Tampereen ammattikorkeakoulu
Tietotekniikan koulutusohjelma
Ohjelmistotekniikka
Tom Vähä-Salo

Opinnäytetyö

Qt Creator -plugin kehittäminen

Työn ohjaaja Lehtori Tony Torp
Työn tilaaja Mowhi Oy, ohjaajana insinööri Arno Saine
Tampere 03/2010

Tampereen ammattikorkeakoulu
Tietotekniikan koulutusohjelma, Ohjelmistotekniikka

Tekijä	Tom Vähä-Salo
Työn nimi	Qt Creator -plugin kehittäminen
Sivumäärä	38 sivua + 15 liitesivua
Valmistumisaika	05/2010
Työn ohjaaja	Lehtori Tony Torp
Työn tilaaja	Mowhi Oy, ohjaajana insinööri Arno Saine

TIIVISTELMÄ

Tässä työssä käsitellään Qt Creator -plugin ohjelmointia. Työn tarkoituksena on perehtyä plugin-ohjelmoinnin lisäksi pakollisiin esivalmisteluihin sekä muihin kehitystyössä huomioon otettaviin seikkoihin.

Työn sisällön perusteella aiheeseen perehtymättömän lukijan toivotaan saavan peruskäsityksen aiheesta sekä perusvalmiudet alkaa omatoimisesti laajentaa osaamistaan. Lukijalla oletetaan olevan perustietämys Qt-sovelluskehityksestä, sekä hyvä perusosaaminen C++-ohjelmoinnista.

Tämän työn yhteydessä esitetyt ohjelmointiesimerkit on varmistettu toimiviksi Qt Creator versiolla 2.0 alpha1. Esimerkit toimivat osittain myös vanhempien versioiden kanssa, mutta uudempia versioita käytettäessä on syytä varautua yhteensopivuusongelmiin.

Lopuksi tässä työssä esitellään tämän dokumentin ohella toteutettua esimerkkipluginia.

Avainsanat Qt-ohjelmointi, Qt Creator, plugin-kehitys

Tampere University of Applied Sciences
Computer Science, Software Engineering

Writer	Tom Vähä-Salo
Thesis	Qt Creator Plugin Development
Pages	38 pages + 15 appendices
Graduation time	May 2010
Thesis Supervisor	Tony Torp
Co-operating Company	Mowhi Corporation, supervisor Arno Saine

ABSTRACT

This thesis covers the basic knowledge that is needed when a programmer is developing a plugin for Qt Creator. This thesis also explains the preliminary work to be done before it is possible to begin programming and which programming aspects should be taken into account.

Programmers with basic knowledge about Qt application development and with good C++ skills should get a quite good introduction to the subject. This thesis also gives a basic readiness to the reader to independently continue expanding the reader's knowledge about the subject.

Examples in this thesis have been tested with Qt Creator version 2.0 alpha1. Some of the examples may work with older versions of Qt Creator, but there are no guarantees that they will work with any newer versions.

Some of the examples are based on the example plugin, which was made for this document. The plugin is described briefly at the end of this thesis and the plugin’s source code is available in the appendix section.

Keywords Qt programming, Qt Creator, Plugin development

Tampereen ammattikorkeakoulu
Tietotekniikan koulutusohjelma
Ohjelmistotekniikka

Sisällysluettelo

Termien ja lyhenteiden luettelo

Qt	Kehitysympäristö alustariippumattomien ohjelmien ja graafisten käyttöliittymien kehitykseen. Ympäristö on pääsääntöisesti tarkoitettu C++-ohjelmointikielelle.
Qt Creator	Avoimeen lähdekoodiin perustuva kehitystyökalu Qt-ohjelmistokehitykseen ja normaaliin C/C++-kehitykseen.
Plugin	Ohjelmaan asennettava lisäkomponentti, joka laajentaa esimerkiksi ominaisuuksia ja toimintoja.
IDE	Integrated Development Environment. Ohjelmointiympäristö, joka tarjoaa kaikki ohjelmistokehityksessä tarvittavat työkalut, kuten kääntäjän ja ohjelmakoodieditorin.
SDK	Software Development Kit. Kokoelma kehitystyökaluja ja rajapintoja, joiden avulla tehdään ohjelmistokehitystä tietylle alustalle tai tiettyyn tarkoitukseen.
Visual Studio	Microsoftin kehittämä IDE, joka tukee useita eri ohjelmointikieliä.
Jaettu kirjasto	Ajonaikaisesti ladattava ohjelmamoduuli, jonka tarjoamia palveluja tai dataa voidaan hyödyntää samanaikaisesti useassa eri ohjelmassa. Jaetut kirjastot ovat Windows-käyttöjärjestelmässä .dll-päätteisiä, UNIX- tai Linux-järjestelmässä .so-päätteisiä ja Mac-järjestelmässä .dylib-päätteisiä.
XML	Extensible Markup Language. Kuvauskieli, jonka avulla eri järjestelmien tai ohjelmien välillä voidaan siirtää tietoa jokaisen osapuolen ymmärtämässä muodossa.
Pri-tiedosto	Qt:n projektitiedoston käyttämä otsikkotiedosto, jonka avulla projektiin voidaan tuoda uusia ominaisuuksia.

1 Johdanto

Qt Creator on Nokian kehittämä IDE (Integrated Development Environment), jonka ensimmäinen virallinen versio julkaistiin maaliskuussa 2009. Alusta lähtien ohjelmaa on yleisesti pidetty eräänä parhaimmista Qt-sovelluskehitykseen soveltuvista työkaluista. Avoimen lähdekoodinsa ja Qt-pohjaisuutensa ansiosta ohjelmaa voidaan hyödyntää usealla eri käyttöjärjestelmällä. Viimeisimpien versioiden myötä Qt Creatoriin on tullut mahdollisuus kehittää Qt-sovelluksia myös Symbian-alustalle.

Lähes kaikki Qt Creatorin toiminnot on toteutettu erilaisilla plugineilla ja tässä työssä onkin tarkoitus perehtyä plugin-kehitykseen, sekä kehityksessä huomioon otettaviin seikkoihin. Työ voidaan jakaa neljään toisiinsa sidoksissa olevaan osaan, joiden kautta lukija voi omaksua Qt Creatorin yleisrakenteen sekä plugin-kehityksessä yleisimmin tarvittavat tekniikat.

Toisessa luvussa tehdään yleiskatsaus Qt Creatoriin ja sen arkkitehtuuriin. Lisäksi käydään läpi pluginien käyttöönottaminen sekä pluginin tietojen selvittäminen käytön aikana.

Kolmannessa luvussa kerrotaan esivalmistelut, jotka on tehtävä ennen kuin plugin-kehitys on mahdollista aloittaa. Tässä luvussa käsitellään muun muassa tarvittavien kehityskirjastojen käyttöönottamista sekä tärkeiden projektitiedostojen käyttöä.

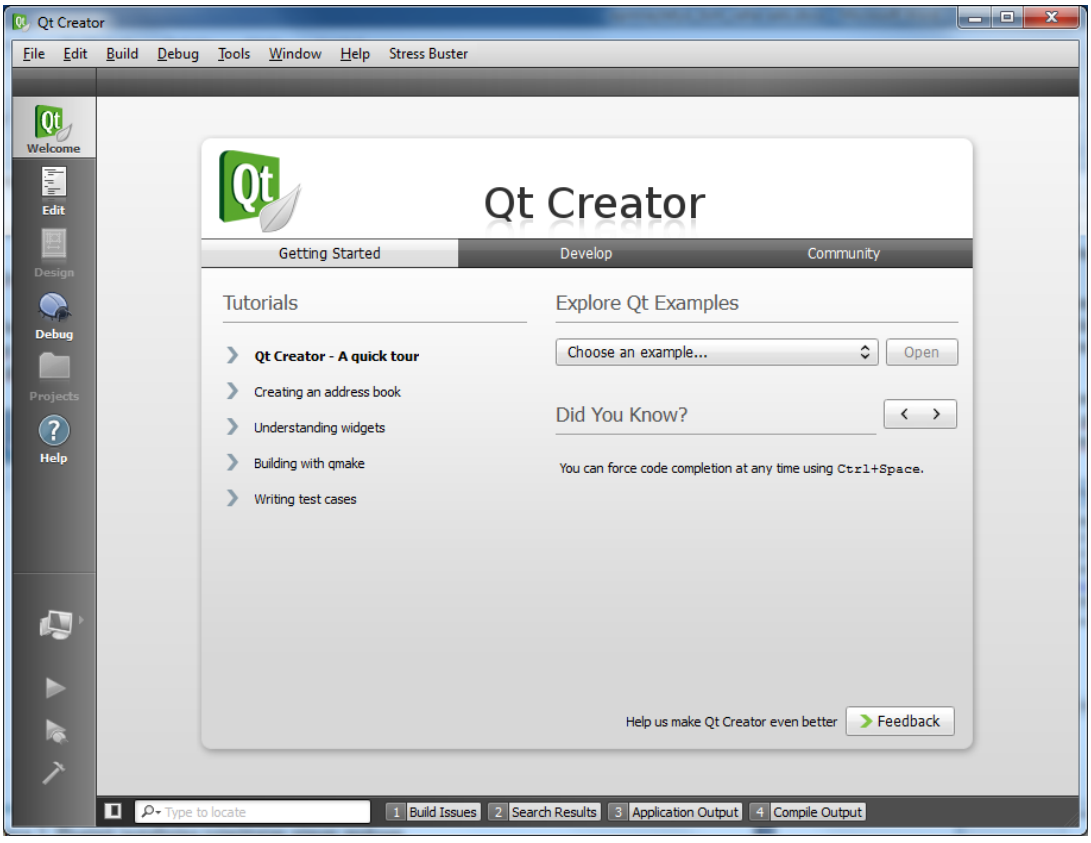
Työn neljännessä luvussa tutustutaan muutamiin valmiisiin kehitysrajapintoihin, sekä käsitellään plugin-ohjelmoinnissa huomioon otettavia seikkoja. Lisäksi tutustutaan pluginin yleiseen rakenteeseen muun muassa moninaisten esimerkkien avulla.

Viidennessä luvussa esitellään tämän dokumentin ohella toteutettua esimerkkipluginia.

2 Qt Creator

Qt Creator on Nokian kehittämä ja avoimeen lähdekoodiin perustuva IDE eli ohjelmointiympäristö. Vaikka Qt Creatorin pääsääntöinen tarkoitus on toimia työkaluna Qt-kehityksessä, voidaan sitä käyttää myös tavallisen C/C++ -ohjelman kehitykseen. Lisäksi ohjelman varsinainen toiminnallisuus toteutetaan yksinomaan erilaisilla plugineilla, joten laajennusmahdollisuuksia on lähes rajattomasti. Tästä syystä ohjelmaan olisi varsin helppo toteuttaa esimerkiksi Java-kielen tuki. Hiljattain julkaistu versio 2.0 alpha1 tarjoaa entistä paremman Symbian-tuen sekä uusina ominaisuuksina ohjelmistokehitystuen Maemo-alustalle ja QML-tuen käyttöliittymäkehitykseen. /3/ /5/

Ohjelman päänäkymä on esitetty kuvassa 1.



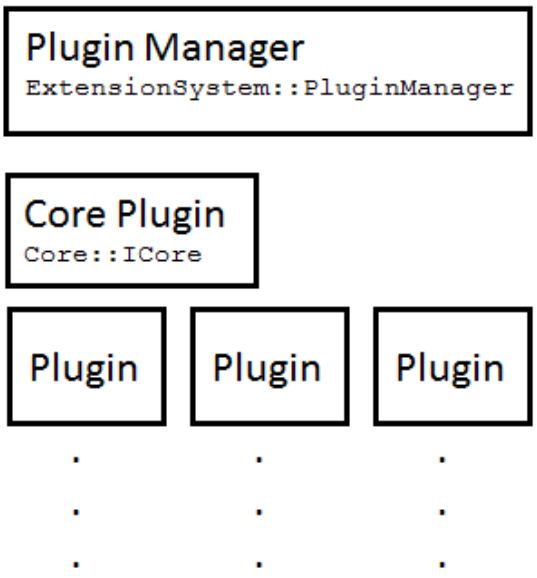
Kuva 1: Qt Creator 2.0 alpha1 -version päänäkymä

2.1 Arkkitehtuuri

Kuten jo aikaisemmin mainittiin, Qt Creatorin toiminnallisuus nojaa täysin erilaisiin plugineihin. Varsinaisen Qt Creator -ohjelman tehtävänä on toimia ainoastaan plugin-

lataajana (eng. plugin loader) ja tätä tehtävää hoitaa Plugin Manager -moduuli. Plugin Managerin toiminnallisuudesta on kerrottu luvussa 2.1.1. /1/

Kuvassa 2 on esitetty yksinkertaistettu kaavio Qt Creatorin arkkitehtuurista. Kaaviosta voidaan myös nähdä Plugin Managerin sijoittuminen plugineihin nähden.



Kuva 2: Qt Creatorin arkkitehtuuri

2.1.1 Plugin Manager

Plugin Manager on Qt Creatorin ExtensionSystem -nimiavaruuteen kuuluva moduuli, jonka tehtävä on hallinnoida ohjelman lisäosia eli plugineita. Plugin Managerin päätehtävät ovat seuraavat:

- Pluginien ja niiden tilojen hallinta
- Yleisen oliosäiliön (Common Object Pool) käsittely.

Plugin Managerilla on sisäinen oliosäiliö, jonne pluginit voivat säilöä instansseja toteuttamiinsa rajapintoihin, joita muut pluginit saattaisivat haluta hyödyntää. Ainoa vaatimus instanssin tallentamiseen on, että tallennettava olio perii QObject-luokan ominaisuudet. Jos plugin toteuttaa jonkin valmisrajapinnan, on tällainen instanssi aina lisättävä oliosäiliöön, jotta toteutettu rajapinta pystytään asettamaan käyttöön. Tällaisia rajapintoja ovat muun muassa `Core::IEditor` ja `Core::IOutputPane`. /2/

2.1.2 Core-Plugin

Vaikka Qt Creatorin toiminnallisuus on pääsääntöisesti toteutettu plugineilla, ei minkään pluginin puuttuminen yleensä estä ohjelman käynnistymistä. Poikkeuksen tähän tekee Core-Plugin. Core-plugin toteuttaa Qt Creatorin matalimman tason toiminnallisuuden ja sen puuttuminen estää ohjelman käynnistämisen. Core-pluginin sijoittuminen ohjelman arkkitehtuurissa voidaan nähdä kuvasta 2.

Lähes poikkeuksetta myös kaikki muut pluginit riippuvat Core-pluginin olemassaolosta. Tämä johtuu siitä, että erilaisia käyttörajapintoja ohjelman hallintaan on mahdollista saada ainoastaan Core-pluginin kautta. Esimerkki tällaisesta rajapinnasta on käyttöliittymävalikoiden lisäys, poisto ja hallinta. /1/ /2/

2.2 Pluginin käyttöönotto

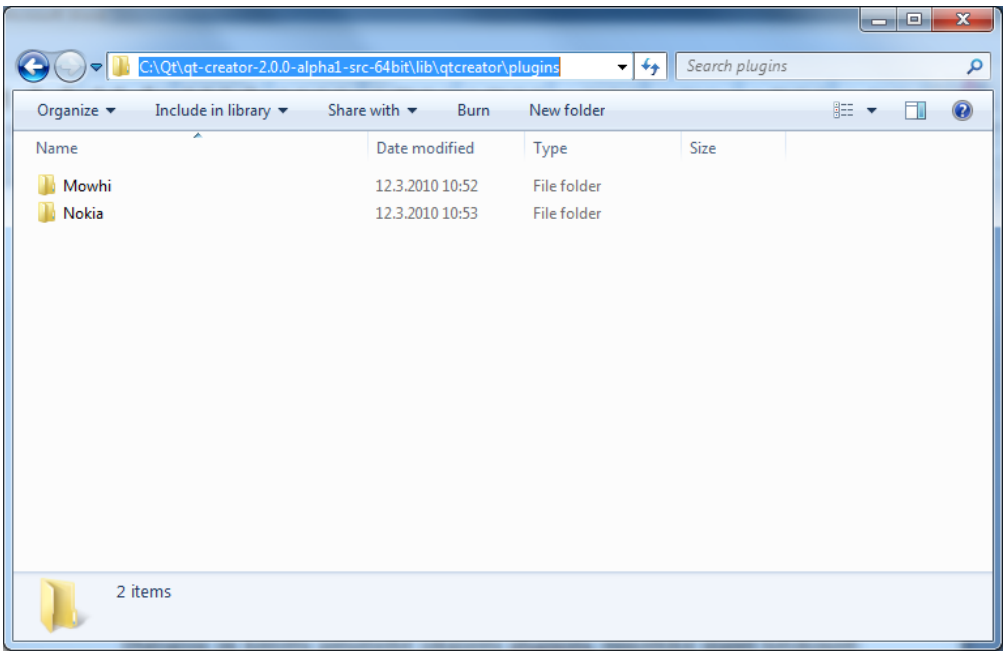
Uuden pluginin asennuksessa on yleensä huolehdittava kahden tiedoston sijoittamisesta oikeaan paikkaan. Ensimmäinen tiedosto on varsinainen plugin, joka on päällisin puolin samanlainen kuin mikä tahansa jaettu kirjasto. Toinen tärkeä tiedosto on .pluginspec-päätteinen tiedosto, jossa on kerrottu käyttöönotettavan pluginin perustiedot. Jälkimmäisestä tiedostosta kerrotaan tarkemmin myöhemmin tässä dokumentissa.

Qt Creatorilla on kaksi oletuskansiota, joista se yrittää käynnistyessään ladata pluginit käyttöönsä. Oletuskansiot ovat Qt Creatorin juurikansiosta lukien `lib/qtcreator/plugins` ja `PlugIns`. Oletuskansiot saattavat kuitenkin vaihdella asennuksen ja käyttöjärjestelmän mukaan. Plugin Manager tarjoaa myös metodit pluginkansioiden kysymiseen ja asettamiseen. Esimerkissä 1 on näytetty, kuinka Plugin Managerilta voidaan kysyä nykyiset pluginien oletuskansiot.

Esimerkki 1: Plugin-kansioiden kysyminen ja tulostus konsoliin

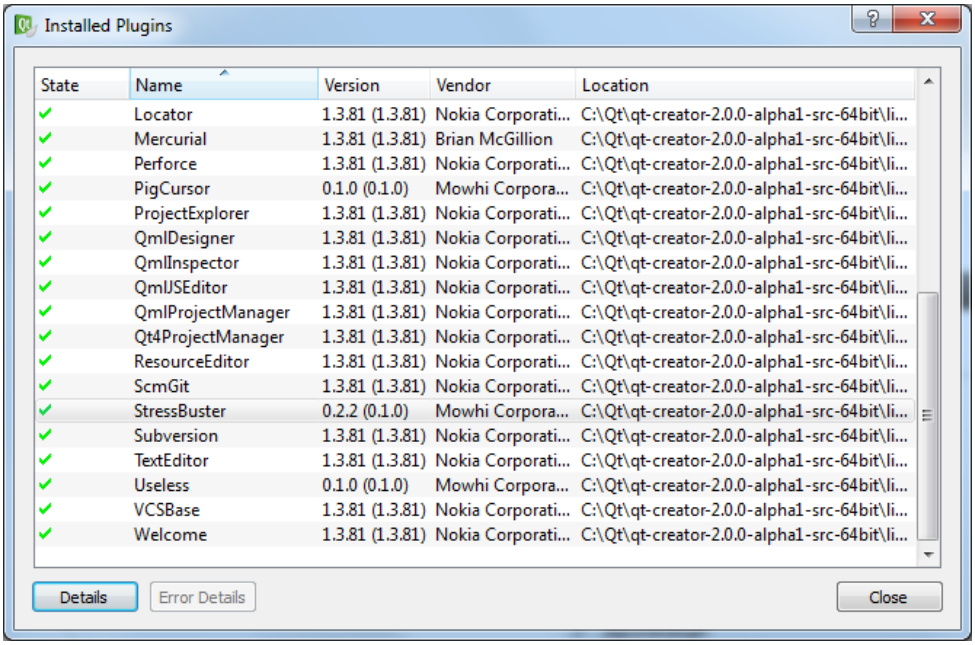
```
qDebug() << ExtensionSystem::PluginManager::instance()->pluginPaths();
```

Uutta pluginia ei välttämättä tarvitse sijoittaa suoraan oletuskansioon, vaan Qt Creator osaa etsiä saatavilla olevia plugineja myös alikansioista. Kuvassa 3 pluginit on eritelty toimittajan nimen mukaan.



Kuva 3: Pluginit jaoteltuina toimittajan nimen mukaan

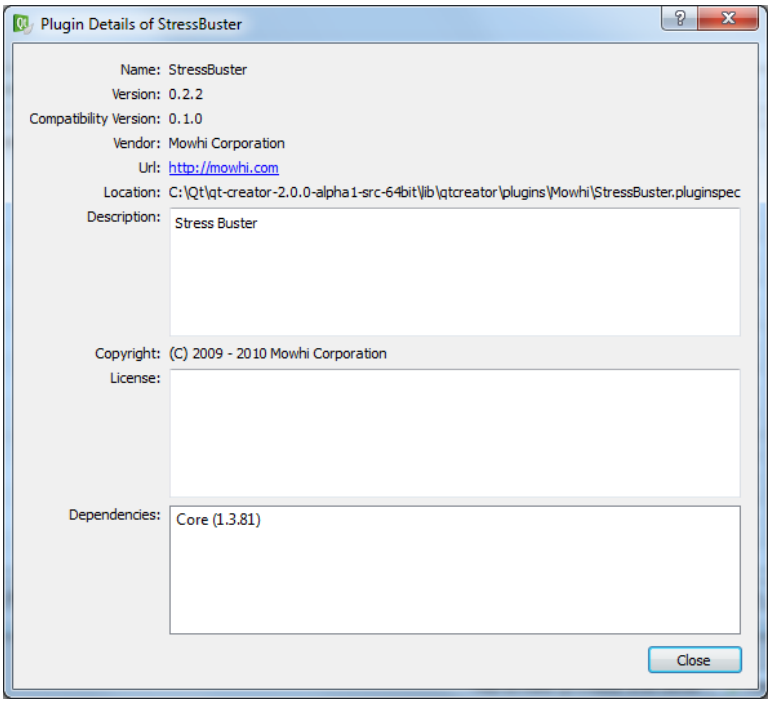
Asennetut pluginit ja niiden statukset saa selvitettyä käytön aikana Help-valikon About Plugins -kohdasta, jonka jälkeen aukeaa kuvan 4 mukainen dialogi.



Kuva 4: Asennetut pluginit

Dialogissa on kerrottu perustiedot jokaisesta pluginista, esimerkiksi sijainti tietokoneella ja käytössä oleva versio. Details-painiketta painamalla pääsee tarkastelemaan pluginin tarkempia tietoja, kuten kuvassa 5 on näytetty.

State-palsta kertoo pluginin latauksen onnistumisen. Jos pluginin lataus epäonnistui, näkyy sen status punaisena ja mahdollisen virheviestin saa näkyviin painamalla Error Details -painiketta.



Kuva 5: Ladatun pluginin tarkemmat tiedot

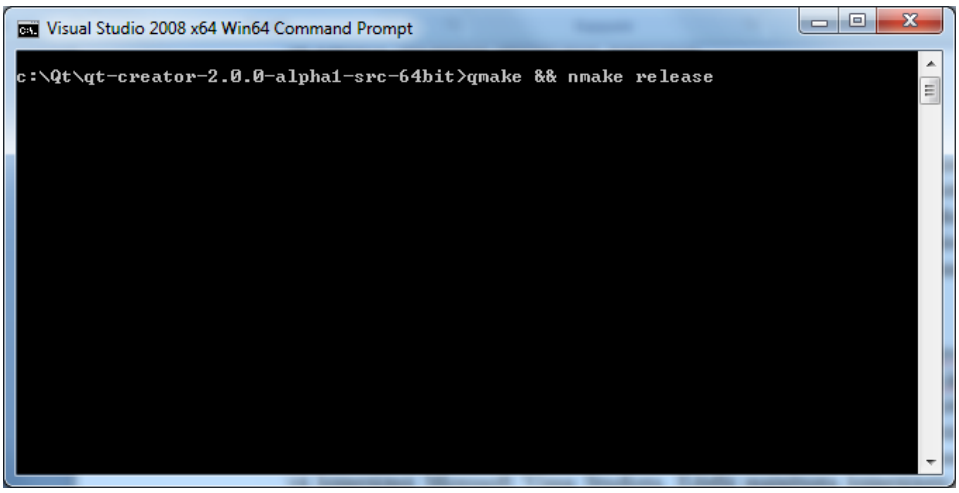
3 Esivalmistelut

Qt Creator -plugin kehityksen aloittaminen ei ole yhtä suoraviivaista kuin normaalin Qt-sovelluksen kehittäminen. Tämä johtuu siitä, että Qt Creator -pluginin kehitykseen ei ole julkaistu mitään julkista kehitysrajapintaa. Tästä syystä, kehittäjän on ensimmäiseksi käännettävä Qt Creator tarvittavien kehityskirjastojen saamiseksi. Käännösoperaatiosta on kerrottu tarkemmin seuraavassa alaluvussa.

3.1 Qt Creatorin kääntäminen

Kuten aikaisemmin mainittiin, Nokia ei ole toistaiseksi julkaissut julkista kehitysrajapintaa Qt Creator -plugin kehitykseen. Tämä johtuu pääosin siitä, että kehitysrajapintojen odotetaan yhä muuttuvan, joten mitään pysyvää rajapintaa ei ole järkevä lyödä lukkoon. Plugin-kehittäjän on myös hyvä muistaa, että uuden ohjelmaversion ilmestyessä, vanhalle ohjelmistoversiolle kehitettyjen pluginien binääriyhteensopivuus saattaa rikkoutua.

Qt Creatorin kääntäminen on varsin suoraviivaista, jos normaalin Qt-sovelluksen kääntäminen on ennestään tuttua. Kääntäminen vaatii, että kehittäjällä on Qt-kehityskirjastot asennettuna (2.0-sarjan Qt Creator vaatii 4.7-sarjan Qt-kirjastot) sekä jokin Qt:n tukema kääntäjä. Windows-käyttöjärjestelmällä on kuitenkin muistettava, että jos oman pluginin haluaa binääriyhteensopivaksi Nokian jakaman valmispaketin kanssa, on käytettävä kääntäjänä Microsoft Visua Studiota. Edellä mainitusta kääntäjästä käyvät versiot 2005 ja 2008, mutta 2005-versiolla käännetyn pluginin mukana pitäisi muistaa toimittaa kääntäjän C- ja C++-ajonaikaiskirjastot. Kuvassa 6 on näytetty kääntämiseen vaadittavat komennot Visual Studiolla. /1/ /2/



Kuva 6: Qt Creatorin kääntäminen Visual Studiolla

Kääntäminen kannattaa tehdä Visual Studion omalla komentorivillä, kuten kuvassa 6 on tehty. Tällöin kääntäjän omia tiedostopolkuja ei tarvitse itse asettaa. Konetehoista riippuen ohjelman kääntämiseen on syytä varata aikaa 10 - 30 minuuttia. Käännöksen jälkeen on suositeltavaa vielä varmistaa ohjelman toimivuus käynnistämällä ohjelma `bin\qtcreator.exe`. /4/

3.2 Projektin luonti

Kun Qt Creator on käännetty ja sen toimivuus on testattu, voidaan vähitellen aloittaa oman pluginin kehittäminen. Ennen varsinaisen ohjelmoinnin aloittamista on kuitenkin syytä tarkastella projektin luontiin liittyviä seikkoja sekä projektin muiden tiedostojen merkitystä.

Projektin luontiin ei ole olemassa mitään valmista pohjaa, koska mitään julkista rajapintaa plugin-kehitykseenkään ei ole olemassa. Näin ollen kaikki oman projektin projektitiedostot on suurimmaksi osaksi luotava itse. Mallia omiin tiedostoihin voi ottaa valmiista plugineista sekä Qt Creatorin lähdekoodien mukana tulevasta HelloWorld -esimerkkipluginista.

Oma projekti on hyvä luoda Qt Creatorin juurikansiosta lukien src\plugins-kansioon, esimerkiksi src\plugins\testplugin, jossa testplugin on oman projektin nimi. Projektin luonti tähän hierarkiaan takaa sen, että projektiin tuotavien kirjastojen ja otsikkotiedostojen polut ovat automaattisesti oikein. Lisäksi käännösvaiheessa plugin tulee kopioiduksi suoraan oikeaan paikkaan. Seuraavissa alaluvuissa käsitellään projektiin kuuluvia erityistiedostoja. /1/

3.2.1 Projektitiedosto

Projektille on luotava oma projektitiedosto, kuten myös normaalissa Qt-sovelluskehityksessä on tapana. Tässä tapauksessa projektitiedostoon tulee muutama erityispiirre, joihin on syytä kiinnittää huomiota. Esimerkissä 2 on esitetty erään pluginin projektitiedosto, jonka jälkeen on kerrottu tiedoston tärkeimmät kohdat.

Esimerkki 2: Projektitiedosto

```
TEMPLATE = lib
TARGET = TestPlugin
PROVIDER = Mowhi
include(testplugin_dependencies.pri)
HEADERS += testplugin.h \
    testview.h \
    testsound.h \
    testplugin_export.h
SOURCES += testplugin.cpp \
    testview.cpp \
    testsound.cpp
DEFINES += TESTPLUGIN_LIBRARY
LIBS += -L$$IDE_PLUGIN_PATH/Nokia
RESOURCES += testresources.qrc
OTHER_FILES += TestPlugin.pluginspec
```

Tärkeimmät kohdat:

- `TEMPLATE = lib`
- `PROVIDER = Mowhi`
- `include(testplugin_dependencies.pri)`
- `LIBS += -L$$IDE_PLUGIN_PATH/Nokia`

Listan ensimmäinen kohta kertoo projektin tyypin. Pluginien toiminnallisuus on hyvin lähellä normaalien ohjelmointikirjastojen toiminnallisuutta, joten myös projektin tyyppi on sama.

Listan toinen kohta kertoo kyseisen pluginin toimittajan. Jos toimittajaa ei määritä, oletetaan pluginin toimittajan olevan Nokia. Tämä vaikuttaa siihen, että minkä nimiseen alikansioon plugin siirretään kääntämisen jälkeen. Tässä tapauksessa plugin siirretään automaattisesti Mowhi-nimiseen kansioon.

Projektitiedostoon voidaan lisätä omia otsikkotiedostoja, joissa on määritelty projektiin vaikuttavia asioita. Tässä tapauksessa listan kolmannessa kohdassa projektiin lisätään otsikkotiedostot, jossa on kerrottu projektin riippuvuudet. Pri-tiedostoista kerrotaan lisää myöhemmin tässä työssä.

Viimeinen kohta on riippuvainen listan toisesta kohdasta. Tässä tapauksessa kirjasto-polkuihin on lisättävä kansio, josta löytyvät Nokian toimittamat pluginit, muun muassa Core-plugin. Jos toimittajaa ei määritä, ei myöskään listan viimeistä kohtaa tarvitse määrittää. /2/

3.2.2 PluginSpec-tiedosto

Kuten aikaisemmin mainittiin, jokaisella pluginilla pitää olla oma PluginSpec-määrittelytiedosto. Tiedosto on XML-muotoinen (Extensible Markup Language) ja siinä määritellään muun muassa pluginin riippuvuudet, jotka Qt Creator lukee käytön aikana. Tietoja voi tarkastella käytön aikana ohjelman About Plugins -valikon kautta. Esimerkissä 3 on esitetty erään pluginin määrittelytiedosto.

Esimerkki 3: Pluginin määrittelytiedosto

```
<plugin name="TestPlugin" version="0.1.0" compatVersion="0.1.0">
    <vendor>Mowhi Corporation</vendor>
    <copyright>(C) 2010 Mowhi Corporation</copyright>
    <license>
    </license>
    <description>Test plugin.</description>
    <url>http://mowhi.com</url>
    <dependencyList>
        <dependency name="Core" version="1.3.81"/>
    </dependencyList>
</plugin>
```

Tiedoston tärkeimmät kohdat ovat ensimmäisellä rivillä ja dependencyList-luettelossa.

Ensimmäisellä rivillä on kerrottu pluginin nimi, jota käytetään tunnistukseen käytön aikana. Muut pluginit voivat asettaa riippuvuuksia tähän pluginiin käyttämällä määrittelytiedostossa kerrottua nimeä.

Nimen jälkeen kerrotaan pluginin versionumero, joka on muotoa xx.xx.xx_xx. Jos jotakin numerokohtaa ei määritetä, korvataan se automaattisesti nollalla. Esimerkiksi 0.1.0 ja 0.1.0_0 ovat sama asia.

Versionumeroinneista `compatVersion` on kaikkein tärkein. Se kertoo yhteensopivuuden aiempien versioiden kanssa ja numero ei saa olla suurempi kuin ilmoitettu versionumero. Muut pluginit käyttävät tätä numeroa ilmoittaessaan omia riippuvuuksiaan, joten

virheellinen numerointi aiheuttaa tästä pluginista riippuvien pluginien latauksen epäonnistumisen.

Määrittelytiedoston toinen tärkeä kohta on `dependencyList` eli pluginin riippuvuuslista. Tähän kohtaan listataan kaikki muut pluginit, joiden olemassaolosta kyseinen plugin riippuu. Qt Creator tutkii jokaisen pluginin riippuvuuslistan läpi, minkä jälkeen pluginit ladataan siinä järjestyksessä, kuin ne toisistaan riippuvat. Jos riippuvuuslistassa ilmoitettua pluginia ei löydy tai vaadittu versionumero ei ole oikealla välillä, pluginin lataus epäonnistuu.

Riippuvuuslistassa voi olla plugineita teoriassa nollasta äärettömään, mutta käytännössä listassa on aina vähintään Core-plugin, kuten esimerkissä 3. Riippuvuuden nimi ja `compatVersion`:n numero nähdään tarvittavan pluginin määrittelytiedostosta, kuten esimerkissä 4 on näytetty. /2/

Esimerkki 4: Core-pluginin nimi ja versionumeroinnit omassa määrittelytiedostossaan

```
<plugin name="Core" version="1.3.81" compatVersion="1.3.81">
...
```

3.2.3 Pri-tiedostot

Pri-tiedostoilla voidaan tuoda projektiin valmiita ominaisuuksia toisista plugineista tai kirjastoista, jolloin näitä ominaisuuksia ei tarvitse itse kirjoittaa omaan projektitiedostoon. Omien pri-tiedostojen luonti ei ole pakollista, mutta erittäin suositeltavaa etenkin, jos plugin tarjoaa jotakin rajapintaa toisille plugineille. Tiedostoja luodaan yleensä kaksi, joista ensimmäisessä kerrotaan pluginin omat riippuvuudet ja toisessa tiedostossa määritellään pluginin jatkokäyttöön tarvittavat asetukset. Esimerkkipluginin pri-tiedostot on esitetty esimerkeissä 5 ja 6.

Esimerkki 5: testplugin_dependencies.pri

```
include(../../qtcreatorplugin.pri)
include(../../libs/extensionsystem/extensionsystem.pri)
include(../../plugins/coreplugin/coreplugin.pri)
```

Esimerkki 6: testplugin.pri

```
include(testplugin_dependencies.pri)
LIBS *= -l$$qtLibraryTarget(TestPlugin)
```

Esimerkin 5 tiedostossa on listattu pluginin riippuvuudet. Tässä tapauksessa pluginin toiminnallisuus riippuu ainoastaan pakollisista riippuvuuksista, jotka ovat samat jokaisella pluginilla (pois lukien Core-plugin). Jos projektissa ei käytä omia pri-tiedostoja, on riippuvuustiedoston rivit lisättävä suoraan projektin projektitiedostoon.

Esimerkissä 6 oleva tiedosto määrittelee asetukset pluginin jatkokäyttöä varten. Jokin toinen plugin voi lisätä tämän tiedoston omiin riippuvuuksiinsa ja tarvittavat käännösasetukset on siltä osin määritelty. Tiedostossa pitää määrittää tarvittavat riippuvuudet sekä linkitysvaiheessa tarvittavan kirjaston nimi. /2/ /4/

3.3 Projektin kääntäminen

Plugin-projektin kääntäminen tapahtuu, kuten minkä tahansa Qt-projektin kääntäminen. Toisin sanoen, kääntäminen voidaan tehdä esimerkiksi joko suoraan Qt Creatorista tai manuaalisesti komentoriviltä.

Kääntäminen komentoriviltä on jossain määrin suositeltavampaa, koska komentorivikäännöksessä käännösvaiheisiin ja järjestykseen voi vaikuttaa itse. Kääntäminen komentoriviltä voidaan tehdä seuraavasti:

1. Siirrytään projektikansioon.

2. Suoritetaan QMake-komento.

3. Suoritetaan Make-komento (Visual Studiolla nmake). Windowsilla Make-komennolle on vielä syytä antaa release-parametri, jotta kääntäjä kääntää ainoastaan lopullista versiota.

Jos kääntäminen onnistui virheittä, plugin kopioitiin automaattisesti oikeaan plugin-kansioon. Tämän jälkeen on syytä käynnistää Qt Creator ja varmistaa pluginin toimivuus.

4 Plugin-ohjelmointi

Moninaisten alkuvalmisteluiden jälkeen, päästään vihdoin aloittamaan varsinainen plugin-ohjelmointi. Tässä luvussa tutustutaan joihinkin valmiisiin rajapintoihin, sekä kerrotaan plugin-ohjelmoinnissa huomioon otettavista seikoista. Lisäksi näytetään muun muassa, kuinka lisätään oma valikkotoiminto sekä laajempana esimerkkinä oman asetusvälilehden luonti.

Tässä luvussa ei käsitellä kaikkia Qt Creatorin tarjoamia rajapintoja, koska niiden käsittely tämän työn rajoissa ei ole järkevää. Valmiit rajapinnat on kuitenkin helppo tunnistaa lähdekoodien seasta, koska luokkien nimet alkavat aina I-kirjaimella (interface). Tällaisia luokkia ovat muun muassa `ExtensionSystem::IPlugin`, `Core::IWizard` ja `Core::IOptionsPage`.

Kuten aikaisemmin tässä työssä mainittiin, plugin-kehitykseen ei ole saatavilla julkista dokumentaatiota. Tämä tarkoittaa sitä, että plugin-kehittäjän on syytä varautua lukemaan paljon lähdekoodia valmiista plugineista ja Qt Creatorin lisäkirjastoista. Yleisimmät rajapinnat ja toiminnallisuudet on kuitenkin yleensä kommentoitu hyvin, joten täysin oman koodinlukutaidon varassa ei tarvitse olla. Hyvä esimerkki mainiosta kommentoinnista on Plugin Manager, jonka lähdekoodit löytyvät Qt Creatorin juurikansiosta lukien `src\libs\extensionsystem`. /2/

4.1 Pääluokka

Jokaisen pluginin pitää toteuttaa tietynlainen pääluokka, jonka kautta Qt Creator hallinnoi muun muassa pluginin latausta. Luokka toteutetaan perimällä ExtensionSystem -nimiavaruudesta löytyvä IPlugin-luokka (`ExtensionSystem::IPlugin`). Luokka on abstrakti kantaluokka, kuten kaikki Qt Creatorin tarjoamat rajapinnat. Esimerkeissä 7 ja 8 on listattu yksinkertaisin mahdollinen pääluokka.

Esimerkki 7: Pääluokan otsikkotiedosto

```
#include <extensionsystem/iplugin.h>

class UselessPlugin : public ExtensionSystem::IPlugin
{
    Q_OBJECT

public:
    UselessPlugin();
    ~UselessPlugin();

    bool initialize(const QStringList &arguments,
                    QString *error_message);

    void extensionsInitialized();

    void shutdown();
};
```

Esimerkki 8: Pääluokan toteutustiedosto

```
UselessPlugin::UselessPlugin()
{
}

UselessPlugin::~UselessPlugin()
{
}

bool UselessPlugin::initialize(const QStringList &arguments,
                               QString *error_message)
{
    Q_UNUSED(arguments)
    Q_UNUSED(error_message)

    return true;
}

void UselessPlugin::extensionsInitialized()
{
}

void UselessPlugin::shutdown()
{
}

Q_EXPORT_PLUGIN(UselessPlugin)
```

Esimerkkien 7 ja 8 ohjelmalistauksilla saadaan toteutettua yksinkertaisin mahdollinen plugin, mutta kyseisellä pluginilla ei vielä ole mitään varsinaista toiminnallisuutta. Ohjelmakoodilistauksista voidaan kuitenkin nähdä, että pluginin perusrakenne on varsin yksinkertainen. Seuraavaksi esitellään yksityiskohtaisesti toteutustiedoston metodit ja toiminnallisuudet.

```
UselessPlugin::UselessPlugin()
{
}
```

Luokan rakentajan tehtävä on ainoastaan alustaa luokan jäsenet, sekä luoda luokan toiminnallisuudesta riippumattomat moduulit, esimerkiksi ladata resurssit muistista tai kiintolevyltä.

```
UselessPlugin::~UselessPlugin()
{
}
```

Purkajan tehtävä on sama, kuin normaalissa olio-ohjelmoinnissa. Olioiden poisto yleisestä oliosäiliöstä pitää myös tehdä viimeistään purkajassa.

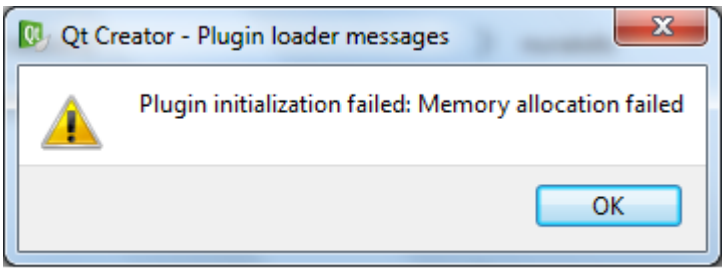
```
bool UselessPlugin::initialize(const QStringList &arguments,
                               QString *error_message)
{
    return true;
}
```

Tämä on koko luokan tärkein metodi ja pakollinen toteuttaa. Qt Creator kutsuu tätä metodia, kun se saanut ladattua kaikki pluginin riippuvuudet. Metodissa tulisi alustaa kaikki pluginin varsinaiset toiminnot ja ulospäin näkyvät komponentit, kuten käyttöliittymä.

Metodin parametreista ensimmäinen on merkkijonolista, jossa on listattuna pluginin mahdolliset argumentit. Toinen parametri on osoitin merkkijono-olioon, jonne voi tallentaa käyttäjälle näytettävän virheilmoituksen, kuten esimerkissä 9 ja kuvassa 7 on näytetty. Metodin paluuarvo kertoo alustuksien onnistumisen.

Esimerkki 9: Virheviestin palautus alustuksen epäonnistuessa

```
bool UselessPlugin::initialize(const QStringList &arguments,
                               QString *error_message)
{
    Q_UNUSED(arguments)

    m_myObject = new MyClass;

    if (!m_myObject) {
        *error_message = “Memory allocation failed”;
        return false;
    }

    return true;
}
```



Kuva 7: Initialize-metodin palauttama virheilmoitus

```
void UselessPlugin::extensionsInitialized()
{
}
```

Tämä on toinen pakollisista metoditoteutuksista. Tätä metodia kutsutaan vasta, kun initialize-metodia on kutsuttu, jolloin myös kaikki tätä pluginia tarvitsevat pluginit on jo alustettu. Jos tämä plugin tarjoaisi jotakin uutta rajapintaa, tässä metodissa olisi hyvä tutkia mahdollisia rajapintatoteutuksia yleisestä oliosäiliöstä.

```
void UselessPlugin::shutdown()
{
}
```

Tätä metodia kutsutaan, kun plugininia ollaan poistamassa ja ennen luokan purkajan kutsumista. Metodia ei ole pakko toteuttaa, jos sille ei ole tarvetta.

```
Q_EXPORT_PLUGIN(UselessPlugin)
```

Tällä makrolla tuodaan plugin muulle maailmalle näkyväksi. Makro on esitelty Qt:n sisäisessä toteutuksessa, joten käyttöä varten pitää lisätä otsikkotiedosto QtCore/QtPlugin. /1/ /2/

4.2 Oliosäiliön käyttö

Qt Creatorin oliosäiliö on käytännöllinen tapa jakaa toteutettuja rajapintoja ja muita hyödyllisiä toimintoja muiden pluginien kesken. Oliosäiliötä hoitaa Plugin Manager -luokka ja myös säiliön pääasiallinen käyttö tapahtuu kyseisen luokan instanssin kautta. Jokainen plugin-luokka kuitenkin perii muutaman hyödyllisen metodin, joilla voidaan suorittaa oliosäiliöön lisäys- ja poistotoimenpiteitä.

Seuraavissa alaluvuissa on käsitelty oliosäiliön käyttöön liittyviä toimintoja.

4.2.1 Instanssien hakeminen oliosäiliöstä

Plugin Manager -luokalla on kaksi metodia (`getObject` ja `getObjects`), joilla voidaan suoraan pyytää oliosäiliöstä joko yksittäistä instanssia tai kaikkia tietyntyyppisiä instansseja. Metodit käyttävät hyväkseen C++:n template-toiminnallisuutta, joten niille pitää ainoastaan kertoa halutun olion tyyppi. Näiden kahden metodien paluuarvot eroavat siinä määrin, että `getObject` palauttaa suoraan osoittimen haluttuun olioon, mutta `getObjects` palauttaa QList -tyyppisen listan, johon on listattu kaikki osoittimet pyydetynlaisiin olioihin. Esimerkissä 10 on näytetty kummankin metodin käyttö.

Esimerkki 10: Olioinstanssien pyytäminen oliosäiliöstä

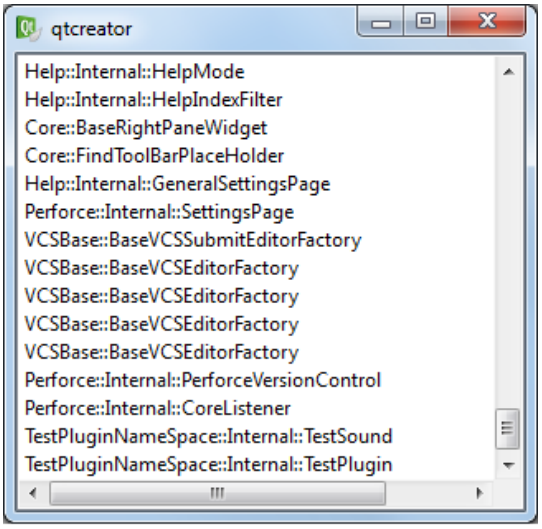
```
// Pyydetään Plugin Manager -instanssi
ExtensionSystem::PluginManager *pm;
pm = ExtensionSystem::PluginManager::instance();

// Haetaan instanssi tietyntyyppiseen olioon
TextEditor::FontSettingsPage *fsp;
fsp = pm->getObject<TextEditor::FontSettingsPage>();

// Haetaan kaikkien tietyntyyppisten olioiden instanssit
QList<Core::IOutputPane*> objects;
objects = pm->getObjects<Core::IOutputPane>();
```

Edellisillä esimerkeillä on mahdollista hakea vain tietyntyyppisiä olioita, mutta Plugin Manager tarjoaa myös kolmannen metodin, jolla on mahdollista hakea kaikkien mahdollisten olioiden instanssit. Tämän metodin nimi on `allObjects` ja se palauttaa QList-olion, johon on listattu osoittimia QObject-olioihin. Nämä osoittimet osoittavat oliosäiliöön tallennettuihin olioihin ja ennen käyttöä ne on tyyppimuunnettava haluttuun muotoon. Esimerkissä 11 ja kuvassa 8 on näytetty, kuinka tallennettujen olioiden nimet voidaan näyttää yksinkertaisessa käyttöliittymäkomponentissa.

Esimerkki 11: Kaikkien olioiden hakeminen oliosäiliöstä ja niiden nimien näyttäminen

```
// Pyydetään Plugin Manager -instanssi
ExtensionSystem::PluginManager *pm;
pm = ExtensionSystem::PluginManager::instance();

// Haetaan kaikki olio-osoittimet oliosäiliöstä
QList<QObject*> objects = pm->allObjects();

// Luodaan käyttöliittymäkomponentti
QListWidget *wid = new QListWidget;

// Kopioidaan olioiden luokkien nimet käyttöliittymäkomponenttiin
foreach (QObject *obj, objects)
    wid->addItem(obj->metaObject()->className());
```

```
// Näytetään käyttöliittymäkomponentti
wid->show();

...

// Tuhotaan käyttöliittymäkomponentti
delete wid;
```



Kuva 8: Esimerkissä 11 luotu käyttöliittymäkomponentti sisältöineen

Olio-osoittimien hakeminen oliosäiliöstä on varsin yksinkertaista, mutta on syytä muistaa, että osoittimen oikeellisuus on aina syytä tarkastaa ennen käyttöä. Jos pyydetynlaista osoitinta ei löydy oliosäiliöstä tai kutsun käsittelyssä tapahtuu virhe, lopputulos on, että käytetyn metodin paluuarvo on nolla. Jokaisen ohjelmoijan pitäisi tietää, että nollaosoittimen käyttö johtaa pahimmillaan ohjelman kaatumiseen.

Erityisen tarkkana nollaosoittimen varalta pitää olla `getObject` -metodia käytettäessä. Kaksi muuta tässä luvussa esiteltyä metodia palauttavat epäonnistuessaan ainoastaan tyhjän listan, joten niiden käyttö on turvallista.

4.2.2 Olioiden tallennus ja poisto

Olioiden lisäämiseen Plugin Manager tarjoaa `addObject` -metodin, joka ottaa parametrinaan osoittimen lisättävään olioon. Olion pitää olla olemassa koko sen ajan, kun olio on lisättynä oliosäiliöön. Lisäksi olion pitää periä QObject-luokan ominaisuudet jossakin periytymishierarkian vaiheessa.

Plugin Manager ei suorita minkäänlaista muistinhallintaa lisättyjen olioiden suhteen. Tästä syystä olioiden lisääjä on itse vastuussa olioiden tuhoamisesta ja olion poistosta oliosäiliöstä ennen tuhoamista. Olion poisto oliosäiliöstä tapahtuu `removeObject` -metodilla, joka ottaa parametrinaan osoittimen poistettavaan olioon.

Esimerkissä 12 on näytetty olion lisäys oliosäiliöön ja olion poisto oliosäiliöstä ohjelman suorituksen loppuessa.

Esimerkki 12: Olion lisäys ja poisto

```
// myclass.h
// Luokan pitää periytyä QObject-luokasta
class MyClass : public QObject
{
    // Q_OBJECT-makro pitää olla tyyppimuunnosten onnistumiseksi
    Q_OBJECT

public:
    MyClass(QObject *parent = 0); // rakentaja
    ~MyClass(); // purkaja
};

// myclass.cpp
// Oliosäiliöön lisäys voidaan tehdä esimerkiksi rakentajassa
void MyClass::MyClass(QObject *parent)
    :QObject(parent)
{
    // Pyydetään Plugin Manager -instanssi
    ExtensionSystem::PluginManager *pm;
    pm = ExtensionSystem::PluginManager::instance();

    // Lisätään tämän luokan instanssi oliosäiliöön
    pm->addObject(this);
}

// Lisätty olio on poistettava viimeistään purkajassa
MyClass::~MyClass()
{
    // Pyydetään Plugin Manager -instanssi
    ExtensionSystem::PluginManager *pm;
    pm = ExtensionSystem::PluginManager::instance();

    // Poistetaan tämän luokan instanssi oliosäiliöstä
    pm->removeObject(this);
}
```

Plugin Managerin lisäys- ja poistometodit myös lähettävät signaalin, kun oliosäiliön sisältöön tulee muutoksia. Näiden signaalien prototyypit ovat void `objectAdded(QObject *obj)` ja `void aboutToRemoveObject(QObject *obj)`. Signaalien parametrina kulkee osoitin lisättyyn tai poistettuun olioon. Plugin Managerilla on myös

kolmas hyödyllinen signaali, jonka prototyyppi on `void pluginsChanged()`. Tämä signaali lähetetään silloin, kun ladattujen pluginien määrä muuttuu.

Jokaisella `ExtensionSystem::IPlugin` -luokasta periytyvällä luokalla on myös omat toteutuksensa oliosäiliön käyttöön. Lisäys- ja poistometodit ovat samanlaiset kuin Plugin Managerilla ja näiden metodien toteutuksessa kutsutaan suoraan Plugin Managerin vastaavia metodeita. Jokaisella plugin-luokalla on kuitenkin yksi oma metodi oliosäiliön käyttöön, jonka prototyyppi on `void addAutoReleasedObject(QObject *obj)`. Metodia käytetään kuten normaalia olionlisäysmetodia, mutta plugin-luokan oma sisäinen toteutus huolehtii olion poistosta ja tuhoamisesta. Esimerkissä 13 on näytetty yksinkertainen käyttötapaus.

Esimerkki 13: Olion lisäys `addAutoReleasedObject` -metodilla

```
bool TestPlugin::initialize(const QStringList &arguments,
                            QString *error_message)
{
    Q_UNUSED(arguments)
    Q_UNUSED(error_message)

    ...

    TestSound *sound = new TestSound;

    // Olion poisto ja tuhoaminen tapahtuu automaattisesti
    addAutoReleasedObject(sound);

    return true;
}
```

Kyseistä metodia ei saa käyttää varsinaisen plugin-olion lisäämiseen oliosäiliöön. Tästä aiheutuisi se, että ohjelman sammutusvaiheessa plugin-olio yritettäisiin tuhota kahteen kertaan. /1/ /2/

4.3 Oliokoosteet

Perinteisesti C++-ohjelmoinnissa on yhdistelty luokkien ominaisuuksia toisiinsa moniperinnällä. Tällöin normaalissa C++-ohjelmoinnissa kokoelmasta takaisin alkuperäiseen muotoon muuntaminen tapahtuu `dynamic_cast`:lla ja Qt-ohjelmoinnissa `qobject_cast`:lla. Moniperinnässä on kuitenkin omat ongelmansa, joten sen käyttöä ei juurikaan suositella.

Qt Creator tarjoaa ainutlaatuisen ratkaisun oliokoosteiden luomiseen. Tämä ratkaisu on Aggregation-kirjasto, jota muun muassa Plugin Managerin oliosäiliö käyttää hyväkseen. Kirjaston käytön vaatimuksena on, että oliokoosteeseen lisättävät oliot perivät QObject-luokan ominaisuudet. Kirjaston tiedostot löytyvät Qt Creatorin juurikansiosta lukien `src\libs\aggregation`.

Kirjaston käyttöä varten pitää ohjelmakoodiin lisätä `aggregate.h` -otsikkotiedosto ja projektitiedostoon `aggregation.pri` -otsikkotiedosto, jollei näitä ole jo ennestään. Oliokoosteen käyttöä on esitelty esimerkissä 14.

Esimerkki 14: Oliokoosteen käyttö

```
#include <aggregation/aggregate.h>

// Testiluokka 1
class Test1 : public QObject
{
    Q_OBJECT

public:
    Test1(QObject *parent = 0)
        :QObject(parent) {}

    void printTest1()
    {
        qDebug() << metaObject()->className();
    }
};

// Testiluokka 2
class Test2 : public QObject
{
    Q_OBJECT

public:
    Test2(quint16 nr, QObject *parent = 0)
        :QObject(parent),
          m_refNr(nr) {}

    void printTest2()
    {
        qDebug() << QString("Test2 ref: %1").arg(m_refNr);
    }

private:
    quint16 m_refNr;
};

void MyClass::aggregateMethod()
{
    // Luodaan oliokooste
    Aggregation::Aggregate bundle;

    bundle.add(new Test1);
```

```
    for (int i=0; i < 10; ++i)
        bundle.add(new Test2(i+1));

    // Pyydetään lisätyt oliot koosteesta
    Aggregation::query<Test1>(&bundle)->printTest1();

    QList<Test2*> t2Objects = bundle.components<Test2>();

    foreach (Test2 *obj, t2Objects)
        obj->printTest2();
}
```

Esimerkissä 14 luodaan ensimmäiseksi kaksi luokkaa, jotka tullaan lisäämään oliokoosteeseen. Toteutusosassa luodaan kooste, jonne lisätään yksi Test1-luokan instanssi ja kymmenen Test2-luokan instanssia. Seuraavaksi pyydetään koosteesta Test1-luokan instanssi ja kutsutaan tämän luokan tulostusmetodia. Viimeisenä kohtana pyydetään lista kaikista Test2-luokan instansseista, jonka jälkeen kutsutaan jokaisen instanssin tulostusmetodia.

Oliokoosteen käyttö on varsin helppoa. Koosteeseen voidaan lisätä oliota antamalla koosteen add-metodille osoitin lisättävään olioon. Olioiden pyytäminen koosteesta tapahtuu joko koosteen `component` ja `components` -metodeilla tai Aggregation-nimiavaruudessa olevilla globaaleilla `query` ja `query_all` -metodeilla. Koosteesta poistaminen tapahtuu kutsumalla koosteen `remove`-metodia, jolle annetaan parametrina osoitin poistettavaan olioon.

Ohjelmoijan ei tarvitse huolehtia koosteeseen lisättyjen olioiden tuhoamisesta, koska ne tuhotaan koosteen tuhoutuessa. Tämä myös tarkoittaa sitä, että koosteeseen lisättävät oliot on suositeltavampaa luoda kekoon. Jos koosteeseen lisätään pino-olioita, aiheuttaa koosteen tuhoaminen pinon korruptoitumisen, jollei koostetta tyhjennä ennen sen tuhoamista. Vaihtoehtoisesti koosteeseen lisätty osoitin saattaa osoittaa olemattomaan olioon, jos pino-olion näkyvyysalue loppuu ennen koosteen tuhoamista.

Oliokooste on myös mahdollista tuhota jonkin siihen lisätyn olion kautta. Tämä tarkoittaa myös sitä, että kaikki koosteeseen lisätyt oliot tuhoutuvat samalla. Esimerkissä 15 esitellään kyseisen ominaisuuden käyttöä. /1/ /2/ /4/

Esimerkki 15: Koosteen tuhoaminen jäsenolion kautta

```
// Luodaan oliokooste
Aggregation::Aggregate *bundle = new Aggregation::Aggregate;

bundle->add(new Test1);

for (int i=0; i < 10; ++i)
    bundle->add(new Test2(i+1));

// Pyydetään Test1-luokan instanssi
Test1 *t1 = Aggregation::query<Test1>(bundle);

/*
    Tuhotaan oliokooste ja kaikki siihen lisätyt oliot
    Test1-luokan instanssin kautta

    Saman asian ajaisi: delete bundle;
*/
delete t1;
```

4.4 Managerit

Qt Creatorin hallitseminen tapahtuu pääsääntöisesti erilaisten managerien kautta, joista yleisimmin käytetyt sijaitsevat Core-pluginissa. Näiden managerien käyttöönotto tapahtuu joko pyytämällä tarvittava instanssi Core-pluginin oman instanssimetodin kautta tai kutsumalla suoraan Manager-luokan staattista instanssimetodia. Joidenkin managerien instanssi on myös mahdollista saada yleisen oliosäiliön kautta.

Tässä luvussa tutustutaan yleisimmin käytettyihin managereihin, pois lukien Plugin Manager, jonka käyttötarkoitus on jo käsitelty aiemmin tässä työssä. Erityisesti tässä luvussa keskitytään käyttöliittymän hallitsemiseen, koska se on yleisin syy käyttää manager-rajapintoja. /2/

4.4.1 Core::UniqueIDManager

Unique ID Manager on vastuussa yksilöllisten tunnusten luonnista halutulle komponentille. Yksilöllisellä ID:llä on pääsääntöisesti merkitystä ainoastaan, kun ohjelmaan rekisteröidään uusi välilehti tai kun kaksi toimintoa rekisteröidään samalla tunnuksella. Esimerkissä 16 on esitelty yksilöllisen ID:n käyttöä.

Esimerkki 16: Yksilöllisen ID:n käyttö

```
// Tarvittavat otsikkotiedostot
#include <coreplugin/uniqueidmanager.h>
#include <coreplugin/actionmanager/actionmanager.h>
#include <coreplugin/icore.h>

Core::ICore *core = Core::ICore::instance();

// Pyydetään ID
QList<int> context;
context = QList<int>() << core->uniqueIDManager()->uniqueIdentifier(
                                  QLatin1String("MyPlugin.MyAction"));

QAction *myAction = new QAction(this);
Core::ActionManager *actionManager = core->actionManager();
Core::Command *command;
command = actionManager->registerAction(
                                        myAction,
                                        "MyPlugin.MyAction",
                                        QList<int>() << 0);
Core::Command *command2;
command2 = actionManager->registerAction(
                                         myAction,
                                         "MyPlugin.MyAction",
                                         context);
```

Esimerkin 16 tapauksessa tapahtuisi rekisteröintiristiriita, jos `command` ja `command2` rekisteröitäisiin samalla ID:llä. Toimintojen rekisteröinnistä kerrotaan tarkemmin luvussa 4.4.4.

Unique ID Manager -luokalla on kolme metodia, joiden avulla tunnusten käsittely tapahtuu;

```
bool hasUniqueIdentifier(const QString &id) const;
```

Tämä metodi tutkii, onko parametrina annetulle tunnukselle luotu yksilöllinen ID. Paluuarvo on tosi tai epätosi tuloksesta riippuen.

```
int uniqueIdentifier(const QString &id);
```

Luo parametrina annetulle tunnukselle yksilöllisen ID:n. Paluuarvona palautetaan kokonaisluku, joka on luotu ID;

```
QString stringForUniqueIdentifier(int uid);
```

Tämä metodi toimii käänteisesti edelliseen metodiin nähden. Metodi palauttaa parametrina annettua ID:tä vastaavan tunnuksen. /2/

4.4.2 Core::MessageManager

Message Manager on toiminnaltaan erittäin yksinkertainen, koska sen tehtävä on ainoastaan näyttää annettua tekstiä omassa viesti-ikkunassaan. Luokalla on kolme public slot -tyyppistä metodia, joiden prototyypit on esitelty seuraavaksi. /2/

```
void printToOutputPane(const QString &text, bool bringToForeground);
```

Näyttää ensimmäisenä parametrina annetun tekstin viesti-ikkunassa. Toinen parametri kertoo, tuodaanko viesti-ikkuna käyttäjän näkyville.

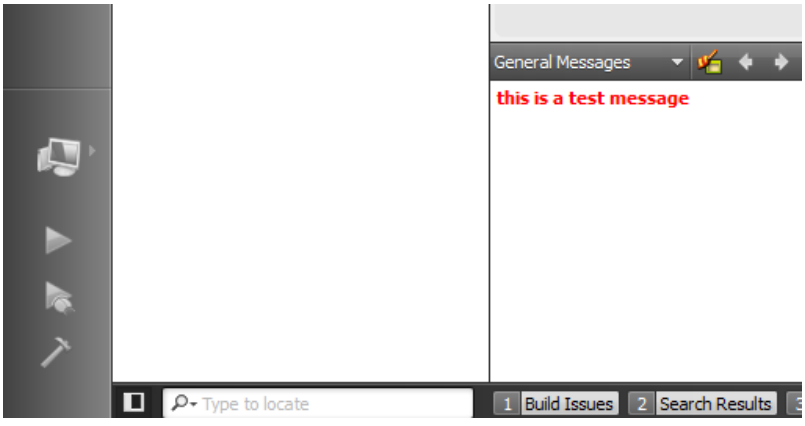
```
void printToOutputPanePopup(const QString &text);
void printToOutputPane(const QString &text);
```

Kaksi jälkimmäistä metodia ovat ainoastaan ennalta määriteltyjä toiminnallisuuksia ensimmäiselle metodille. Erona näillä kahdella on se, että viimeinen metodi ei tuo viesti-ikkunaa käyttäjän näkyville.

Esimerkissä 17 ja kuvassa 9 on esitelty luokan käyttöä.

Esimerkki 17: Viestin näyttäminen viesti-ikkunassa

```
QString msg = "<strong><font color=\"#ff0000\">this is a test mes-
                sage</font></strong>";

Core::ICore::instance()->messageManager()->
                          printToOutputPanePopup(msg);
```



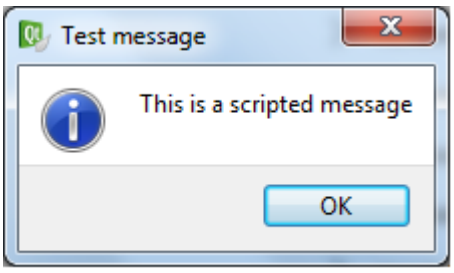
Kuva 9: Esimerkissä 17 luotu viesti

4.4.3 Core::ScriptManager

Script Manager tarjoaa mahdollisuuden suorittaa QtScript-muotoisia scriptejä Qt Creatorissa. Lisäksi kyseinen manageri tarjoaa rajapinnan muun muassa Core-pluginin käyttöön. Ohjelmoijalle on myös tarjottu mahdollisuus käyttää muutamaa valmista käyttöliittymädialogia informaation tai virheviestien näyttämiseen sekä informaation kysymiseen käyttäjältä. Esimerkissä 18 ja kuvassa 10 on näytetty, kuinka käyttäjälle voidaan näyttää yksinkertainen informaatiodialogi.

Esimerkki 18: Dialogin näyttäminen käyttäjälle Script Manager -rajapinnan kautta

```
// Lisätään tarvittava otsikkotiedosto
#include <coreplugin/scriptmanager/scriptmanager.h>

// Pyydetään Script Manager instanssi
Core::ScriptManager *sm = Core::ICore::instance()->scriptManager();

QString err;
bool success;

// Suoritetaan scripti
success = sm->runScript("information(0, \"Test message\", \"This is a
                        scripted message\");", &err);

// Tulostetaan mahdollinen virheviesti konsoliin
if (!success)
    qDebug() << err;
```



Kuva 10: Esimerkissä 18 ohjelmoitu viesti

Scriptien suoritus tapahtuu esimerkin 18 mukaisesti kutsumalla Script Managerin `runScript`-metodia. Tämä metodi ottaa ensimmäisenä parametrinaan QString-tyyppisen olion, johon on tallennettu suoritettava scripti. Toinen parametri on osoitin QString-tyyppiseen olioon, jonne tallennetaan mahdollinen virheviesti. Metodin paluuarvo kertoo suorituksen onnistumisen, jolloin esimerkiksi voidaan tulostaa saatu virheviesti. /2/ /4/

4.4.4 Core::ActionManager

Action Managerin päätehtävät ovat ohjelmavalikkojen ja valikko-osien rekisteröinti sekä pikanäppäinten hallinta. Tästä syystä Action Manager on ehdottomasti yleisimmin käytetty manager-rajapinta.

Omia ohjelmavalikoita tehdessä on syytä huomioida, että ne eivät näy ennen kuin ne on rekisteröity Action Managerin kautta. Lisäksi rekisteröinti pitää tehdä plugin-luokan `initialize`-metodissa tai `extensionsInitialized`-metodissa, koska muutoin Action Manager ei rekisteröi komponentteja käyttöön oikeassa vaiheessa. Sama rekisteröintisääntö pätee myös pikanäppäimille.

Jokainen Qt Creatorin ohjelmavalikko identifioidaan omalla yksilöllisellä nimellään. Näitä nimiä hyödynnetään, kun valmiisiin valikoihin halutaan lisätä uusia toiminnallisuuksia. Yksilöllisiä nimiä hyödynnetään myös, kun uusi valikko halutaan lisätä jo olemassa olevien valikoiden väliin. Valmiit valikot tunnuksineen on listattu taulukossa 1.

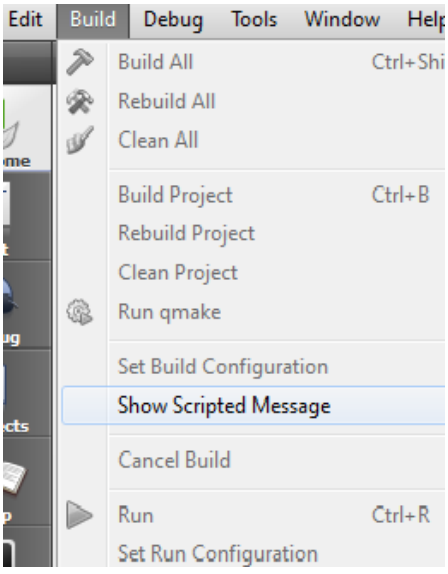
Taulukko 1: Valmiit ohjelmavalikot tunnuksineen

Valikon nimi	**Tunnus**
File	`Core::Constants::M_FILE`
Edit	`Core::Constants::M_EDIT`
Build	`ProjectExplorer::Constants::M_BUILDPROJECT`
Debug	`ProjectExplorer::Constants::M_DEBUG`
Tools	`Core::Constants::M_TOOLS`
Window	`Core::Constants::M_WINDOW`
Help	`Core::Constants::M_HELP`

Esimerkissä 19 on näytetty, kuinka Build-valikkoon voidaan lisätä uusi toiminto. Kuvassa 11 on näytetty esimerkin sijoittuminen valikossa.

Esimerkki 19: Uuden valikkotoiminnan lisääminen

```
// Action Managerin otsikkotiedosto
#include <coreplugin/actionmanager/actionmanager.h>

// Core-plugin vakiot
#include <coreplugin/coreconstants.h>

// Project Explorer –pluginin vakiot
#include <projectexplorer/projectexplorerconstants.h>

// Haetaan Action Manager instanssi
Core::ActionManager *am = core->actionManager();
Core::Command *showCmd; // osoitin varsinaiseen toiminto-olioon

// Rekisteröidään uusi toiminto "TestPlugin.ShowMessage" nimellä
showCmd = am->registerAction(new QAction(this),
                             "TestPlugin.ShowMessage",
                             QList<int>()<<
                              Core::Constants::C_GLOBAL_ID);

// Asetetaan nimi, joka näkyy valikossa
showCmd->action()->setText("Show Scripted Message");

// Pyydetään osoitin Build-valikkoon Action Managerilta
Core::ActionContainer *bp;
bp = am->actionContainer(ProjectExplorer::Constants::M_BUILDPROJECT);

// Lisätään luotu toiminto Build-valikkoon
// jälkimmäinen parametri kertoo ennalta määritetyn ryhmän
bp->addAction(showCmd, ProjectExplorer::Constants::G_BUILD_OTHER);

// Yhdistetään valikkotoiminnon painaminen viestinnäyttömetodiin
connect(showCmd->action(),SIGNAL(triggered()),this,
        SLOT(showMessage()));
```



Kuva 11: Esimerkissä 19 luotu valikkotoiminto

Pikanäppäimen rekisteröinti valikkotoiminnolle ei ole pakollista, mutta suositeltavaa. Rekisteröinti voidaan tehdä kutsumalla `Core::Command`-luokan metodia `setShortcut`. Esimerkissä 20 on näytetty erään pikanäppäimen rekisteröinti.

Esimerkki 20: Pikanäppäimen rekisteröinti

```
/*
    Rekisteröidään uusi toiminto "TestPlugin.ShowMessage" tunnuksella.
    Viimeisellä parametrilla on merkitystä ainoastaan, kun monta
    toimintoa on rekisteröity samalla tunnuksella.
*/
showCmd = am->registerAction(new QAction(this),
                             "TestPlugin.ShowMessage",
                             QList<int>()<<
                              Core::Constants::C_GLOBAL_ID);

// Rekisteröidään toiminnolle pikanäppäin
showCmd->action()->setShortcut(QKeySequence("Shift+F12"));
```

Pikanäppäimen rekisteröintimetodi ottaa parametrinaan QKeySequence-tyyppisen olion, jonka avulla pikanäppäinyhdistelmä ilmoitetaan. Edellisessä esimerkissä rekisteröidään pikanäppäinyhdistelmäksi shift-näppäimen ja F12-näppäimen yhtäaikainen painaminen.

Jos pikanäppäimen rekisteröinti onnistui, tulee pikanäppäimen (tai näppäinyhdistelmän) nimi lukemaan valikkotoiminnon perään, kuten kuvasta 11 voidaan nähdä joidenkin valikkotoimintojen kohdalla. Pikanäppäimien hallinta käytön aikana onnistuu menemällä ohjelman asetuksissa Environment-välilehdelle ja sieltä edelleen Keyboard-välilehdelle. /1/ /2/

4.5 Asetusvälilehden toteuttaminen

Tässä luvussa on tarkoitus näyttää hieman laajempana esimerkkinä, kuinka luodaan oma asetusvälilehti. Esimerkki on varsin hyödyllinen, koska hyvin suunniteltu ohjelma (tai plugin) antaa käyttäjälle mahdollisuuden vaikuttaa erilaisiin toimintoihin. Qt Creator -pluginin tapauksessa on järkevää laittaa oma asetusvalikko sinne, missä muidenkin toiminnallisuuksien asetusvalikot ovat. Asetusikkunan saa avattua Tools-valikon Options-toiminnon kautta.

Oman asetusvälilehden luonti on varsin helppoa ja toteutetaan seuraavien vaiheiden mukaisesti:

1. Toteutetaan `Core::IOptionsPage`-luokka.
2. Lisätään toteutetun luokan osoitin yleiseen oliosäiliöön.

Toteutetun luokan osoittimen lisääminen oliosäiliöön on välttämätöntä, koska muutoin Core-plugin ei osaa asettaa välilehteä näkyväksi. Esimerkissä 21 on näytetty oman asetusvälilehtiluokan otsikkotiedoston toteuttaminen. /1/ /2/

Esimerkki 21: Asetusvälilehtiluokan otsikkotiedosto

```
// Kantaluokan määrittelytiedosto
#include <coreplugin/dialogs/ioptionspage.h>

class OptionsPage : public Core::IOptionsPage
{
    Q_OBJECT

public:
    OptionsPage(QObject *parent = 0);
    ~OptionsPage();

    QString id() const;
    QString displayName() const;
    QString category() const;
    QString displayCategory () const;
    bool matches(const QString &searchKeyWord);

    QWidget *createPage(QWidget *parent);
    void apply();
    void finish();
};
```

Esimerkin 21 metodeista ainoastaan `matches`-metodi on vapaaehtoinen, muut metodit on pakko toteuttaa. Esimerkissä 22 näytetään yksinkertainen luokan toteutus sekä kerrotaan jokaisen metodin merkitys.

Esimerkki 22: Asetusvälilehtiluokan toteutus

```
// Rakentaja
OptionsPage::OptionsPage(QObject *parent)
    :Core::IOptionsPage(parent)
{
    ExtensionSystem::PluginManager *pm;
    pm = ExtensionSystem::PluginManager::instance();

    pm->addObject(this);
}
```

Luokan rakentajassa lisätään luokan instanssi yleiseen oliosäiliöön. Lisäyksen voi myös tehdä vasta, kun tämän luokan instanssi on luotu.

```
// Purkaja
OptionsPage::~OptionsPage()
{
    ExtensionSystem::PluginManager *pm;
    pm = ExtensionSystem::PluginManager::instance();

    pm->removeObject(this);

}
```

Purkajassa tehdään käänteinen toiminto rakentajaan nähden eli poistetaan luokan instanssi oliosäiliöstä. Poistamisen voi myös tehdä muualla koodissa ennen luokan instanssin tuhoamista.

```
QString OptionsPage::id() const
{
    return QString("Test Plugin");
}
```

Id-metodi palauttaa asetusvälilehden yksilöllisen tunnuksen. Tunnus voi käytännössä olla mitä tahansa, kunhan se ei ole ristiriidassa muiden asetusten tunnuksien kanssa.

```
QString OptionsPage::displayName() const
{
    return tr("Test Plugin");
}
```

DisplayName-metodi palauttaa asetusvälilehden nimen, jonka käyttäjä näkee sivupalkissa sekä asetuslehden vasemmassa yläkulmassa. Nimen palautuksessa on syytä käyttää tr-makroa, jotta nimi voidaan tarvittaessa lokalisoida.

```
QString OptionsPage::category() const
{
    return QString("Other");
}
```

Category-metodi palauttaa kategorian tunnuksen, jonka alle asetusvälilehti asetetaan. Tunnus voi olla mitä tahansa, kunhan se on yksilöllinen ja sama kaikilla kategoriaan kuuluvilla jäsenillä.

```
QString OptionsPage::displayCategory() const
{
    return tr("Other");
}
```

DisplayCategory-metodi palauttaa kategorian nimen, jonka käyttäjä näkee. Nimen luonnissa on syytä käyttää tr-makroa, jotta nimi voidaan tarvittaessa lokalisoida.

```
bool OptionsPage::matches(const QString &searchKeyWord)
{
    return displayName().contains(searchKeyWord, Qt::CaseInsensitive);
}
```

Matches-metodi palauttaa boolean-tyyppisen paluuarvon riippuen siitä, täsmääkö parametrina tuleva hakusana ennalta määriteltyihin hakuehtoihin. Tätä metodia ei ole pakko toteuttaa, koska kantaluokassa on olemassa oletustoteutus.

```
QWidget* OptionsPage::createPage(QWidget *parent)
{
    return new QLabel("<strong>Hello options!</strong>", parent);
}
```

Tämä metodi palauttaa osoittimen käyttöliittymäkomponenttiin, jonka käyttäjä näkee. Metodin parametrina tulee osoitin varsinaiseen asetusikkunaan. Asetusvälilehden käyttöliittymäkomponentti tuhotaan asetusikkunan sulkeutuessa, joten käyttöliittymäkomponentti on luotava joka kerralla uudestaan.

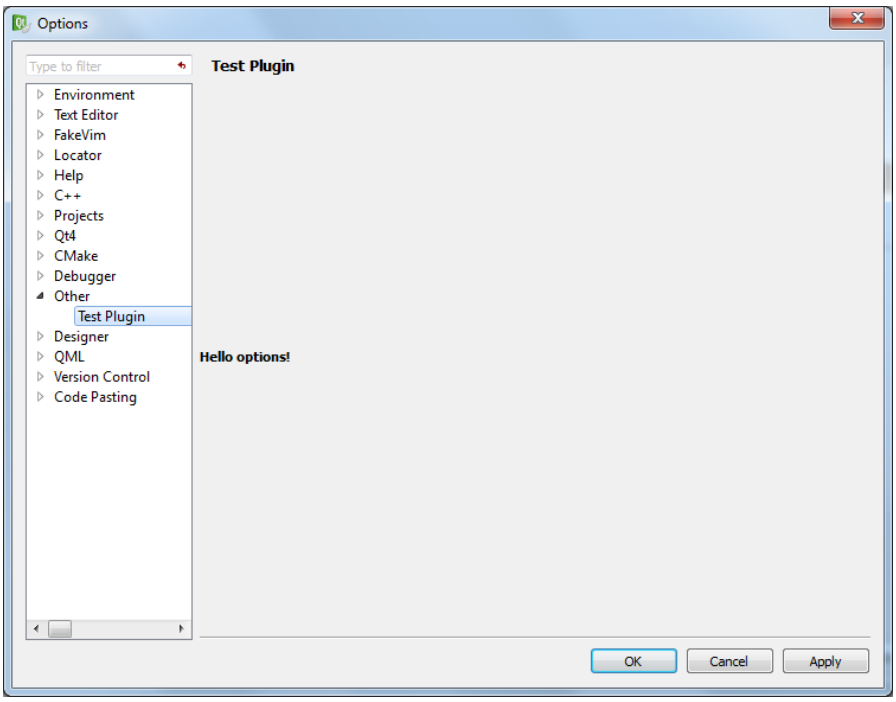
```
void OptionsPage::apply()
{
}
```

Tätä metodia kutsutaan, kun käyttäjä painaa asetusikkunassa Apply-painiketta tai Ok-painiketta. Tässä metodissa kannattaa tehdä asetuksien tallennus.

```
void OptionsPage::finish()
{
}
```

Finish-metodia kutsutaan, kun käyttäjä sulkee asetusikkunan tavalla tai toisella, esimerkiksi Cancel-painiketta painettaessa.

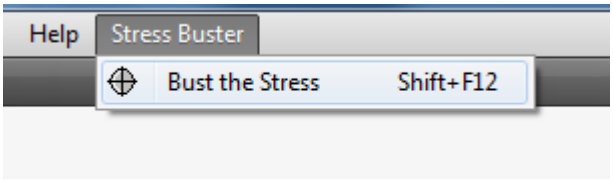
Kuvassa 12 on esitetty kuvankaappaus esimerkeissä 21 ja 22 toteutetusta asetusvälilehdestä.



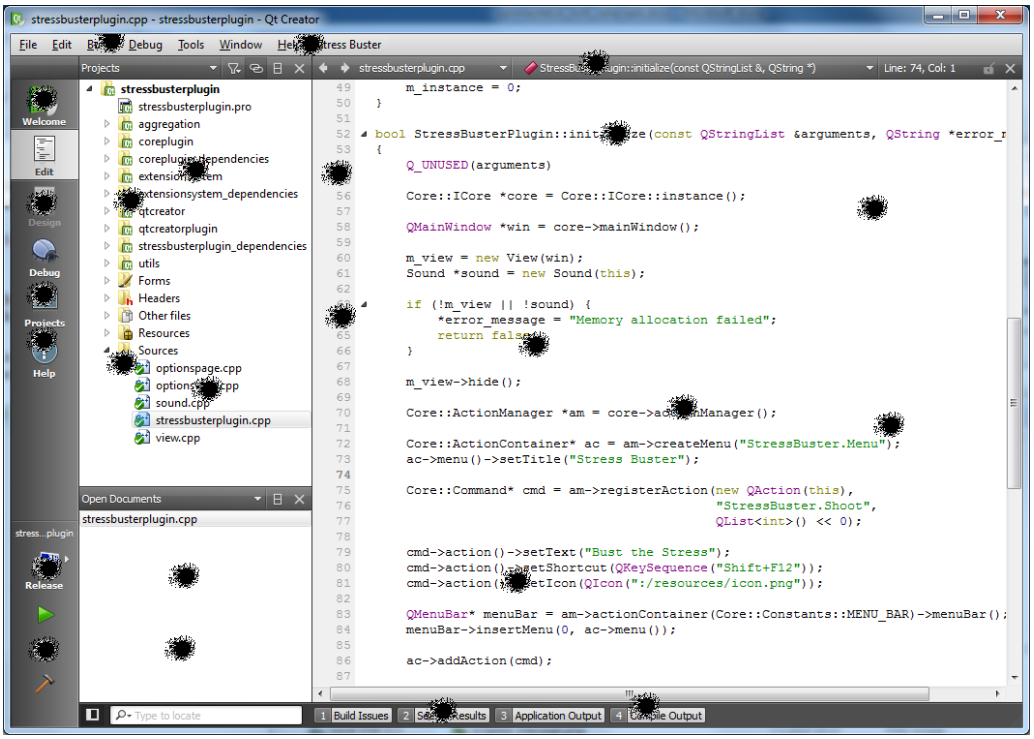
Kuva 12: Toteutettu asetusvälilehti

5 Toteutetun pluginin esittely

Tämän dokumentin ohella toteutettiin Stress Buster -plugin, jonka tarkoitus on toimia stressinlievittäjänä ohjelmoijan turhautuessa. Toimintaperiaate on varsin yksinkertainen; ohjelmoija käynnistää stressinpoistajan joko pikanäppäimellä tai omalla valikkotoiminnollaan, jonka jälkeen ohjelmoija voi "ampua" koodinsa (tai koko ohjelman) täyteen reikiä. Toistaiseksi stressinpoistajan sammutus tapahtuu ainoastaan käyttämällä pikanäppäinyhdistelmää. Kuvissa 13 ja 14 on havainnollistettu valikkorakennetta sekä varsinaista toiminnallisuutta.



Kuva 13: Pluginin käyttöliittymävalikko

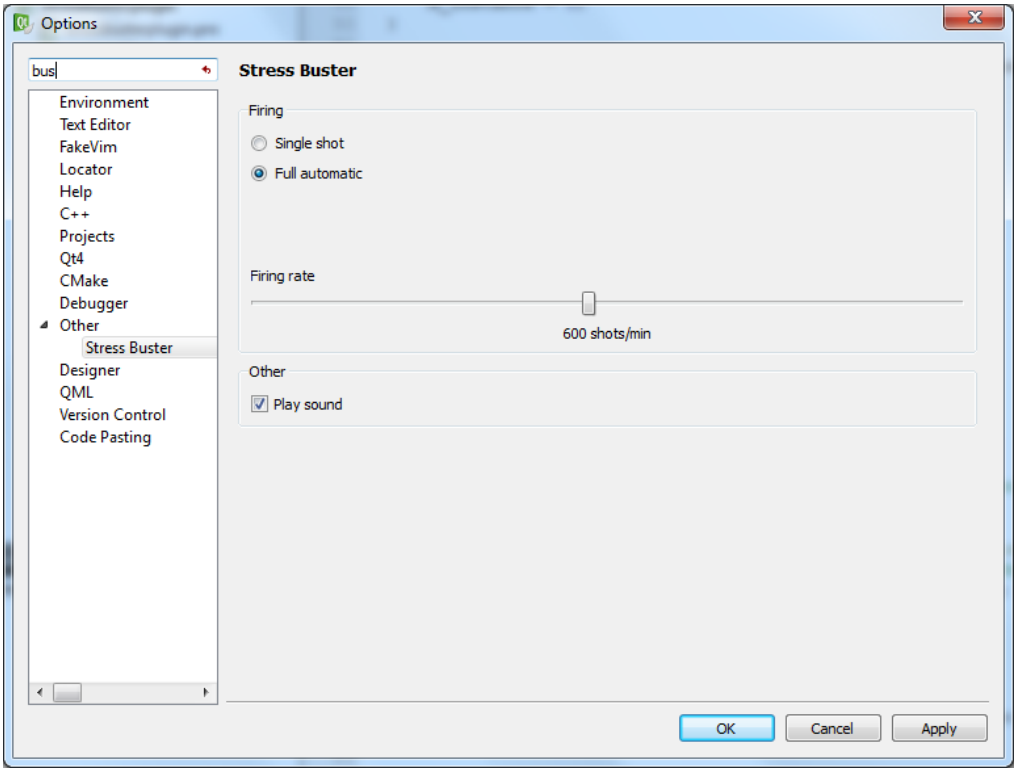


Kuva 14: Stressinpoistaja käytössä

Päätoiminnallisuuden toteutus on teknisesti varsin yksinkertainen. Ohjelman pääikkunan päälle asetetaan läpinäkyvä piirtopinta, jonka jälkeen muutetaan hiiren kursori tähtäimeksi. Käyttäjän painaessa hiiren painiketta piirretään piirtopinnalle luodinreikää muistuttava kuva hiiren kursorin kohdalle. Piirtämisen jälkeen soitetaan aseen laukaisuääntä muistuttava äänitiedosto, jos järjestelmässä on saatavilla tarvittavat äänikirjastot.

Jos käyttäjällä ei ole samaan aikaan muita äänilähteitä käytössä, esimerkiksi musiikkisoitinta, saattaa äänitiedosto toistua väärin käytettäessä sarjatulitoimintoa. Tämä vika on havainnoitu erityisesti Windows-käyttöjärjestelmällä ja se johtuu äänikirjaston toteutuksesta, joka ei sovellu äänitiedoston soittamiseen tiheällä toistovälillä.

Plugin luo itselleen oman asetusvälilehden, jonka kautta käyttäjä voi säädellä muutamaa tarjolla olevaa asetusta. Äänitoimintoon ei voi vaikuttaa, jos Qt ei löydä tarvittavaa kokoonpanoa äänitiedoston soittamiseen. Asetusvälilehden sisältöä on havainnollistettu kuvassa 15.



Kuva 15: Stress Buster -pluginin asetukset

Esitellyn pluginin toimivuutta on kokeiltu Windows-käyttöjärjestelmällä sekä FreeBSD-käyttöjärjestelmällä. Lähdekoodit ovat saatavilla tämän dokumentin liitteenä.

6 Yhteenveto

Qt Creator on avoimeen lähdekoodiin perustuva ohjelmistokehitystyökalu, jonka Qt-pohjaisuus mahdollistaa ohjelman saatavuuden usealle eri käyttöjärjestelmälle. Vaikkakin Qt Creator on pääsääntöisesti tarkoitettu Qt C++-ohjelmointiin, voidaan sillä kehittää myös tavallisia C/C++-projekteja. Viimeisimpien versioiden myötä, ohjelmaan on tullut tuki myös Symbian-ohjelmistokehitykseen.

Qt Creatorin sisäinen toiminnallisuus nojaa täysin erilaisiin plugineihin, mikä mahdollistaa ohjelman ominaisuuksien laajentamisen lähes rajattomasti. Plugin-ohjelmoinnin aloittaminen ei kuitenkaan ole helppoa, koska saatavilla ei ole minkäänlaista valmista SDK:ta (Software Development Kit), joka tarjoaisi tarvittavat ohjelmointirajapinnat. Tästä johtuen, kehittäjän on ensimmäiseksi käännettävä Qt Creator tarvittavien kirjastojen ja otsikkotiedostojen saamiseksi, mikä kasvattaa kehitystyön aloituskynnystä ja vie aikaa varsinaiselta kehitystyöltä. Tämän lisäksi varsinaista ohjelmointiprosessia hidastaa julkisen dokumentaation puuttuminen.

Plugin-kehittäjälle on tarjolla monia valmiita toiminnallisuuksia, joita käyttämällä kehitystyö tehostuu merkittävästi. Yksi tärkeimmistä toiminnallisuuksista on yleinen oliosäiliö, jonka kautta olioinstansseja voidaan jakaa eri pluginien kesken. Toinen tärkeä toiminnallisuus on Qt Creatorin oma oliokoostekirjasto, jonka ansiosta perinteistä moniperintää ei tarvita oliokoosteita luodessa. Lisäksi saatavilla on myös erilaisia rajapintoja, joita toteuttamalla kehittäjä voi luoda esimerkiksi omia asetusvälilehtiä tai editoreita.

Kaiken kaikkiaan plugin-kehitys on aiheena mielenkiintoinen ja hyödyllinen, koska ohjelmistokehittäjä pääsee muokkaamaan ja laajentamaan kehitysympäristöään mieleisekseen. Nykyisen plugin-kehitystyön hankaluus kuitenkin aiheuttaa sen, että kehitystyöhön on varattava tavallista enemmän aikaa. Tämän hetkinen Qt Creator on edelleen erittäin suuren kehitystyön alla, joten rajapintojen vakiintuessa, myös plugin-ohjelmointiin tullaan todennäköisesti tarjoamaan kehittäjiä helpottavia ratkaisuja.

Lähteet

/1/ VCreate Logic 2009. Writing Qt Creator Plugins (Beta). [pdf-tiedosto] [viitattu 14.3.2010]. Saatavissa: http://www.vcreatelogic.com/downloads/files/Writing-Qt-Creator-Plugins.pdf

/2/ Qt Creator 2.0 alpha source code. [online] [viitattu 14.3.2010]. http://qt.gitorious.org/qt-creator/qt-creator/trees/2.0.0-alpha

/3/ Qt Development Tools. [www-sivu] [viitattu 14.3.2010]. http://qt.nokia.com/products/developer-tools?currentflipperobject=821c7594d32e33932297b1e065a976b8

/4/ Qt Reference Documentation. [www-sivu] [viitattu 14.3.2010]. http://doc.trolltech.com/4.7-snapshot/index.html

/5/ Qt Labs Blogs 2010. Qt Creator 2.0 alpha. [www-sivu] [viitattu 16.3.2010]. http://labs.trolltech.com/blogs/2010/03/11/qt-creator-20-alpha/

Liitteet

Liite 1: Toteutetun esimerkkipluginin lähdekoodit

stressbusterplugin.pro

```
TEMPLATE = lib
TARGET = StressBuster
PROVIDER = Mowhi
include(stressbusterplugin_dependencies.pri)
HEADERS += stressbusterplugin.h \
    view.h \
    sound.h \
    stressbusterplugin_export.h \
    optionspage.h \
    optionsview.h \
    settings.h
SOURCES += stressbusterplugin.cpp \
    view.cpp \
    sound.cpp \
    optionspage.cpp \
    optionsview.cpp
DEFINES += STRESSBUSTER_LIBRARY
LIBS += -L$$IDE_PLUGIN_PATH/Nokia
win32:LIBS += -lwinmm
RESOURCES += resources.qrc
OTHER_FILES += StressBuster.pluginspec
FORMS += optionsview.ui
```

stressbusterplugin.pri

```
include(stressbusterplugin_dependencies.pri)
LIBS *= -l$$qtLibraryTarget(StressBuster)
```

stressbusterplugin_dependencies.pri

```
include(../../qtcreatorplugin.pri)
include(../../libs/extensionsystem/extensionsystem.pri)
include(../../plugins/coreplugin/coreplugin.pri)
```

StressBuster.pluginspec

```
<plugin name="StressBuster" version="0.2.3" compatVersion="0.2.2">
    <vendor>Mowhi Corporation</vendor>
    <copyright>(C) 2009 - 2010 Mowhi Corporation</copyright>
    <license>
</license>
    <description>Stress Buster</description>
    <url>http://mowhi.com</url>
    <dependencyList>
        <dependency name="Core" version="1.3.81"/>
    </dependencyList>
</plugin>
```

stressbusterplugin_export.h

```
#ifndef STRESSBUSTER_EXPORT_H
#define STRESSBUSTER_EXPORT_H

#include <QtCore/qglobal.h>

#if defined(STRESSBUSTER_LIBRARY)
#  define STRESSBUSTER_EXPORT Q_DECL_EXPORT
#else
#  define STRESSBUSTER_EXPORT Q_DECL_IMPORT
#endif

#endif
```

stressbusterplugin.h

```
#ifndef STRESSBUSTERPLUGIN_H
#define STRESSBUSTERPLUGIN_H

#include "stressbusterplugin_export.h"
#include <extensionsystem/iplugin.h>

namespace StressBuster {

class View;
struct Settings;

class STRESSBUSTER_EXPORT StressBusterPlugin :
                                        public ExtensionSystem::IPlugin
{
    Q_OBJECT

public:
    StressBusterPlugin();
    ~StressBusterPlugin();

    static StressBusterPlugin* instance();

   bool initialize(const QStringList &arguments, QString
                   *error_message);

    void extensionsInitialized();

    void shutdown();

    Settings* settings() const { return m_settings; }

private slots:
    void toggleState();

private:
    void readSettings();
    static StressBusterPlugin *m_instance;
    View *m_view;
    bool m_running;
    Settings *m_settings;
};
}
#endif
```

stressbusterplugin.cpp

```
#include "stressbusterplugin.h"
#include "view.h"
#include "sound.h"
#include "optionspage.h"
#include "settings.h"

#include <coreplugin/actionmanager/actionmanager.h>
#include <coreplugin/coreconstants.h>
#include <coreplugin/icore.h>
#include <extensionsystem/pluginmanager.h>

#include <QMainWindow>
#include <QtCore/QtPlugin>
#include <QtGui/QAction>
#include <QtGui/QMenu>
#include <QSettings>

using namespace StressBuster;

StressBusterPlugin* StressBusterPlugin::m_instance = 0;

StressBusterPlugin* StressBusterPlugin::instance()
{
    return m_instance;
}

StressBusterPlugin::StressBusterPlugin()
: m_running(false),
  m_settings(new Settings)
{
    m_instance = this;
}

StressBusterPlugin::~StressBusterPlugin()
{
    delete m_settings;

    removeObject(this);
    m_instance = 0;
}

bool StressBusterPlugin::initialize(const QStringList &arguments,
                                    QString *error_message)
{
    Q_UNUSED(arguments)

    Core::ICore *core = Core::ICore::instance();
    readSettings();

    QMainWindow *win = core->mainWindow();

    m_view = new View(win);
    Sound *sound = new Sound(this);

    if (!m_view || !sound) {
        *error_message = "Memory allocation failed";
        return false;
    }

    m_view->hide();
```

```
    Core::ActionManager *am = core->actionManager();

    Core::ActionContainer* ac = am->createMenu("StressBuster.Menu");
    ac->menu()->setTitle("Stress Buster");

    Core::Command* cmd;
    cmd = am->registerAction(new QAction(this),
                             "StressBuster.Shoot",
                             QList<int>() <<
                              Core::Constants::C_GLOBAL_ID);

    cmd->action()->setText("Bust the Stress");
    cmd->action()->setShortcut(QKeySequence("Shift+F12"));
    cmd->action()->setIcon(QIcon(":/resources/icon.png"));

    QMenuBar* menuBar = am->actionContainer(
                            Core::Constants::MENU_BAR)->menuBar();
    menuBar->insertMenu(0, ac->menu());

    ac->addAction(cmd);

    connect(cmd->action(),SIGNAL(triggered()), this,
            SLOT(toggleState()));

    connect(m_view, SIGNAL(mouseClicked()), sound, SLOT(play()));

    addObject(this);

    addAutoReleasedObject(new OptionsPage);

    return true;
}

void StressBusterPlugin::extensionsInitialized()
{
}

void StressBusterPlugin::shutdown()
{
    QSettings *set = Core::ICore::instance()->settings();
    set->beginGroup("StressBuster");
    set->setValue("fireMode", m_settings->m_fireMode);
    set->setValue("speed", m_settings->m_speed);
    set->setValue("sound", m_settings->m_playSound);
    set->endGroup();
}

void StressBusterPlugin::toggleState()
{
    if (m_running) {
        m_view->stop();
        m_running = false;
    } else {
        m_view->start();
        m_running = true;
    }
}
```

```
void StressBusterPlugin::readSettings()
{
    QSettings *set = Core::ICore::instance()->settings();
    set->beginGroup("StressBuster");
    uint fm = static_cast<uint>(m_settings->m_fireMode);
    fm = set->value("fireMode", fm).toUInt();
    m_settings->m_fireMode = static_cast<Settings::FireMode>(fm);
    m_settings->m_speed = set->value("speed", m_settings->m_speed).
                                                        toInt();
    m_settings->m_playSound = set->value("sound", m_settings->
                                                m_playSound).toBool();
    set->endGroup();
}

Q_EXPORT_PLUGIN(StressBusterPlugin)
```

view.h

```
#ifndef VIEW_H
#define VIEW_H

#include <QGraphicsView>
#include <QVector>

class QGraphicsScene;
class QGraphicsPixmapItem;
class QPoint;

namespace StressBuster {

struct Settings;

class View : public QGraphicsView
{
    Q_OBJECT

public:
    View(QWidget *parent=0);
    ~View();

    void start();
    void stop();

signals:
    void mouseClicked();

private slots:
    void pullTheTrigger();

private:
    void clearItems();
    void paintEvent(QPaintEvent *event);
    void mousePressEvent ( QMouseEvent * event );
    void mouseReleaseEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void shoot();

    QGraphicsScene *m_scene;
    QPixmap *m_hole;
    QVector<QGraphicsPixmapItem*> m_items;
    QPoint m_pos;
```

```
    bool m_mousePressed;
    Settings *m_settings;
};

}

#endif
```

view.cpp

```
#include "view.h"
#include "stressbusterplugin.h"
#include "settings.h"
#include <QGraphicsScene>
#include <QGraphicsPixmapItem>
#include <QBitmap>
#include <QMouseEvent>
#include <QTimer>

using namespace StressBuster;

View::View(QWidget *parent)
: QGraphicsView(parent)
{
    setStyleSheet("background: transparent");

    m_scene = new QGraphicsScene;

    setScene(m_scene);

    setCursor(QCursor(QPixmap(":/resources/cursor.png")));

    m_hole = new QPixmap(":/resources/hole.png");
    m_hole->setMask(m_hole->createHeuristicMask(true));

    setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
    setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
    setFrameStyle(0);

    m_settings = StressBusterPlugin::instance()->settings();
}

View::~View()
{
    clearItems();

    delete m_hole;
    delete m_scene;
}

void View::start()
{
    show();

    if (m_settings->m_fireMode == Settings::FireFullAuto)
        setMouseTracking(true);

    m_mousePressed = false;
}
```

```
void View::stop()
{
    hide();
    clearItems();

    if (m_settings->m_fireMode == Settings::FireFullAuto)
        setMouseTracking(false);
}

void View::pullTheTrigger()
{
    if (m_mousePressed)
        shoot();
}

void View::clearItems()
{
    QVector<QGraphicsPixmapItem*>::iterator it;
    QVector<QGraphicsPixmapItem*>::iterator end;

    end = m_items.end();

    for (it = m_items.begin(); it != end; ++it)
        delete *it;

    m_items.clear();
    m_scene->clear();
}

void View::paintEvent(QPaintEvent *event)
{
    QRect mainRect = parentWidget()->rect();

    if (geometry() != mainRect) {
        setGeometry(mainRect);
        setSceneRect(mainRect);
    }

    QGraphicsView::paintEvent(event);
}

void View::mouseReleaseEvent(QMouseEvent *event)
{
    if (event->button() == Qt::LeftButton)
        m_mousePressed = false;
}

void View::mousePressEvent ( QMouseEvent * event )
{
    if (event->button() == Qt::LeftButton) {
        m_mousePressed = true;
        m_pos = event->pos();
        pullTheTrigger();
    }
}

void View::mouseMoveEvent(QMouseEvent *event)
{
    if (m_mousePressed)
        m_pos = event->pos();
}
```

```
void View::shoot()
{
    int x = m_pos.x() - (m_hole->width() >> 1);
    int y = m_pos.y() - (m_hole->height() >> 1);

    QGraphicsPixmapItem *pit = new QGraphicsPixmapItem(*m_hole, 0,
                                                       m_scene);
    pit->setPos(x, y);

    m_items.push_back(pit);

    if (m_settings->m_fireMode == Settings::FireFullAuto)
        QTimer::singleShot(1000/(m_settings->m_speed/60), this,
                           SLOT(pullTheTrigger()));

    emit mouseClicked();
}
```

sound.h

```
#ifndef SOUND_H
#define SOUND_H

#include <QObject>

namespace StressBuster {

struct Settings;

class Sound : public QObject
{
    Q_OBJECT

public:
    Sound(QObject *parent=0);
    ~Sound();

public slots:
    void play();

private:
    void extractSound();
#ifndef Q_OS_WIN32
    QString m_soundFile;
#else
    const char *m_soundData;
#endif
    Settings *m_settings;
};

}

#endif
```

sound.cpp

```
#include "sound.h"
#include "settings.h"
#include "stressbusterplugin.h"
#include <QResource>
#include <QSound>

#ifndef Q_OS_WIN32
#include <QFile>
#include <QDir>
#else
/*
    Native Windows api allows us to play
    the sound direcly from the resources
 */
#include <windows.h>
#endif

/*
    Qt Documentation says:

    QSound X11:

    The Network Audio System is used if available,
    otherwise all operations work silently.
    NAS supports WAVE and AU files.

    So if there is no NAS, there is no sound
 */

using namespace StressBuster;

static const char *FILENAME = "gunshot.wav";

Sound::Sound(QObject *parent)
: QObject(parent)
{
    m_settings = StressBusterPlugin::instance()->settings();

    m_settings->m_soundAvailable = QSound::isAvailable();

    if (m_settings->m_soundAvailable)
        extractSound();
}

Sound::~Sound()
{
#ifndef Q_OS_WIN32
    if (QFile::exists(m_soundFile))
        QFile::remove(m_soundFile);
#endif
}
```

```
void Sound::play()
{
    if (m_settings->m_soundAvailable & m_settings->m_playSound) {
#ifndef Q_OS_WIN32
        QSound::play(m_soundFile);
#else
        PlaySoundA(m_soundData, NULL, SND_MEMORY|SND_ASYNC);
#endif
    }
}

void Sound::extractSound()
{
    QResource res(QString(":/resources/")+QString(FILENAME));
    const char *data = reinterpret_cast<const char*>(res.data());

#ifndef Q_OS_WIN32
    m_soundFile = QString(QDir::tempPath()+
                          QString("/")+QString(FILENAME));

    QFile file(m_soundFile);

    if (file.open(QFile::WriteOnly)) {
        file.write(data, res.size());
        file.close();
    } else {
        m_settings->m_soundAvailable = false;
    }
#else
    m_soundData = data;
#endif
}
```

settings.h

```
#ifndef SETTINGS_H
#define SETTINGS_H

namespace StressBuster {

struct Settings
{
    enum FireMode {
        FireSingleShot,
        FireFullAuto
    };
    FireMode m_fireMode;
    int m_speed;
    bool m_soundAvailable;
    bool m_playSound;

    Settings()
        :m_fireMode(FireSingleShot),
         m_speed(100),
         m_soundAvailable(true),
         m_playSound(true)
    {
    }
};
}
#endif
```

optionspage.h

```
#ifndef OPTIONSPAGE_H
#define OPTIONSPAGE_H

#include "stressbusterplugin_export.h"
#include <coreplugin/dialogs/ioptionspage.h>

namespace StressBuster {

class OptionsView;

class STRESSBUSTER_EXPORT OptionsPage : public Core::IOptionsPage
{
    Q_OBJECT

public:
    OptionsPage(QObject *parent = 0);
    ~OptionsPage();

    QString id() const;
    QString displayName() const;
    QString category() const;
    QString displayCategory() const;
    bool matches(const QString &searchKeyWord);

    QWidget *createPage(QWidget *parent);
    void apply();
    void finish();

private:
    OptionsView *m_view;
};

}

#endif
```

optionspage.cpp

```
#include "optionspage.h"
#include "optionsview.h"
#include "stressbusterplugin.h"
#include "settings.h"
#include "ui_optionsview.h"

using namespace StressBuster;

OptionsPage::OptionsPage(QObject *parent)
    :Core::IOptionsPage(parent)
{
}

OptionsPage::~OptionsPage()
{
}
```

```
QString OptionsPage::id() const
{
    return QString("Stress Buster");
}

QString OptionsPage::displayName() const
{
    return tr("Stress Buster");
}

QString OptionsPage::category() const
{
    return QString("Other");
}

QString OptionsPage::displayCategory() const
{
    return tr("Other");
}

bool OptionsPage::matches(const QString &searchKeyWord)
{
    return displayName().contains(searchKeyWord, Qt::CaseInsensitive);
}

QWidget* OptionsPage::createPage(QWidget *parent)
{
    return (m_view = new OptionsView(parent));
}

void OptionsPage::apply()
{
    Settings *settings = StressBusterPlugin::instance()->settings();
    const Ui::OptionsView *ui = m_view->uiPtr();

    if (ui->radioSingleShot->isChecked()) {
        settings->m_fireMode = Settings::FireSingleShot;
    } else if (ui->radioFullAuto->isChecked()) {
        settings->m_fireMode = Settings::FireFullAuto;
    }

    settings->m_speed = ui->speedSlider->value();
    settings->m_playSound = ui->playSoundBox->isChecked();
}

void OptionsPage::finish()
{
}
```

optionsview.h

```
#ifndef OPTIONSVIEW_H
#define OPTIONSVIEW_H

#include <QWidget>

namespace Ui {
    class OptionsView;
}

namespace StressBuster {

class OptionsView : public QWidget
{
    Q_OBJECT

public:
    OptionsView(QWidget *parent = 0);
    ~OptionsView();
    const Ui::OptionsView *uiPtr() const
    {
        return ui;
    }

protected:
    void changeEvent(QEvent *e);

private slots:
    void enableFullAuto();
    void disableFullAuto();
    void sliderValueChanged(int value);

private:
    Ui::OptionsView *ui;
};

}

#endif
```

optionsview.cpp

```
#include "optionsview.h"
#include "ui_optionsview.h"
#include "stressbusterplugin.h"
#include "settings.h"

using namespace StressBuster;

OptionsView::OptionsView(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::OptionsView)
{
    ui->setupUi(this);

    connect(ui->radioFullAuto, SIGNAL(clicked()), this,
            SLOT(enableFullAuto()));
    connect(ui->radioSingleShot, SIGNAL(clicked()), this,
            SLOT(disableFullAuto()));
    connect(ui->speedSlider, SIGNAL(valueChanged(int)), this,
            SLOT(sliderValueChanged(int)));

    StressBusterPlugin *sbp = StressBusterPlugin::instance();

    Settings *settings = sbp->settings();

    switch (settings->m_fireMode) {
    case Settings::FireSingleShot:
        ui->radioSingleShot->setChecked(true);
        disableFullAuto();
        break;
    case Settings::FireFullAuto:
        ui->radioFullAuto->setChecked(true);
        enableFullAuto();
        break;
    default:
        break;
    }

    ui->speedSlider->setValue(settings->m_speed);

    if (settings->m_soundAvailable) {
        ui->playSoundBox->setChecked(settings->m_playSound);
    } else {
        ui->playSoundBox->setChecked(false);
        ui->playSoundBox->setDisabled(true);
    }

}

OptionsView::~OptionsView()
{
    delete ui;
}
```

```
void OptionsView::changeEvent(QEvent *e)
{
    QWidget::changeEvent(e);
    switch (e->type()) {
    case QEvent::LanguageChange:
        ui->retranslateUi(this);
        break;
    default:
        break;
    }
}

void OptionsView::enableFullAuto()
{
    ui->speedSlider->setEnabled(true);
    ui->speedEcho->setEnabled(true);
    sliderValueChanged(ui->speedSlider->value());
}

void OptionsView::disableFullAuto()
{
    ui->speedSlider->setDisabled(true);
    ui->speedEcho->setDisabled(true);
    ui->speedEcho->clear();
}

void OptionsView::sliderValueChanged(int value)
{
    if (ui->speedSlider->isEnabled())
        ui->speedEcho->setText(QString("%1 shots/min").arg(value));
}
```