä helpotti yrityksen projektinhallintaa ja integroi aikaisemmin hajanaiset palvelut yhdessä toimivaksi kokonaisuudeksi. Uusi järjestelmä on vanhaa käytettävämpi ja tukee vanhaa paremmin projektinhallintaa sekä työntekijöiden työmäärän ja projektikiinnitysten ennakoivaa suunnittelua.

Diplomityössä toteutettu Qubit-tuntikirjaussovellus saatiin Qt-ohjelmistokehyksen avulla onnistuneesti toteutettua eri käyttöjärjestelmille. Eri käyttöjärjestelmäversiolle jouduttiin tekemään vain vähän käyttöjärjestelmäkohtaista koodia liittyen koneen käyttämättömänä olon tunnistamiseen ja asennuspaketteihin. Qubitissa onnistuttiin myös parantamaan merkittävästi käytettävyyttä vanhaan Vtt-sovellukseen nähden.

Qt-ohjelmistokehys soveltui joustavuutensa ansiosta hyvin käytettäväksi työssä tutkitun DCI-arkkitehtuurin kanssa. DCI-arkkitehtuuri osoittautui hyväksi toteutustavaksi, ja se soveltuu käytettäväksi käytännön projekteissa. Arkkitehtuuri erottaa selkeästi erilleen sovelluksen tietosisällön ja olemuksen sovelluksen toiminnasta.

Monimutkaistenkin käyttötapausten toiminta on arkkitehtuurin ansiosta selkeästi nähtävissä sovelluksen koodissa. DCI-arkkitehtuurin mukainen koodi dokumentoi järjestelmän toiminnan perinteistä olioarkkitehtuuria paremmin, sillä käyttötapaukset ovat järjestelmän rakennuspalikoita yksittäisten luokkien sijaan. DCI:n esittelemä roolipohjainen ohjelmointi eli olioiden roolien erottaminen itse olioista mallintaa myös paremmin todellista maailmaa ja käyttäjien ajatusmalleja.

DCI mahdollistaa myös aiempaa paremmin järjestelmän ketterän kehityksen, jossa järjestelmän toimintaa laajennetaan käyttötapaus kerrallaan. Uuden käyttötapauksen lisääminen tai käyttämättömän toiminnon poisto aiheuttaa vain vähän muutoksia sovelluksen koodiin.

DCI on toteutettu C++-ohjelmointikielellä käyttäen luokkamalleja. Vaikka DCI rakenteena parantaakin koodin luettavuutta ja dokumentoi koko järjestelmätason toiminnan, on luokkamallien syntaksia C++-kielen käytön vuoksi aluksi hankala lukea. Syntaksin vaikeus johtuu kuitenkin ohjelmointikielestä eikä itse arkkitehtuurista.

C++11-standardi sisälsi käyttökelpoisia uudistuksia, joita hyödynnettiin Qubit-sovelluksen toteutuksessa. Koodin luettavuus parani ja syntaksi helpottui *auto-*

avainsanan, uuden *for_ each*-silmukkarakenteen ja lambdafunktioden käytöllä.

Kokemuksen perusteella DCI-arkkitehtuurin käyttöä voi suositella etenkin paljon käyttötapauksia sisältävissä sovelluksissa. Oppimiskynnys on aluksi korkea arkkitehtuurin erilaisesta ajatusmallista johtuen. Syvemmän perehtymisen jälkeen vaikuttaa kuitenkin siltä, että arkkitehtuurissa on tehty oikeita valintoja. Arkkitehtuurista on merkittävää hyötyä varsinkin sovelluksen koon kasvaessa.

Pienistä selkeistä toiminnoista koostuviin ja vain muutaman käyttötapauksen sisältäviin sovelluksiin arkkitehtuuri ei kuitenkaan sovellu. Rajatapauksissa erilaiset välimuodot perinteisen oliosuunnittelun ja DCI-arkkitehtuurin välillä ovat järkevin vaihtoehto.

DCI tukee ketterää iteratiivista projektikehitystä ja *Lean*-mallia, jotka molemmat ovat viime aikoina yleistyneet ohjelmistokehityksessä. Arkkitehtuuri on selkeästi askel kohti koko järjestelmätason toiminnan kuvausta ja käyttäjäkeskeistä suunnittelua. Arkkitehtuurin käyttö yleistynee tulevaisuudessa.

LÄHTEET

[1] Reenskaug, T. *Home page of Trygve M. H. Reenskaug.* 2010. [Viitattu 15.03.2013] Saatavilla: http://heim.ifi.uio.no/trygver/.

[2] Coplien, J., Bjørnvik, G. *Lean Architecture: for Agile Software Development.* 2010. Great Britain: John Wiley & Sons Ltd. 357 sivua.

[3] Coplien, J. The DCI Architecture: Supporting the Agile Agenda. *Øredev Conference*, 2009. [Viitattu 15.03.2013] Saatavilla: http://vimeo.com/8235574.

[4] Öberg, R. DCI in practice. *Øredev Conference*, 2009. [Viitattu 17.03.2013] Saatavilla: http://vimeo.com/8235651.

[5] Blanchette, J., Summerfield, M. *C++ GUI Programming with Qt4. Second Edition.* 2008. United States: Prentice Hall. 718 pages.

[6] Digia. *Digia ostaa koko Qt-kehitysympäristön Nokialta.* 2012. [Viitattu 14.09.2012] Saatavilla: http://www.digia.com/fi/Digia/Yritys/Uutiset/Digia-ostaa-koko-Qt-kehitysympariston-Nokialta/.

[7] Qt Developer Network. *Qt 4.8 Documentation.* 2012. [Viitattu 14.09.2012] Saatavilla: http://qt-project.org/doc/qt-4.8/.

[8] Digia. *Qt Reference Documentation - All Modules.* 2012. [Viitattu 22.09.2012] Saatavilla: http://doc.qt.digia.com/4.8-snapshot/modules.html.

[9] Trolltech. *Qt 4.3 class chart - a selection of classes provided by Qt 4.3.* 2007. [Viitattu 05.10.2012] Saatavilla: http://doc.qt.digia.com/extras/qt43-class-chart.pdf.

[10] Digia. *Qt Reference Documentation - Signals & Slots.* 2012. [Viitattu 05.10.2012] Saatavilla: http://doc.qt.digia.com/qt/signalsandslots.html.

[11] Digia. *Threading Basics.* 2012. [Viitattu 21.09.2012] Saatavilla: http://doc.qt.digia.com/4.7/thread-basics.html.

[12] Digia. *Qt Reference Documentation - Using the Meta-Object Compiler.* 2012. [Viitattu 20.09.2012] Saatavilla: http://doc.qt.digia.com/4.7/moc.html.

[13] Koskimies, K., Mikkonen, T. *Ohjelmistoarkkitehtuurit.* 2005. Jyväskylä: Talentum Media Oy. 250 sivua.

[14] Amaya, S. *History of C++.* [Viitattu 21.09.2012] Saatavilla: http://www.cplusplus.com/info/history/.

[15] Boost Community. *Boost C++ Libraries.* [Viitattu 30.10.2012] Saatavilla: http://www.boost.org/.

[16] Stroustrup, B. *C++11 Style – A Touch of Class.* 2012. [Viitattu 21.09.2012] Saatavilla: http://ecn.channel9.msdn.com/events/GoingNative12/GN12Cpp11Style.pdf.

[17] Rintala, M., Jokinen, J. *Olioiden ohjelmointi C++:lla.* 2005. Jyväskylä: Talentum Media Oy. 377 sivua.

[18] Korpela, H. *C++11 - New features - Variadic templates.* 2012. [Viitattu 08.10.2012] Saatavilla: http://www.cplusplus.com/articles/EhvU7k9E/.

[19] *Lambda-functions (Since C++11).* [Viitattu 22.09.2012] Saatavilla: http://en.cppreference.com/w/cpp/language/lambda.

[20] Kumar, A., Sutton, A., Stroustrup, B. Rejuvenating C++ Programs through Demacrofication. *28th IEEE International Conference on Software Maintenance*, 2012. [Viitattu 14.09.2012] Saatavilla: http://www.stroustrup.com/icsm-2012-demacro.pdf.

[21] *Storage duration specifiers.* 2012. [Viitattu 08.10.2012] Saatavilla: http://en.cppreference.com/w/cpp/language/storage_duration.

[22] *Auto Specifier.* 2012. [Viitattu 08.10.2012] Saatavilla: http://en.cppreference.com/w/cpp/language/auto.

[23] *JavaScript Object Notation.* 2012. [Viitattu 11.10.2012] Saatavilla: http://www.json.org/.

[24] Fielding, R. *Architectural Styles and the Design of Network-based Software Architectures.* 2000. Doctoral dissertation, University of California. [Viitattu 24.10.2012] Saatavilla: http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

[25] Elkstein, M. *Learn REST: A Tutorial.* [Viitattu 24.10.2012] Saatavilla: http://rest.elkstein.org/.

[26] OpenID Community. *OpenID Authentication 2.0.* 2007. [Viitattu 02.11.2012] Saatavilla: http://openid.net/specs/openid-authentication-2_0.html.

[27] OpenID Community. *OpenID Foundation Website.* [Viitattu 02.11.2012] Saatavilla: http://openid.net/l.

[28] Google Developers. *Federated Login for Google Account Users.* [Viitattu 02.11.2012] Saatavilla: https://developers.google.com/accounts/docs/OpenID.

[29] *JIRA Documentation.* 2012. [Viitattu 23.11.2012] Saatavilla: https://confluence.atlassian.com/display/JIRA/JIRA+Documentation.

[30] Oracle Corporation. *MySQL - The world's most popular open source database.* 2013. [Viitattu 15.03.2013] Saatavilla: http://www.mysql.com/.

[31] Spring Source Community. *What Is Spring.* 2013. [Viitattu 15.03.2013] Saatavilla: http://www.springsource.org/.

[32] The PostgreSQL Global Development Group. *PostgreSQL.* 2013. [Viitattu 15.03.2013] Saatavilla: http://www.postgresql.org/.