VAASAN AMMATTIKORKEAKOULU
VASA YRKESHÖGSKOLA
UNIVERSITY OF APPLIED SCIENCES

Robert Metsäranta

Filhanterare i Qt/C++

Företagsekonomi och turism

2014

VASA YRKESHÖGSKOLA

Utbildningsprogrammet för informationsbehandling

ABSTRAKT

Författare	Robert Metsäranta
Lärdomsprovets titel	Filhanterare i Qt/C++
År	2013
Språk	svenska
Sidantal	43
Handledare	Börje Harju, Kenneth Norrgård

Syftet med lärdomsprovet var att skriva en filhanterare med hjälp av det grafiska ramverket Qt. Eftersom filhanteraren är skriven i Qt borde den gå att användas i alla operativsystem som Qt stöder, detta betyder att den borde kunna användas i bland annat Microsoft Windows samt Linux. Dessa operativsystem är dock väldigt olika så vissa egenskaper fungerar endast i Linux och andra i windows.

Filhanteraren riktar mer in sig på filbläddring än filhanteraring, dock finns de vanligaste egenskaperna som förväntas finnas i filhanterare, så som kopiering av filer, namngivning av filer, flyttning av filer, borttagning av filer med mera. Mest har jag dock riktat mig på egenskaper som gör det lätt att bläddra i filsystemet så som olika vyer att visa upp filer i, samt tabbar (en egenskap som flera filhanterare / filbläddrare inte ännu har idag).

Ämnesord Qt, filehanterare, programmering, C++

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Utbildningsprogrammet för informationsbehandling

ABSTRACT

Author	Robert Metsäranta
Title	A filemanager in Qt/C++
Year	2013
Language	Swedish
Pages	43
Name of Supervisor	Börje Harju, Kenneth Norrgård

The purpose of this thesis was to write a filemanager with the help of the graphical framework Qt. As the file manager is written with Qt it is usable in every operating system supported by Qt, this means that it should be usable in Microsoft Windows and Linux amongst others. These operating systems are very different so some features will only be available in Linux and some in Windows.

The file manager focuses more on file browsing then file managing, the core features that are expected of a file manager are present however, like copying files, renaming files, moving files and deleting files amongst other features. Mostly I have focused on features that makes it easier to browse the file system like different views to display files with, and tabs (a feature that is still not present in all filemanagers).

Keywords Qt, filemanager, programming, C++

INNEHÅLL

BETECKNINGAR OCH FÖRKORTNINGAR

Variabel	På engelska variable. En variabel är ett värde som är sparat i arbetsminnet.[7]
Pekare	På engelska pointer. En pekare är speciell typ av variabel som istället för att ha ett värde pekar mot en address i arbetsminnet var värdet man är intresserad av finns.[7]
utvecklingsmiljö	På engelska integrated development environment, förkortas IDE. Ett program som är avsett för att underlätta programmering.[5]
luckor	På engelska slots. Mottagare av signal som Qt's object skickar ut.[5]
miniatyrbild	På engelska thumbnail. En nerskalad version av bilden i fråga.
fönsterkontrol	På engelska window gadget, förkortas ofta ”widget”. Är ett objekt innanför applikations fönstret.
rullningslist	På engelska scrollbar. En fönsterkontroll som finns på sidan om eller under vyer som visar mer material än vad som ryms i vyen samtidigt, med hjälp av denna kan man skrolla genom materialet som inte är synligt för tillfället.
Flik	På engelska tab. En fönsterkontroll som finns på någon sida av en fönsterkontroll, oftast ovanför, där man kan välja vilken fönsterkontroll som skall synnas i denna fönsterkontroll. Det mest använda fliksystemet skulle vara i en modern webbläsare (tex. Firefox) där man kan ha flera webbsidor upp i samma fönster samtidigt

	genom att öppna dem i olika flikar.
Process	På engelska också process. En instans av ett program som körs. [9]
Tråd	På engelska thread. En semi-process, som körs inom en process och på så vis delar samma minne som andra trådar i processsen.[8]

1 INLEDNING

Det finns många filhanterare för många operativsystem, dessa filhanterare är oftast bundna till ett operativsystem, tex. Microsoft Windows. Det finns nog även filhanterare som inte är operativsystemsbundna, men dessa kräver oftast att man installerar massor av bibliotek för att dessa skall kunna köras. Syftet med denna filhanterare är att den skall kunna köras på flera platformar med bara Qt bibliotek installerade. Jag har valt detta ämne för att jag ville lära mig att programmera i C++ med hjälp av Qt.

Arbetet består av en en teoridel och en praktisk del. I Kapitel 2 redogörs för programmeringsspråket arbetet är programmerat i, C++, där redogörs bland annat lite om språkets historia samt vad det betyder att språket är objektorienterat.

I kapitel 3 redogörs för ramverket, Qt, som använts för programmeringen, samt vad ett ramverk är och vilka teknologier som är specifika för just Qt, hur man skapar användargränssitt med hjälp av Qt samt Qt's egna utvecklingsmiljö som använts för att skapa programmet.

I kapitel 4 redogörs för hur programmet kom igång och hur utvecklingen av programmet har framskridit.

I kapitel 5 redogörs skilt för sig om varje vy som finns i programmet som visar upp data både hur filsystemet representeras i filsystemsvyerna samt hur bokmärken uppvisas i bokmärkesvyn.

I kapitel 6 redogörs för andra fönsterkontroller som programmet visar upp.

I kapitel 7 redogörs för några exempel på utmaningar som kommit upp under utvecklingen av programmet.

Programmet är licenserat med GPL och finns att laddas ner på sourceforge, mer om detta i slutledningen. Detta för att jag använder själv mest GPL licenserad opensource programvara och tycker att opensource andan är bra för att skapa bra program som ett community.

2 C++

C++ är ett programmeringspråk skrivet av Bjarne Stroustrup. C++ är ett objekt-orienterat programmeringsspråk och inte procedurellt. Programmeringsspråket C är procedurellt som C++ är baserat på. C++ är ett ISO-standardiserat programmeringsspråk. C++ är ett kompilerat programmeringsspråk, vilket gör det möiligt för C++ att vara ett av de snabbaste programmeringsspråken i världen.[2]

2.1 C++ historia

C++ fick sitt början då Bjarne Stroustrup började skriva sin doktorsavhandling på Cambridge University år 1979. Ett programmeringsspråk som Bjarne Stroustrup hade arbetat med kallat Simula 67 var det första objektorienterade programmeringsspråket, Bjarne Stroustrup tyckte att denna modellen var ytterst användbar i programmutveckling.[2]

Efter sin doktorsavhandling började Bjarne utveckla C++, vilket han då kallade för ”C with Classes” vilket skulle bli ”C med klasser” på svenska. Med detta hade han som mål att lägga till objektorienterad funktionalitet till C programmeringsspråket. Detta gör C++ till en vidareutveckling av C, som endast lägger till funktionalitet, så allt som finns tillgängligt i C finns även i C++.[2]

Eftersom C++ var ett nytt språk måste Bjarne Stroustrup även skriva en kompilerare för att kunna omvandla C++ till maskinkod som maskinen förstår. Så kom Cfront till, vilket var en kompilerare baserad på Cpre. Detta var ett program som översatte C++ kod till vanlig C kod, som sen gick att kompilera som C kod. Denna kompilerare var skriven i C++ och kunde kompilera sig själv. År 1993 lade man ner utvecklingen av denna kompilerare på grund av problematik att lägga till ny funktionalitet.[2]

Det var först år 1983 som C++ fick namnet C++, hittils hade det varit C with Classes. Beteckningen ++ kommer från programmeringsspråket C operatorn som ökar på värdet på en variabel, som om vi deklarerade en heltalsvariabel (integer)

med värdet 0 och ville öka det till 1 så kunde vi: int i = 0; i++; variabeln ”i” skulle nu ha värdet 1. Detta förklarar ganska bra vad Bjarne tänkte sig att C++ skulle ha för betydelse.[2]

År 1985 blev C++ verkställt som en kommersiell produkt, C++ var vid detta skede ännu inte ett standardiserat språk, därför var det viktigt att detta år gav Bjarne Stroustrup även ut boken ”The C++ Programming Language”, som användes som hänvisning till C++ programmeringsspråket.[2]

År 1990 lanserades en ny kompilerare för C++ kallad ”Borlands Turbo C++ compiler” som en kommersiell produkt. Denna tillade mycket ny funktionalitet till C++ och bidrog så till utvecklingen av C++.[2]

År 1998 publiserades första internationella standarden för C++ kallad ”C++ ISO/IEC 14882:1998” av C++ standarder kommitteen, vilken är inofficiellt känd som ”C++98”.[2]

År 2005 hade C++ standardernas kommittee gett ut en report var det fanns detaljer om flera egenskaper som var planerade för nästa C++ standard. Denna nya standarden är inofficiellt känd med namnet ”C++0x”, Nollan kommer från att man från första början hade förväntat sig att denna standard skulle genomföras under första decenniet efter år 2000.[2]

År 2011 var den nya standarden (före detta kallad C++0x) färdig och blev kallad ”C++11”. En notervärd ny egenskap var stöd för reguljära uttryck vilket är ett sätt att beskriva och söka text strängar genom uttryck. [2]

2.2 Objektorienterat programmeringsspråk

Det finns flera olika tekniker att programmera i, här kommer en kort förklaring på vilka som finns.

Tabell 1. nedan en tabell som listar upp olika programmeringsspråkstyper.

Ostrukturerad programmering	Här modifieras data genom en serie av kommandon eller påståenden genom hela programmet.
Procedurmässig programmering	Här kan man kombinera återkommande sekvenser av påståenden till en plats. Man kallar på en metod för att uföra en serie av instruktioner, efter att instruktionerna blivit utförda så fortsätter koden att köras där varifrån metoden blev kallad.
Modulär programmering	Här grupperas metoder med liknande funktionalitet i moduler. På detta sättet är programmet uppspjälkt i många små logiska delar istället för att vara en lång serie av kommandon.
Objekt-orienterad programmering	Här har vi till skillnad från alla andra programmeringssätt ett nät av objekt som kommunicerar med varandra men varje objekt har sitt eget tillstånd.

I objektorienterad programmering delar man programmet i abstrakta data typer. Till exempel ifall man skall skriva ett program som administrerar arbetstagare i ett företag skulle man ha en modell ”anställd” med egenskaperna födelsedatum, förnamn, efternamn, kontor, uppgifter.[3][4]

Dessa egenskaper kallas data som modellen innehåller. Denna data struktur kommer man åt endast genom definierade operationer, dessa operationer kallas för en gränsyta(interface), vilket är ett sätt för två föremål att kommunicera. När man skapar en ny anställd skapar man en såkallad instans av den abstrakta datatypen anställd, denna fylls nu med data. Man kan skapa hur många instanser som helst av datatypen anställd med olik data i varje anställd.[3][4]

3 QT

Qt är ett ledande platformsoberoende ramverk som används för att utveckla system för olika platformer som skrivbord, inbäddade platformer samt mobila platformer. Qt använder sig av programmeringsspråket C++ och är mycket använt för att utveckla grafiska och text baserade program. Qt ägs för tillfället av Digia, före det ägdes det av Nokia och före det Trolltech som var första att äga Qt.

3.1 Historia

Det hela började när Eirik Chambe-Eng VD och grundare för det Norska företaget Trolltech och Haavard Nord satte på en parkbänk en solig sommardag utanför ett sjukhus i Trondheim, Norge sommaren 1991. ”Vi behöver ett objektorienterat uppvisnings system” sade Nord, ”Vad är det?” tänkte Chambe-Eng. Då sketchade Haavard upp någonting som då verkade som en uppenbar idê. Ett bibliotek av användargränssnittskomponenter skrivna i C++ som skulle ha samma Api på alla platformer.

Tre år senare, efter många upprepningar och mycket hackande inkorporerade Chambe-Eng och Nord Trolltech. De skulle nu arbeta med sin stora passion, det vill säga att göra livet för programmerare lättare och trevligare.

Chambe-Eng och Nord satte igång och utformade och omutformade och hamnade slänga bort mycket kod förrän de var nöjda. De försökte hitta den absolut bästa lösningen på alla problem, ifall någonting gick att skriva om med bara en mindre kodrad men ändå vara läsbart -det är viktigt att koden är läsbar men också att koden inte är onödigt komplicerad, så gjorde de om utformningen.

Efter att Trolltech hade blivit ett större företag med många anställda lämnade Nord och Chambe-Eng Qt's öde åt mer kapabla programmerare än vad de var.

Matthias Ettrich (som grundade det välkända skrivbordet KDE för GNU/Linux) och Lars Knoll (som grundade KDE's html uppritningsmotor KHTML) skulle

börja leda teamet som skulle fortsätta utveckla Qt.

När Qt's version 4 kom ut (Qt 4) så var den nästan helt och hållet en komplett omskrivning av hela Qt. En del som utvecklarna av Qt är mest stolta över är den nya uppritningsmotorn som skulle klara av att rita upp alla fina effekter som dator användare har börjat förvänta sig i grafiska applikationer.[1]

3.2 Stödda plattformar

Qt är plattformsoberoende och stöder de mest använda plattformarna i dagens läge. Detta gör det möiligt att utveckla ett program en gång och sean kompilera det för de stödda plattformerna som Qt stöder utan att skriva om det.[5][6]

Man kan utveckla med Qt på dessa stationära plattformar: Microsoft Windows, Linux/X11, Mac OS X.[5][6]

Man kan utveckla med Qt på dessa inbäddade platformar: inbäddad Linux och Windows samt vissa realtids plattformer (tex. INTEGRITY).[5][6]

Mobila platformar:

I framtiden kommer Qt att stöda alla större mobila platformar, det finns dock redan stöd för bland andra: Android (Googles mobila platform), iOS (Apples mobila platform), Windows 8 (Microsofts Platform).[5][6]

3.3 Qt Teknologier

3.3.1 Signals and slot (signaler och luckor)

Qt har ett eget system för att objekt skall kunna kommunicera med varandra. I stället för det osäkra traditionella återanropningssystemet (callback) som de flesta andra ramverk erbjuder har Qt signaler och luckor. Detta är en central egenskap i Qt och möiligtvis den egenskap var Qt är mest olik andra ramverk. Då en fönsterkontroll ändras så vill man oftast att andra fönsterkontroller får veta att denna fönsterkontroll har ändrats på något vis. Då tex. Stäng knappen klickas i ett fönster klickas så vill vi kanske att fönstrets stäng funktion upprings.[5]

Andra ramverk erbjuder denna funktionaliteten genom återanropning. En återanropning är en pekare (pointer) som pekar mot en funktion, så när vi vill att den funktionen som körs skall meddela oss om någon händelse så skickar vi en pekare till en annan funktion, i detta fall den återuppringda funktionen. Den återanropade funktionen ringer då tillbaka när det är lämpligt. [5]

återanropning har två grundläggande fel: det är inte typsäkert, vi kan aldrig vara säkra på att funktionen som körs kommer att ringa upp återanropningfunktionen med de rätta argumenten. Sen är återanropningfunktionen väldigt ihopparad med funktionen som körs, eftersom den måste veta vilken funktion som den skall ringa upp.[5]

I Qt sköts detta via signaler och luckor. I Qt har klasserna som baseras på QObject fördefinierade signaler och luckor. Programmeraren kan även lägga till sina egna signaler och luckor genom att skriva en ny klass baserad på en QObject klass, en såkallad underklass eller subklass. Tex. Ifall vi har en knapp som skickar ut en signal när den har blivit klickad, om vi vill även veta när knappen får muspekaren över sig, så skulle vi då skriva en ny klass baserad på Qt's knapp-klass (QpushButton) och lägga till denna signalen samt se till att varje gång användaren för muspekaren över knappen skicka ut signalen. Man kan även lägga till nya

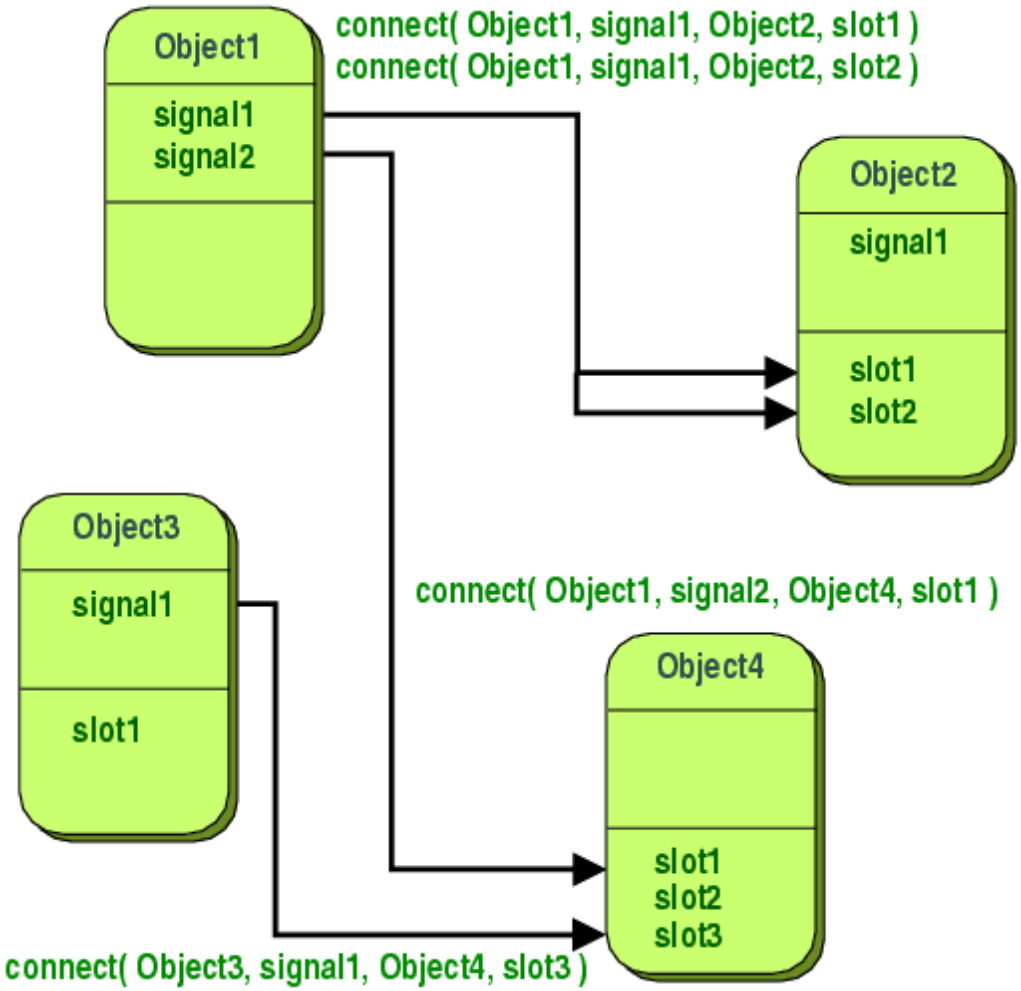
luckor (slots), som tex. Ifall vi vill att när något händer i fönstret så skall knappen bli röd. Då måste fönstret skicka ut en signal och knappen ha en passande lucka så att man kan koppla fönstrets signal till knappens lucka.[5]

Signaler och luckor är ett typsäkert sätt för objekt att kommunicera. Signalerna och luckorna måste ha matchande signaturer. Med undantaget att signalen kan ha en längre signatur än luckan, så ifall signalen har tex. 3 argument så kan den kopplas till en lucka som inte tar emot några argument, luckan då bara ignorerar argumenten.[5]

Signaler och luckor är löst kopplade, när en klass skickar ut en signal vet den inte och bryr sig inte ifall någon lyssnar på signalen, så ett objekt kan ha flera signaler som den skickar ut som ingen tar emot eller lyssnar på. Detta är allt ett objekt gör i Qt för att kommunicera. På så sätt hålls informationen inkapslad och försäkrar att objektet går att använda som en programvaru komponent. Qt's signalmekanism ser till att ifall man kopplat en signal till en lucka så körs luckan då tiden är rätt. [5]

Luckorna är på samma sätt som signalerna oberoende på ifall något är kopplat till dem. Objektet bryr sig inte ifall något är kopplat och på så sätt försäkrar man sig att objektet är oberoende av andra objekt och försäkrar att man med Qt kan utveckla objekt som är oberoende från varandra. Luckorna kan även användas som vanliga funktioner inom objektet.[5]

Signalerna kan avlyssnas av flera luckor och i olika object och man kan koppla flera signaler till en och samma lucka. [5]



Figur 1. Demonstration av signaler och luckor. [5]

3.3.2 Qt's meta-objekt system

Qt erbjuder ett unikt händelsesystem som är baserat på metaobjekt, signalmekanismen samt Qt's egenskapssystem. Det är detta meta-objekt system som även erbjuder signalmekanismen. Detta meta-objekt system erbjuder även typ information under körtid (då koden körs) och dynamiska egenskaper. Meta-objekt systemet är baserat på tre saker, QObject klassen som erbjuder en bas klass för alla objekt som kan ta vara på denna funktionalitet, ett Q_OBJECT macro som man lägger inne i klassen man vill att skall ha denna funktionalitet, utan detta macro så är det inte möiligt att använda denna funktionalitet. Tredje saken som behövs är MOC (meta-object compiler) som tillfogar all kod som behövs för att

man skall kunna genomföra meta-objekt funktionalitet.[5]

Moc läser källkodsfilen, ifall den hittar mer än en klass med Q_OBJECT-makrot i filen så producerar den en ny källkodsfil för varje klass. Denna källkods fil är sedan kompilerad och linkad med klassens implementation. Den kan även vara inkluderad i klasses källkodsfil, men vanligare är att den linkas med klassen.[5]

Det är möiligt att använda Qobject-klassen utan Q_OBJECT-makrot i klassen men då kan man inte använda sig av någon av denna funktionalitet.[5]

Detta metaobjekt system har även hand om händelser i Qt, och i Qt, så är alla händelser även objekt, dessa händelser är alla härledda från den abstrakta klassen QEvent. Denna klass representerar saker som kommer att hända eller har hänt inom Qt programmet. [5]

Dessa händelser kan bli mottagna och hanterade i alla QObjekt baserade klasser och är speciellt relevanta för såkallade widgets (fönsterkontroller som uppritas). [5]

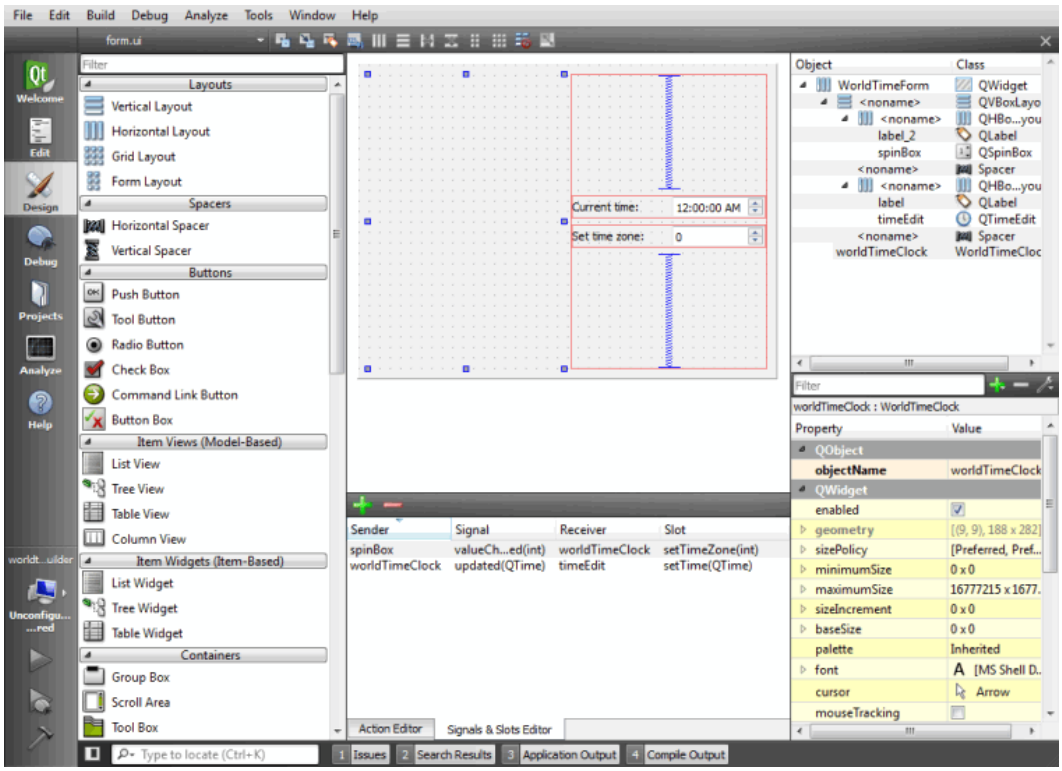
När det händer någonting inom Qt-programmet så konstruerar Qt en ny händelse med den rätta klasses härledd från QEvent klassen, (tex. QMouseEvent ifall andändaren använde musen för att skapa en ny händelse) och levererar den till en särskilld instans av QObject eller en av dess härledda klasser. Objektet då antingen accepterar händelsen eller inte. Det finns även händelser som kommer från systemet och inte användaren, ett exempel skulle vara en timerhändelse QTimerEvent som händer då en timer's intervall har körts färdigt.[5]

3.3.3 Qt-Creator

Qt-Creator är Qt's egna utvecklingsmiljö som erbjuder verktyg som behövs för att designa och utveckla program med Qt ramverket. Qt-Creator erbjuder allt som behövs för att starta ett nytt Qt-projekt, utveckla projektet samt utplacera det till de platformerna som programmet är meningen att användas på.[5]

3.3.4 Skapande av grafiskt användargränssnitt

Det finns flera sätt att skapa grafiska användargränssnitt i Qt, de två vanligaste är att man antingen gör det genom att själva skriva koden som behövs för att lägga till fönsterkontroller till applikationen, eller att man använder Qt-Designer var man kan rita upp fönsterkontroller och Qt skriver då koden för fönsterkontrollerna i en skilld fil. Qt-Designer finns som ett skillt program eller som en integrerad del av Qt-Creator.[5]



Figur 2. Bild av Qt-Creator med den integrerade Qt-Designern. [5]

3.3.5 Vyer och modeller

Qt har ett eget sätt att visa upp data i vyer, ett underliggande datastruktursystem och sen de grafiska vyerna som visar datat åt användaren. Detta ger möiligheten att visa samma data åt användaren i flera olika vyer, utan att ändra på datan. Modellen är den delen som har hand om datan och vyerna visar upp datan. Modellen kommunicerar med vyn genom att skicka ut signaler när datan ändras som vyerna lyssnar efter och ändrar sig därefter.[5]

Alla dessa modellklasser är härledda från den abstrakta klassen QAbstractItemModel som man inte direkt kan använda utan man måste förverkliga viss funktionalitet för att modellen skall vara komplett. Qt erbjuder färdiga implementationer för:[5]

- QStringListModel, en modell som listar upp en simpel lista av text strängar.[5]
- QStandardItemModel, en mer avancerad modell som kan lista upp mer komplexa datastrukturer i en hierarkisk struktur.[5]
- QFileSystemModel, en modell som samlar data från filsystemet.[5]
- Modeller för att komma åt sql databaser: QSqlQueryModel, QSqlTableModel samt QSqlRelationalTableModel .[5]

Jag provade först att använda QFileSystemModel modellen i min filhanterare men det visade sig att jag behövde mer/annorlunda funktionalitet så jag skrev en egen modell på basen av QAbstractItemModel.

Alla vyer som använder dessa modeller är härledda från den abstrakta klassen QAbstractItemView, denna abstrakta klass går precis som QAbstractItemModel inte heller direkt att använda utan man måste förverkliga en viss funktionalitet själv för att klassen skall vara komplett och gå att använda. Qt erbjuder vissa färdiga vyer som direkt går att använda:[5]

- QListView, visar upp data i en lista.[5]
- QTableView, visar upp data i en tabell.[5]
- QTreeView, visar upp data i en hierarkisk struktur.[5]

Dessa vyer använder sig av delegat för att rita upp data samt förser vyn med ett sätt att kommunicera med dess underliggande modell för att editera datat i vyn. Alla dessa delegat är baserade på den abstrakta klassen QAbstractItemDelegate. Vyerna i Qt använder som standard sedan version Qt 4.4 klassen

QStyledItemDelegate för att rita upp data och förse vyerna med ett sätt att editera datat i den underliggande modellen. QStyledItemDelegate samt QItemDelegate är färdiga klasser som erbjuder denna funktionalitet och är oberoende av varandra, skillnaden är att QStyledItemDelegate använder den valda stilen att rita upp datat med.[5]

Qt erbjuder även vissa bekvämlighetsvyer, dessa kommer färdiga med en modell att använda utan att programmeraren själv behöver fundera på hur modellen eller vyn fungerar internt. Dessa är inte avsedda att bli härledda till andra klasser utan är tänkta att användas som sådanna. Exempel av dessa klasser skulle vara:[5]

- QTreeWidget, erbjuder en klass med färdig modell för att lätt lägga till/ta bort data i en hierarkisk struktur.[5]
- QListWidget, erbjuder en klass med färdig modell för att lägga till/ta bort data i en lista.[5]

4 UTVECKLING AV FILHANTERAREN

Då jag började utveckla filhanteraren hade jag länge haft som idé att skriva en grafisk filhanterare med dessa centrala egenskaper: kategoriserade bokmärken (hade inte hittat någon annan filhanterare som hade detta); flikar, de flesta filhanterare numer har detta men ändå inte alla, flera olika vyer för att bläddra genom filerna (De flesta filhanterare erbjuder någon form av olika vyer men jag ville ha mer funktionalitet); optimerad, jag ville skriva filhanteraren i C++ för att jag visste att detta programmeringsspråk var rätt för att skapa snabba optimerade applikationer, så hittade jag ett passande ramverk – Qt; interaktiv uppvisning av sökvägen för mappen man befinner sig i (de flesta filhanterare erbjuder en textbox med hela sökvägen i som man måste editera manuellt); platformsoberoende, jag ville skriva filhanteraren för både windows och linux, Qt erbjuder denna funktionalitet; dynamiskt användargränssnitt, användaren får själv bestämma vissa saker i användargränssnittet, såsom vilka kontroller som skall vara synliga; snabb filtrering av mapp, snabbt filtrera innehållet i en mapp enligt filnamn; miniatyrbilder av filer, mest tänkt för bilder, via ett plugin system; väsentlig funktionalitet för en filhanterare, givetvis skulle filhanteraren vara kapabel av att klara de typiska operationerna som förväntas av en filhanterare, tex. kopiering/flyttning/namngivning/sortering av filer.

4.1 Projektstart

När jag började utveckla filhanteraren hade jag inte mycket kunskap av Qt/C++, så det var ett av målen med att utveckla filhanteraren, att lära sig programmering med C++/Qt. Jag började simpelt efter att ha läst mycket Qt-dokumentation och studerat olika exempel som Qt kommer med lyckades jag skapa ett fönster med två vyer, som kunde kommunicera, när man klickade på något i den ena vyn så hända det något i den andra. Detta var början till bokmärkssystemet som jag visste att jag ville ha i min filhanterare. För detta använda jag en QTeeWidget för bokmärkesvyn samt en QTreeView med en QFileSystemModel för vyn som visar filer och mappar i filsystemet i den andra vyn. För att få något att hända i vyn som visar filsystemet när jag klickar i bokmärkesvyn så använde jag mig av Qt's

signalmekanism. Jag kopplade bokmärksvyns signaler till filsystemsvyns modell som ändrade vilken mapp den visar innehållet för och denna modell sedan signalerade filsystemsvyn att den skall ändra vad den visar. Då var nästa steg i att bygga ett fungerande bokmärkssystem att skriva ett system så att det går att lägga till / ta bort bokmärken ur bokmärkesvyn. Detta gjorde jag först genom att ha ett snabbkommando, så då användaren tryckte in två tangenter på tangentbordet så läggdes mappen som användaren befann sig i till bokmärken. Samt då användaren klickade på bokmärket i bokmärkesvyn laddades innehållet för denna mapp som användaren hade laggt till in i vyn som visar filsystemet. Samma sak då användaren hade valt bokmärket och tryckte på tangenten delete så togs detta bokmärke bort.

5 VYER

5.1 Ikonvy

Ikonvyn är en vy som visar filerna i filsystemet som ikoner med filnamnen under ikonerna. Varje fil/undermapp i mappen som visas i vyn representeras av en ikon med filnamnet under ikonen. Denna vyn är mest tänkt för att bläddra i mappar med bilder i, detta då när man har laddat pluginen för att visa miniatyrbilder direkt i vyn av de bilder som finns i mappen som visas av vyn samt valt i konfigurationsdialoget att man vill se miniatyrbilder av filer som är bilder. För att kunna använda sig av plugin'en som laddar miniatyrbilder måste denna kompileras in skillt så filhanteraren laddar in den i arbetsminnet vid programmets uppstart. Ikonvyn är härledd direkt från QAbstractItemView, vilket betyder att jag inte kunde dra nytta av tex. arrangeringen för ikonerna i vyn som jag skulle ha kunnat gjort ifall jag använt mig av tex. QListView vilken erbjuder flera olika arrangeringar för ikonerna i vyn. Jag valde att härleda från QAbstractItemView istället för att QListView skulle mest varit i vägen för viss funktionalitet som tex. kategoriseringen av ikoner, denna kategorisering kan användaren välja att använda i inställnings dialogen.

Några egenskaper för ikonvyn:

- Användaren kan ställa in ikonstorleken mellan 16 och 256 pixlar.
- Stöd för animerad bläddring när man bläddrar igenom vyn med mushjulet.
- Användaren kan ställa in hur mycket rum det skall reserveras för texten under ikonen, ifall filnamnet är kort lägger detta till tomt utrymme på sidorna av ikonen.
- Användaren kan ställa in hur många rader text som skall reserveras för filnamnet, ifall hela filnamnet ryms på en rad så lägger detta bara till tomt utrymme under ikonen.
- Kategoriserad uppvisning av filer, hur filerna kategoriseras beror på hur

användarent valt att sortera filerna.

Figur 3. Bild av ikonvyn som visar miniatyrbilder.

5.1.1 Kod exempel för ikonvy

Kod för att arrangera ikonerna i vyn.

```
const int hsz = gridSize().width();
const int vsz = gridSize().height();
m_contentsHeight = -vsz;

if (isCategorized())
{
    m_categories = m_model->categories();
    QFont f(font());
    f.setBold(true);
    QFontMetrics fm(f);
    for (int cat = 0; cat < m_categories.count(); ++cat)
    {
        m_contentsHeight+=vsz;
        const int startv = m_contentsHeight;

        m_contentsHeight+=fm.boundingRect(m_categories.at(cat)).height();
        int col = -1;
        const QModelIndexList &block(m_model->category(m_categories.at(cat)));
        for (int i = 0; i < block.count(); ++1)
        {
            if (col+1==m_horItems)
            {
                m_contentsHeight+=vsz;
                col=0;
            }
            else
                ++col;
            const QModelIndex &index(block.at(i));
            if (index.isValid()&&index.internalPointer())
                m_rects.insert(index.internalPointer(), QRect(hsz*col, m_contentsHeight, hsz, vsz));
        }
        m_catRects.insert(m_categories.at(cat), QRect(0, startv, viewport()->width(), (m_contentsHeight+vsz)-startv));
    }
}
else
{
    for (int i = 0; i < m_model->rowCount(rootIndex()); ++i)
    {
        const QModelIndex &index(m_model->index(i, 0, rootIndex()));
        const int col = i % m_horItems;
        m_contentsHeight = vsz * (i / m_horItems);
        if (index.isValid()&&index.internalPointer())
            m_rects.insert(index.internalPointer(), QRect(hsz*col, m_contentsHeight, hsz, vsz));
    }
}
m_contentsHeight+=vsz;
if (m_contentsHeight>viewport()->height())
    verticalScrollBar()->setRange(0, m_contentsHeight-viewport()->height());
else
    verticalScrollBar()->setRange(0, -1);
```

Figur 4: Kod för att räkna ut positionerna för ikonerna i ikonvyn.

Vi börjar med att spara cellernas bredd och höjd i variablerna hsz och vsz, dessa är vanliga heltalsvariabler med typen integer.

Vi lägger variabeln m_contentsHeight att ha värdet av cellhöjden, detta för att i fall vyn visas som kategoriserad så lägger vi till först för varje kategori denna cellhöjd, variabeln m_contentsHeight är en heltalsvariabel och därmed typen integer. Denna används av vyn för att kontrollera bland annat hur mycket det går att bläddra upp och ner i vyn.

Ifall vyn visas som kategoriserad så hämtar vi först kategorierna från vår QAbstractItemModel härlädda model, och sparar dem i variabeln m_categories, denna variabel är av typen QStringList vilket är typdefinition av QList<QString>,

en textlista, denna används av vyn bland annat som hjälp för att rita upp kategorierna.

Vi sparar en fet kopia av fonten som användaren använder i variabeln f, vilken är av typen QFont, detta är en Qt klass som behåller information om fonter.

Vi använder fonten som vi sparat för att kunna räkna ut bredd och höjd på text i variabeln fm, vilken är av typen QFontMetrics, en Qt klass fom erbjuder metrisk infomation om fonter.

Vi går igenom kategorierna en efter en.

Vi börjar med att lägga till höjden för celler i m_contentsHeight variabeln för varje kategori, detta utgör vertikala värdet för var kategorin börjar.

Vi sparar detta värde i startv som vi senare använder för att räkna ut cellen för kategorin.

Vi lägger till höjden på namnet på kategorin i m_contentsHeight.

Vi hämtar en lista på alla index som finns i kategorin från modellen och sparar i variabeln block, vilket är en variabel av typen QModelIndexList, vilket är en typdefinition för QList<QModelIndex> , en lista av modell index QModelIndex, vilket är en klass som representerar ett index i modellen som innehåller information om det index det representerar i modellen.

Vi går igenom alla index i kategorin och räknar ut cellernas plats och storlek för dem.

Variabeln col är en heltalsvariabel var vi sparar information om hur mycket till höger från vänstra kanten av vyn ikonen för indexet skall komma, tex. 0 är första och helt till vänster, sen lägger vi till i den variabeln tills det inte ryms flera ikoner på samma.

Vi kollar col variabeln mot heltalsvariabeln m_horItems vilket är en heltalsvariabel vilken innehåller en nummer på hur många ikoner det ryms på en

rad, vi kontrollerar med col+1 eftersom 0 är första ikonen, inte 1.

Ifall det inte ryms flera ikoner på raden i fråga så lägger vi till cellernas höjd i m_contentsHeight eftersom vi måste börja rada ut ikonen i fråga på nästa rad, då skall row vara 0, ifall det ryms fler ikoner på samma rad så ökar vi bara col variabeln med 1.

Vi sparar indexet vi arbetar med för tillfället i variablen index, vilken är av typ QModelIndex.

Vi kontrollerar ifall indexet är giltigt och har en internalPointer, detta är en intern variabel som modellen använder för att komma åt den underliggande datastrukturen som kopplar modellen till filsystemet.

Vi skapar en cell av typen QRect vilket är en Qt klass som håller information om rektanglar som höjd, bredd och position, till detta använder vi variablerna hsz, col, m_contentsHeight och vsz. För att få den horizontella positionen relativt till den vänstra kanten av vyn räknar vi hsz gånger col, vertikala positionen är m_contentsHeight, som vi plussat till vartefter på, bredden på cellen blir hsz, och höjden vsz. Denna cellen/rektangeln lägger vi där till i variabeln m_rects, vilken är av typen QHash<void *, QRect>, detta är en Qt klass som associerar alla värden man sparar i den i detta fall QRect med i detta fall en void pointer. Denna variabeln använder vyn för att hålla information om var nånstans i vyn ett index finns.

När vi räknat ut alla celler i kategorin gör vi på ett liknande sätt för att räkna ut rektangeln för kategorin i fråga och börjar med nästa kategori.

Ifall vyn dock inte visas som kategoriserad så radar vi bara upp ikonerna i rader av ikoner tills vi kommit till slutet.

Vi går igenom alla index som skall visas i vyn.

Igen sparar vi indexet vi arbetar med i en variabel med identifieraren index, vilket är av typ QModelIndex.

Vi räknar ut den horizontella platsen genom att ta resten av raden för indexet och hur många ikoner som horizontellt ryms i vyn, ifall raden är 22 och m_horItems är 8, så blir resten 6. Då skall denna ikon vara den sjätte ikonen från vänster på raden av ikoner. Detta för vi genom att använda oss av den C++ inbyggda operatorn för modulus vilken representeras av procent tecknet. Detta sparar vi i heltalsvariabeln col.

Vi räknar ut höjden av innehållet i vyn genom att ta raden för indexet delat med mängden ikoner vi vill ha horizontellt som vi sen tar gånger höjden för ikonerna i vyn. Sparar detta i variabeln m_contentsHeight.

Sen gör vi på samma sätt som när vi har kategorisering, vi kontrollerar i fall indexet vi arbetar med för tillfället är giltigt samt har en intern koppling till modellen (internalPointer) , sedan kalkylerar vi rektangeln för indexet och sparar det i m_rects.

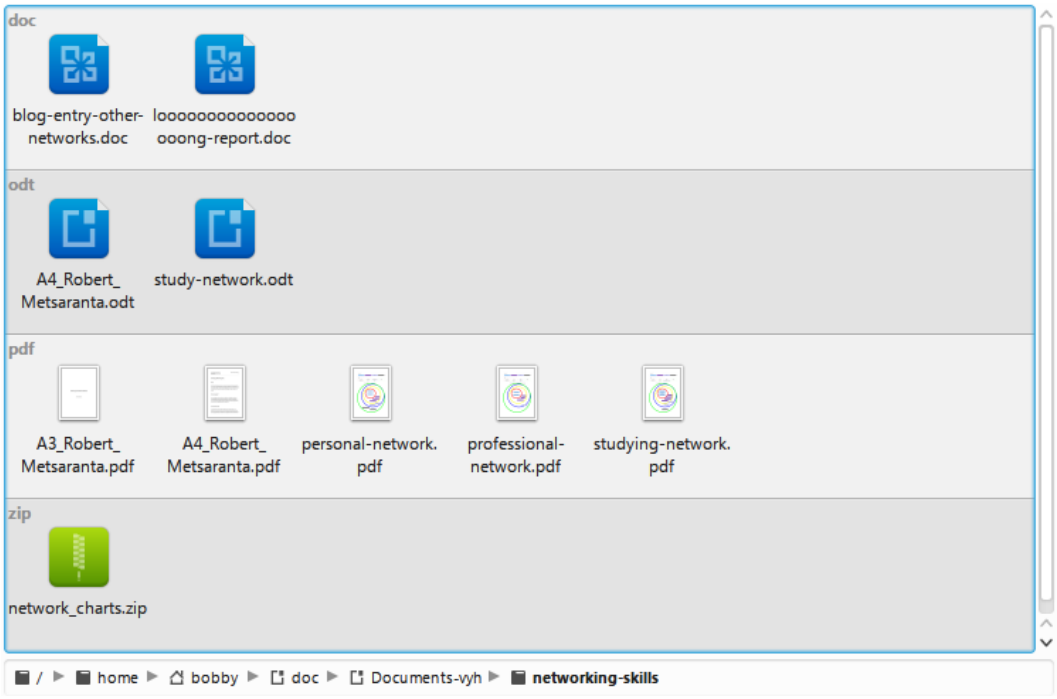
För att ha kalkyrerat höjden för innehållet korrekt så måste vi ännu lägga till höjden för den sista raden av ikoner, detta gör vi genom att bara plussa till värdet på vsz till m_contentsHeight.

Efter att vi kalkylerat höjden för innehållet måste vi uppdatera området för rullningslisten, så att användaren kan bläddra fritt i ikonerna genom att flytta rullningslisten uppåt och nedåt.

Vi kontrollerar först ifall innehållet alls behöver bläddras i, ifall höjden på innehållet är mindre än höjden på vyn så syns alla ikoner redan och man behöver inte kunna bläddra uppåt och nedåt.

Ifall höjden på innehållet i vyn är större än vyns höjd så måste vi kunna bläddra höjden på innehållet – höjden på vyn, så vi updaterar omfånget på rullningslisten.

Ifall höjden är mindre så sätter vi det största giltiga värdet för rullningslisten till mindre än det minsta, vilket gör att rullningslistens enda giltiga värde är det minsta, i detta fall 0.



Figur 5. Bild av ikonvyn som visar filerna i en mapp sorterad enlig typ av fil.

5.2 Detaljvy

Detaljvyn visar filerna och mapparna i en hierarkisk struktur med flera kolumner. Varje kolumn visar olik data om filen/mappen, i denna vy representeras varje fil och mapp av en rad av data. Detaljvyn är härledd från Qt klassen QtreeView. Denna klass erbjuder all funktionalitet för att rada upp datat så jag behövde inte skriva uppradningen själv för detaljvyn. Detta var till en stor hjälp för att få mappstrukturen att uppritas inkrementellt som skulle annars varit en hel del jobb. Varje rad har 4 columner, varje kolumn ger olik information om filen som raden representerar. Kolumnerna visar:

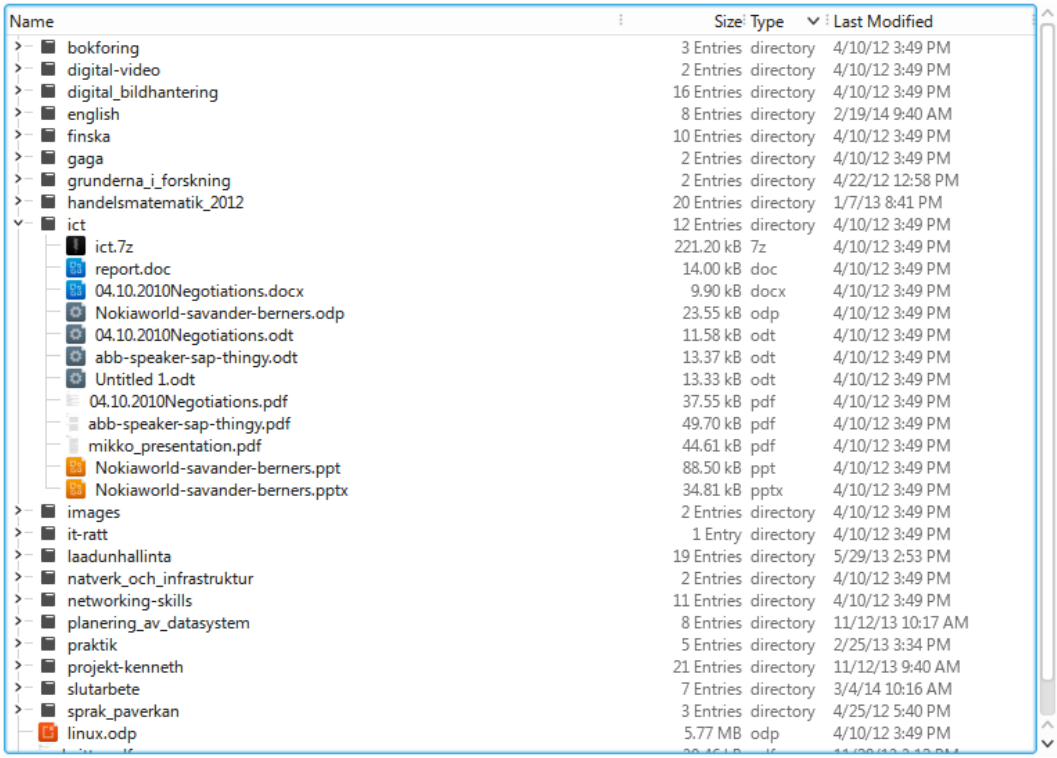
1. Namnet på filen eller mappen i fråga, detta är samma information som ikonvyn även visar, detta är information som varje vy som visar filsystemet förväntas visa. I denna kolumn visas även en liten ikon för att beskriva filen.

2. Storleken på filen. Vilken informationssenhet som anges för att visa storleken beror på storleken på filen. Till exempel ifall filen är under 1 megabyte men över 1 kilobyte så används kilobyte. Ifall filen är en mapp så visas hur många filer och undermappar denna mapp innehåller sammanlagt.

3. Typen på filen, till exempel ifall filen är en mapp så listas directory upp och ifall filen är en png bildfil så listas png upp.

4. När filen senast blivit modifierad, visar datumet och tiden på dygnet då filen blivit modifierad.

Ifall filen som demonstreras i en rad av data är en mapp kan den även expanderas så den visar innehållet i den mappen med litet inbuktning så man ser tydligt att dessa filer tillhör mappen i fråga. Eftersom den underliggande modellen som detaljvyn hämtar data från befolkas med filer och mappar från filsystemet inkrementellt så läser den inte innehållet i mapparna som visas förrän användaren valt att visa innehållet i mapparna antingen genom att expandera dem eller fara in i dem, på detta viset håller man minnesanvändningen på mindre nivå.



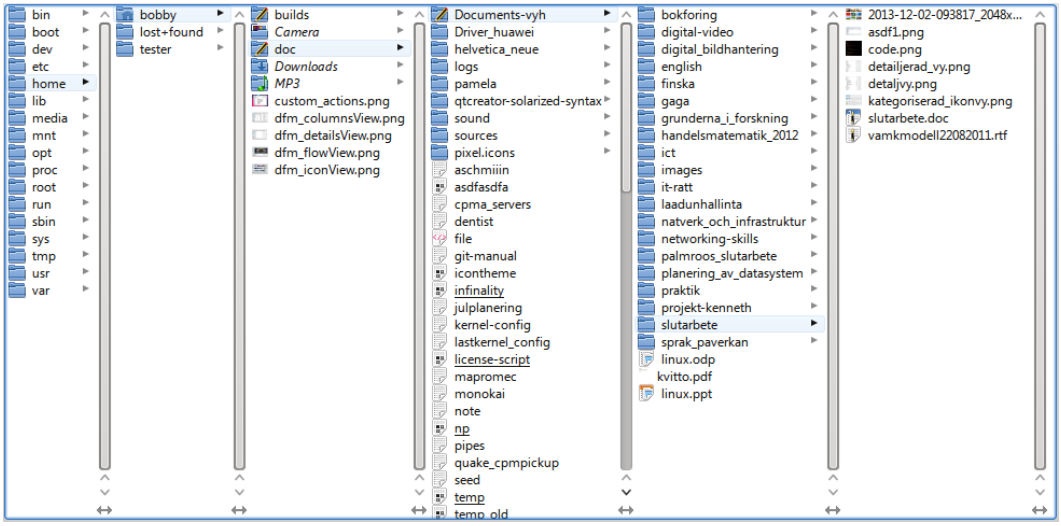
Figur 6. Bild av detaljvyn som visar innehållet i en mapp och undermapp.

5.3 Kolumnvy

Kolumnvyn är en lite mer avancerad vy som i sig inte är en vy utan visar många vyer som kolumner av alla mappar över den man är i samt som sist den man är i t.ex. ifall man befinner sig i mappen C:/users/username så visas innehållet i C: längst till vänster i en vy, sedan innehållet i C:/users i nästa vy mot höger, sedan i nästa visas innehållet i C:/users/username. Syftet med denna vy är att man får en snabb överblick över var i filsystemet filen/filerna man är intresserad av finns. De övre mapparna i strukturen får en liten markering för den undermapp som är den nästa i den hierarkiska mapp strukturen. Den högsta mappen som inte är innanför någon annan mapp visas längst till vänster och mappen man befinner sig i längst till höger. Mapparna blir alltså arrangerade från vänster till höger med den högsta först. Kolumnvyn är härledd från Qt klassen QScrollArea, denna klass är användbar här för att när man har fler kolumner än vad som kan visas samtidigt så förser denna klass automatiskt en horisontell rullningslist så man kan bläddra horisontellt mellan de synliga vyerna. Vyn i sig som visar data från filsystemet är härledd från Qt klassen QlistView.

Några egenskaper för kolumnvyn:

- Användaren kan förstora/förminska kolumnerna individuellt.
- Ifall sökvägen till mappen man befinner sig i har mycket övermappar kan man lätt bläddra horisontellt med en horisontellt med en rullningslist.



Figur 7. Kolumnvyn visar en sökväg i kolumner.

5.4 Flödesvy

Flödesvyn är en vy som är mest tänkt för att bläddra igenom mappar med fotografier i, den visar stora miniatyrbilder i ett ständigt flöde då användaren använder rullningslisten för att visa nästa/föregående bild i mappen. Den filen som visas för tillfället placeras i mitten och de andra aningen svängda. De föregående filerna kommer till vänster om den nuvisande filen och de nästa bilderna till höger om mittenbilden. Nedan on flödesvyn finns en vanlig detaljerad vy som följer flödesvyn så ifall användaren bläddrar i innehållet i flödesvyn så följer den detaljerade vyn med så man lätt kan välja och manipulera filer, detta är inte själva flödesvyn tänkt för, den är enkom tänkt för att användaren kan snabbt bläddra igenom bilder och snabbt hitta den bild denne söker efter eller bara annars bläddra igenom bilder i en mapp.

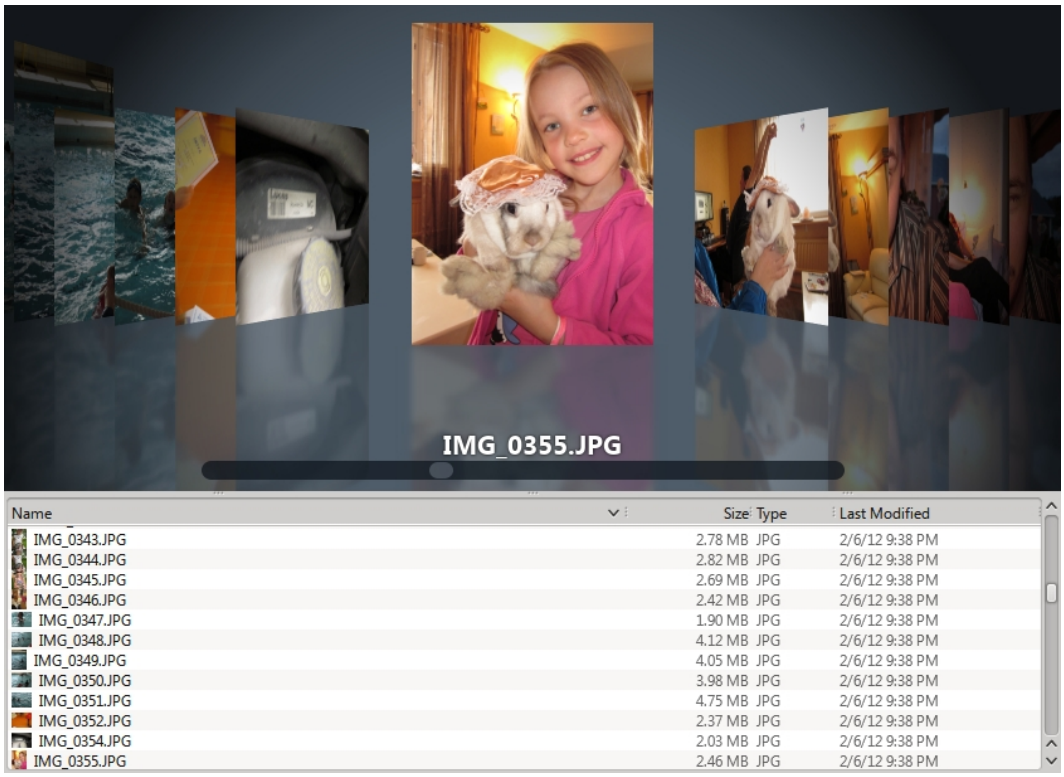
Några egenskaper för flödesvyn:

- Animerad bläddring av bilder, ifall användaren klickar på en bild som inte är den i mitten eller på annat sätt går vidare till nästa bild animeras detta så användaren lätt kan gå igenom bilder i ett ständigt flöde av bilder.

- Vyn har en rullningslist som användaren även kan använda sig av för att snabbt bläddra i filerna i mappen.

- Använder OpenGL som hjälp för uppritning för att förbättra prestandan.

- Använder sig av samma miniatyrbilds bibliotek som resten av vyerna så alla miniatyrbilder som laddas av resten av vyerna laddas även av flödesvyn.



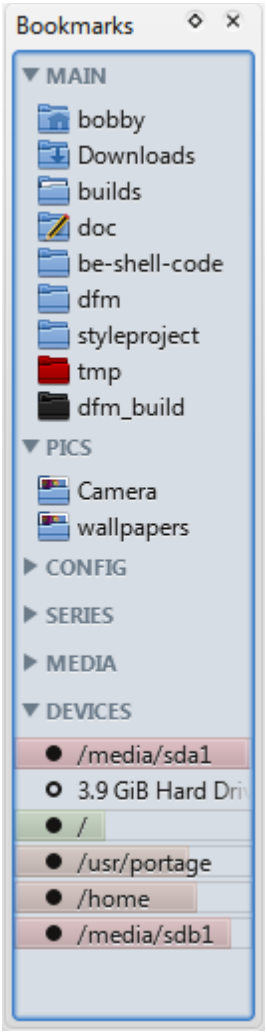
Figur 8. Bild av flödesvyn som visar fotografier i en mapp.

5.5 Bokmärkesvy

Bokmärkesvyn är en vy där användaren själv kan lägga till mappar för att snabbt komma åt dem eller minnas dem för senare bruk. För att lägga till mappar som bokmärken kan man anting dra mappar mellan existerande bokmärkta mappar eller högerclicka på en kategori och välja 'Add Place' i menyn, då läggs mappen man befinner sig i till som ett bokmärke i denna kategori. Ifall användaren drar en eller flera mappar på ett tomt område i vyn så läggs automatiskt dessa i en kategori som heter 'Places' ifall denna kategori inte finns så skapas den. Före man laggt till bokmärken finns det inga bokmärken i denna vy. Vyn är härledd från Qt klassen QtreeView.

Några egenskaper för bokmärkesvyn:

- Stöd för att användaren kan ordna bokmärken genom att dra och släppa.
- Ifall användaren drar och släpper en eller flera filer direkt över ett bokmärke så flyttas dessa filer till den mapp som bokmärket representerar.
- Visar alla enheter som kan användas att spara data på i en kategori kallad ”Devices”, i denna kategori kan användaren inte själv lägga till bokmärken, här visas enkom enheter med sparningskapaciteter såsom usb minnen och hård diskar.
- Ifall en enhet kommer till eller tas bort updateras ”Devices”.
- Användaren kan själv ändra namn på bokmärkerna genom att högerklicka på ett bokmärke och välja ”Rename”.
- Användaren kan lätt öppna en bokmärkt mapp i en ny flik genom att klicka med mittersta knappen på ett bokmärke.



Figur 8. Alla bokmärkta bokmärken samt kopplade enheter.

6 ANDRA FÖNSTERKONTROLLER

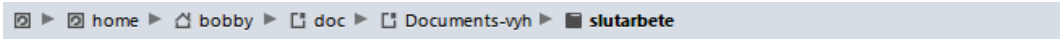
6.1 Sökvägsnavigeraren

Sökvägsnavigeraren hjälper användaren att lätt hålla reda på var någonstans i filsystemet användaren befinner sig. Den befinner sig antingen under eller ovanför vyerna som visar filsystemet eller så visas den inte alls, detta beror på hur användaren konfigurerat programmet.

Vad denna fönsterkontroll gör är att den listar upp alla mappar ovanför mappen som man befinner sig i som klickbara objekt. Den har även klickbara trianglar som gör att användaren kan lätt navigera sig till vilken mapp som helst av de mappar som finns ovanför denna mapp i filsystemet.

Några egenskaper för sökvägsnavigeraren:

- Användaren kan lätt klicka på mappnamnen som om de var länkar för att visa innehållet i den mappen.
- Går att lägga i editeringsläge där användaren själv kan skriva in en sökväg manuellt.
- Har en indikator som indikerar när filhanteraren arbetar.
- Klickbara trianglar som listar upp undermappar för mappen i fråga.



Figur 9. Bild av sökvägsnavigeraren som visar klickbara knappar.

Figur 10. Bild av sökvägsnavigeraren i editeringsläge.

7 UTMANINGAR

7.1 Updatera vyer

När jag bestämde mig för att klassen QFileSystemModel som Qt erbjuder för att läsa filsystemet och för att ge mig en färdig implementation för data och information om filerna i filsystemet så hade jag ett stort projekt framför mig. Rojektet innebär att själv implementera ett system som läser filsystemet och förser vyerna med relevant data. Först började jag med en enkel modell som läste innehållet i mappen som användaren visade i en av vyerna och indexerade detta och försedde vyn med data om denna mapp. Jag märkte snabbt att detta inte var optimalt. Denna implementation fungerade tillräckligt bra för små mappar där det inte fanns mycket filer, men för stora mappar med flera tusen filer och undermappar i märkte jag snabbt ett problem: eftersom detta skedde i samma tråd som användargränssnittets händelseloop (på engelska eventloop) kördes updaterades inte användargränssnittet alls medan mappen laddades. Användargränssnittet reagerade inte på mushändelser eller annat innan mappen var laddad. På så vis upplevde användaren att programmet hade låst sig eller inte svarade.

Jag kom snabbt på att jag måste sköta detta parallellt i en annan tråd utanför den tråden där användargränssnittet körs och då meddela tråden var användargränssnittet körs genom Qt's signaleringsmekanism om ny data och på så vis undvika att användargränssnittet inte svarar under längre perioder.

Detta i sig för med sig en hel del problem eftersom trådarna delar samma minne och på så vis kan dela samma variabler. Ifall en tråd försöker ändra på en variabel samtidigt som en annan tråd försöker ändra på samma variabel slutar det oftast i att programmet avslutar onormalt (”krashar”) eller i andra odefinierade problem. Säker onormal avslutning händer åtminstone ifall en tråd deleterat någon variabel från datorns arbetsminne och någon tråd försöker komma åt denna variabelns minnesaddress. Efter mycket funderande över saken kom jag fram till att aldrig deletera variabler från minne från någon annan tråd än den som användargränssnittet körs i. Så när t.ex. användaren deleterat en fil ur filsystemet

och vyn behöver updateras så meddelar tråden där arbetet sker den tråd där användargränssnittet befinner sig i att denna data kan deleteras ur datorns arbetsminne. Det som användaren ser i vyerna är bara en representation av filsystemet, så varje fil som representeras i vyerna finns altså i datorns minne och när dessa inte längre behövs så måste de tömmas ur datorns arbetsminne så att de inte tar upp onödigt utrymme och på så vis skapar onödigt mycket minnesallokation till filhanteraren.

7.2 Filoperationer

Då användaren vill ta bort, flytta eller kopiera en fil vill man inte att programmet avslutas onormalt under filoperationen. Säkraste sättet att implementera detta är att flytta alla filoperationer till en helt separat process, så ifall processen där filhanteraren körs krashar så tar den inte den processen där filoperationen körs med sig. Jag valde att göra att när användaren vill göra en filoperation så startar filhanteraren en helt separat process var detta arbete sköts. Denna process har en skild tråd där arbetet sköts. Användargränssnittet är en dialogruta. Användaren kan här pausa arbetet, eller annullera arbetet, det finns även en kryssruta för om dialogen skall stänga sig själv efter arbetet är färdigt. Ifall det uppstår ett fel och arbetet inte kan fortsätta får användaren en till dialogruta med felmeddelande.

8 SLUTLEDNING

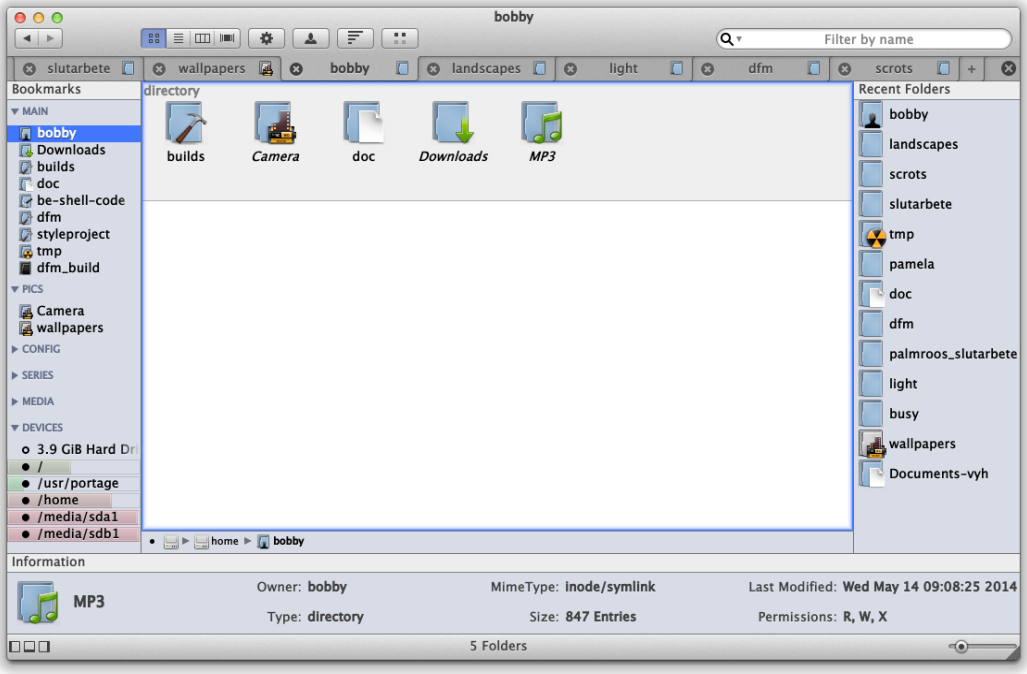
Syftet med lärdomsprovet var att jag skulle lära mig programmera i C++ / Qt och på så vis få till stånd en filhanterare. Jag har fått ihop kod till en filhanterare som klarar av de vardagligaste filoperationerna som en vanlig användare kan tänkas behöva. Filhanteraren har ännu flera problemområden som ännu skulle behöva förbättring, men i det stora hela gör den det som en filhanterare förväntas göra.

Koden för filhanteraren lever på sourceforge.net (http://sourceforge.net/projects/dfilemanager/), där kan man ladda ner en packad fil med alla källkodsfiler som behövs för att kompilera filhanteraren eller också kan man synka åt sig en git repositorie som man lätt kan synka när det kommit ändringar till koden. Jag fortsätter fortfarande att utveckla filhanteraren så koden kommer att ändras med tiden.

Utvecklingen av filhanteraren har fortlöpt under flera år då jag suttit periodvis och kodat mer och periodvis mindre, beroende på hur jag haft tid samt inspiration. Under utvecklingen har jag haft kontakt med vissa anonyma personer över internet som visat intresse för filhanteraren, som klonat min git repositorie och börjat använda filhanteraren. Dessa individer har inte bara reporterat buggar och andra problem med filhanteraren utan jag har även fått relativt mycket förfrågningar om funktionalitetsutökningar i filhanteraren. Så under åren har mer och mer funktionalitet blivit tilläggd i filhanteraren och mer kommer att läggas till. Dock hade jag alltid haft som mål att skapa mig en filhanterare, inte att skapa åt någon annan, men var givetvis glad när någon annan visade intresse.

Under tiden jag programmerat filhanteraren har jag lärt mig mycket om hur det är att programmera med C++ och Qt, och nu förstår jag varför Qt är så populärt som det är. Qt är väldigt omfattande och är mycket mer än bara ett grafiskt ramverk som har mycket underliggande funktionalitet. En av Qt's styrka är dokumentationen. Qt är väldigt bra dokumenterat vilket gör det mycket lättare för en ny person att börja programmera med Qt. Det är även lätt att bygga på Qt med sin egen funktionalitet.

Innan jag började med filhanteraren hade jag bara lite modifierat någon annans kod i deras programm för att bättre få dem att passa mina egna behov. Så först när jag började med filhanteraren var det svårt att komma igång och veta hur man skulle tänka för att få flödet i filhanteraren att fortlöpa.



Figur 11. Bild av filhanteraren.

KÄLLOR

Böcker

[1] Molkentin, Daniel (2007). Book of Qt 4 : The Art of Building Qt Applications. San Fransisco. No Starch Press Incorporated.

Elektroniska publikationer

[2] History of C++. The C++ Resources Network. Hänvisat 31.10.2013 http://www.cplusplus.com/info/history/

[3] A brief description. The C++ Resources Network. Hänvisat 31.10.2013 http://www.cplusplus.com/info/description/

[4] Introduction to Object-Oriented Programming Using C++. Hänvisat 01.11.2013 http://www.desy.de/gna/html/cc/Tutorial/tutorial.htm

[5] Qt Project, Qt 4.8 dokumentation. Hänvisat 13.11.2013 https://qt-project.org/doc/qt-4.8/

[6] Qt Project, Qt 5.1 dokumentation. Hänvisat 14.11.2013 https://qt-project.org/doc/qt-5.1/qtdoc/index.html

[7] C++ Language Tutorial, The C++ Resources Network. Hänvisat 25.11.2013 http://www.cplusplus.com/doc/tutorial/

[8] Multi-Threaded Programming With POSIX Threads. Hänvisat 05.05.2014

http://www.cpp-home.com/tutorials/128_1.htm

[9] Process and Program. Hänvisat 05.05.2014

http://whatis.thedifferencebetween.com/compare/process-and-program/