Avdelning för datavetenskap

Andreas Svensson
och
Björn Olsson

Winpaz

Ett GUI till en ny komprimeringsalgoritm

Winpaz

A GUI for a new compression algorithm

Examensarbete 10 poäng
DAI

Datum:	07-06-05
Handledare:	Robin Staxhammar
Examinator:	Martin Blom
Löpnummer:	C2007:06

Karlstads universitet 651 88 Karlstad
Tfn 054-700 10 00 Fax 054-700 14 60
Information@kau.se www.kau.se

Winpaz

Andreas Svensson och Björn Olsson

Denna rapport är skriven som en del av det arbete som krävs för att erhålla en kandidatexamen i datavetenskap. Allt material i denna rapport, vilket inte är vårt eget, har blivit tydligt identifierat och inget material är inkluderat som tidigare använts för erhållande av annan examen.

Andreas Svensson

Björn Olsson

Godkänd, 2007-06-05

Handledare: Robin Staxhammar

Examinator: Martin Blom

Sammanfattning

Detta är ett 10-poängs examensarbete på C-nivå, som vi under vårterminen avlagt vid Karlstads universitet. Målet med vårt projekt var att skapa ett nytt grafiskt gränssnitt till en komprimeringsalgoritm vid namn PAZ. Nuförtiden finns en hel uppsjö av grafiska gränssnitt till komprimeringsalgoritmer, men inget av dem stöder möjligheten att plugga in en egenutvecklad algoritm. Därför var vi tvungna att skapa vårt eget gränssnitt, som förutom PAZ också skulle stödja ZIP, RAR och andra etablerade algoritmer. Vi har alltså skapat ett grafiskt gränssnitt, vilket är anpassat för en implementation av den nya PAZ-algoritmen.

Algoritmen har utvecklats av vår uppdragsgivare Martin Larsson. Gränssnittet har vi, i samarbete med Martin, valt att kalla för Winpaz. Vi började med att göra två prototyper, som vi bland annat delade ut till personer på universitetet för att få respons på användarvänligheten hos gränssnitten. Med utgångspunkt från resultaten av dessa tester, konstruerade vi sedan vårt slutgiltiga gränssnitt.

Kravet vi hade på oss var att skapa ett användarvänligt grafiskt gränssnitt, som skulle stödja de vanligaste algoritmerna. Vi nådde målet att implementera stöd för algoritmerna ZIP och TAR. Dock fick vi göra avkall på implementationen av algoritmerna PAZ och RAR på grund av tidsbrist, men anpassade gränssnittet så att en senare inpluggning av dessa algoritmer är möjlig.

Vi är nöjda med vår slutprodukt, men inser också att det troligen krävs en del ytterligare funktionalitet för att gränssnittet ska fungera exemplariskt. Under projektet har vi, förutom att vi stärkte våra kunskaper i C++-programmering, även lärt oss att använda utvecklingsverktyget wx-DevCpp. Utöver detta har vi lärt oss att programmera grafiska gränssnitt med hjälp av wxWidgets.

Abstract

This is a bachelor´s project that we have been working on during a period of ten weeks, full time. The goal of our project was to design and implement a GUI for a new data compression algorithm called PAZ. At present, a vast array of compression/extraction GUIs is available, but none of them provide the possibility to incorporate a user developed algorithm. Thus, we had to create our own, with support for not only PAZ, but also ZIP, RAR, and other well known archiving algorithms. And so, we have created a GUI that is well suited for an implementation of the new PAZ algorithm.

The PAZ algorithm has been developed by Martin Larsson. We chose, in collaboration with Martin, to name the GUI application Winpaz. We began by implementing two separate prototypes, which we sent out to be tested by a closed group of beta testers. The reason for this test was to investigate how to design various parts of the application to be user friendly. Using the results from the testers, we then began the development of our final version of the GUI.

Our goals were to implement a user friendly GUI, that supported PAZ as well as the most widespread algorithms already in use. We achieved our first goal, to design a user friendly GUI, and we implemented support for both ZIP and TAR, but had to abandon our efforts in implementing RAR and PAZ support due to lack of time. The interface is however designed with the future incorporation of these algorithms in mind.

We are fairly pleased with our work, but we also recognize the need for added functionality in order to make the GUI a commercial grade product. During this project we have, apart from broadening our knowledge and skill in C++ programming, also learned to use the IDE wxDevCpp, a powerful open source tool for developing GUI applications based on the wxWidgets framework.

Innehållsförteckning

1. Inledning

1.1. Bakgrund

Vår uppdragsgivare, Martin Larsson som driver projektet Winpaz, har bett oss att designa och implementera en prototyp till ett grafiskt användargränssnitt till en komprimeringsalgoritm kallad paz. Martins förhoppning är att paz ska bli den nya standarden för komprimering av filer. Användargränssnittet ska vara intuitivt och användarvänligt. För att främja användbarheten, bör applikationen även stödja andra komprimeringsstandarder, åtminstone för extrahering. Att göra ett gränssnitt för enbart paz kan göra att programmet får begränsad spridning bland allmänheten, eftersom det idag finns flera andra komprimeringsalgoritmer med fast förankring bland användarna. Därför anser både vi och vår uppdragsgivare att vår komprimeringsapplikation också bör stödja de flesta redan etablerade algoritmerna på marknaden.

1.2. Mål

Målet med det här projektet är att designa och implementera ett grafiskt användargränssnitt för datakomprimering, som stödjer paz och alla de vanligaste komprimerings- och arkiveringsstandarderna på marknaden. Följande format ska stödjas:

1. paz
2. zip
3. rar
4. tar.gz

Applikationen ska vara lättanvänd och intuitiv, men också estetiskt tilltalande. Den ska även finnas för flera plattformar, exempelvis Windows, MacOSX och Linux. Drag-n’-Drop, alltså möjligheten att ta tag i filer med musen och dra dem dit man vill ha dem, ska implementeras, och knappar ska vara tydliga och ha beskrivande bilder. Det ska vara lätt att länka in nya algoritmer till programmet, och denna länkning ska kräva små eller inga ändringar i existerande programkod. Då tiden för projektet är begränsad, bör de mål vi satte upp ovan delas upp i delmål, som kan uppnås ett i taget:

1. Börja med en version för Windows.
2. Designa huvudfönstret och placera ut knappar och textfält.

3. Implementera komprimering med zip- och tar-algoritmerna. Dock är komprimering med rar-algoritmen licensbelagd, så det väntar vi med.
4. Implementera extrahering med ovanstående algoritmer.
5. Implementera Drag-n'-Drop-funktionalitet.
6. Implementera en Mac-version.
7. Implementera en Linux-version.

Ett sekundärt mål är att också implementera en enkel version av paz-algoritmen för testsyften. Detta kommer att genomföras om tid finnes när gränssnittet är implementerat.

1.3. Avgränsningar

Vi kommer inte att implementera splittning av arkiv under det här projektet. Det skulle ta för lång tid. En meny som kommer upp om man högerklickar på filer på exempelvis skrivbordet, där man kan välja hur och var man vill komprimera eller extrahera filen ifråga, skulle kräva att vi, förutom programmet självt, måsta skriva en installationsapplikation till programmet. Det finns det inte tid för, så det kommer inte heller implementeras.

1.4. Uppsatsens upplägg

Eftersom vår applikation ska utföra komprimering, går vi i kapitel 2 igenom olika former av komprimering, förlustbaserad (lossy) och förlustfri (lossless). Vi ger olika exempel på hur man använder komprimering, och när det kan vara praktiskt att kunna komprimera sina filer.
Då vi inte har utvecklat grafiska användargränssnitt tidigare, går vi i kapitel 3 djupare in på vad som är karakteristiskt för en grafisk applikations uppbyggnad. Vi diskuterar hur så kallad händelsedriven programmering fungerar, och på vilket sätt det möjliggör grafiska gränssnitt. Därefter fortsätter vi kapitlet om grafiska gränssnitt med att ställa upp ett antal regler för hur ett bra och användarvänligt gränssnitt bör utformas. I sista avsnittet av kapitel 3 förklarar vi närmare hur ramverket wxWidgets är konstruerat, och hur man kan använda det för att implementera grafiska applikationer. Vi tar också upp wxWidgets historia och lite av dess framtid, samt vilka för- och nackdelar vi identifierat för wxWidgets.
I kapitel 4 ställer vi upp en kravspecifikation för uppdraget. Vi delar upp kravspecifikationen i två delar. Den första delen beskriver de krav vi tillsammans med vår uppdragsgivare ställt upp

för det grafiska gränssnittet. Del två tar upp andra krav på applikationen, exempelvis vilka komprimeringsstandarder som ska stödjas i prototypen.
I kapitel 5 beskriver vi de två testversioner vi utvecklade för att undersöka olika grafiska komponenter och tekniska lösningar. Vi diskuterar också kring resultatet av våra tester, och vilka slutsatser vi drog utifrån den återkoppling vi fick från testpersonerna.
Med utgångspunkt i slutsatserna från kapitel 5, tar kapitel 6 upp vår konstruktionslösning. Vi förklarar hur vi har delat upp vår applikation i moduler, och hur dessa moduler relaterar till varandra. Vi beskriver vad de olika programkomponenterna gör, och förtydligar diskussionen med figurer över beroenden mellan modulerna, och bilder på de grafiska delarna av lösningen.
Det sjunde kapitlet beskriver implementationen av applikationen. Avsnitt 1 ger en översiktlig bild av hur det grafiska gränssnittet implementerats, och vilka lösningar vi valt för olika delar av användarmiljön. Det andra avsnittet ger en beskrivning av de klasser vi har skapat för olika ändamål, exempelvis arkiveringsgränssnittet. Till sist sammanfattar vi hur implementationen har gått, med avseende på vilka problem vi fick och hur vi löste dem.
I kapitel 8 undersöker vi resultatet av projektet, och diskuterar om vi har lyckats nå våra mål. Utifrån denna diskussion drar vi slutsatser och ger rekommendationer till vad man bör tänka på när man genomför liknande projekt.
Uppsatsens sista kapitel är en summering av vårt arbete. Vi identifierar vad vi lyckats bra med, och vad vi lyckades mindre bra med. Dessutom diskuterar vi vilka problem vi stött på under arbetet, och om vi förvärvat några nya kunskaper under projektets gång.

2. Komprimering

Varför behövs komprimering? I dagens samhälle tas större och större hårddiskar fram, och det som en gång tog en stor del av hårddiskens utrymme, utgör idag inte alls så stor procentuell plats, som det brukade göra. Är då inte komprimering ett avslutat kapitel nu när vi har utvecklat större hårddiskar? Borde det inte endast vara de äldre hårddiskarna, som behöver funktionalitet för att kunna komprimera? Nej. I och med att utvecklingen går framåt så får vi, för att ta ett exempel, filmfiler med högre kvalitet och upplösning än tidigare, vilket leder till större filer. Nya program tas ständigt fram och dessa har fler, bättre och mer avancerade funktioner än sina föregångare, vilket leder till att de filer de producerar innehåller en större mängd data i form av information, som måste sparas. Mer data ger större filer, vilka tar mer plats. För att återgå till exemplet med filmfiler, så är det populärt att krympa DVD-filmer till avi-filer (DivX och Xvid). Dessa ligger på ca: 700 MB och är ganska mycket mindre än de stora DVD-filmerna, som ligger på allt från 4,7 GB (som är DVD5-formatet) upp till 8,5 GB (som är DVD9-formatet, vilket använder sig av dubbla lager). Idag har den nya högupplösta teknologin redan börjat ta fart. Dessa format är Sonys BluRay och Samsungs HD-DVD. Dessa skivor ser precis ut som vanliga CD- och DVD-skivor, men istället för att använda den vanliga röda laser, som används till CD och DVD, så har man nu tagit fram en blå laser, som tack vare att den har kortare våglängd än den röda, får plats med mer information per skiva. Blålaserformaten rymmer upp till 50 GB per skiva. Detta är enorma mängder data som kan krympas till storlekar som kan lagras och skickas på nätet med hjälp av komprimering.

Två tillfällen då komprimering passar utmärkt är dels då man vill lagra en eller flera stora filer på en hårddisk, kanske för en lång tid framöver och som då inte ska ta för stor plats på hårddisken, dels då man vill skicka en eller flera filer över ett nätverk. Då får man till att börja med en enda fil som allt ligger i, till skillnad från att behöva skicka allt från 10 till 10 000-tals filer självständigt, och denna är sedan också mindre till storleken. Detta är ett argument för att komprimering i högsta grad är användbart.

2.1. Komprimering i allmänhet

Komprimering är ett sätt att reducera storleken på en viss datafil. Det finns dock inte en entydig lösning på problemet, utan det finns mängder av olika sätt att komprimera filer på. Till att börja med kan man dela in komprimering i två huvudklasser, *ickedestruktiv* och *destruktiv* komprimering (lossless and lossy compression).[4]

När man tillämpar icke-destruktiv komprimering, minskar man ner storleken på en fil utan att det går någon information förlorad. Man kan alltså komprimera en fil och sedan extrahera (eller packa upp) filen igen utan att den har ändrats på något vis, indatafilen är exakt lik utdatafilen efter de båda stegen. Man kan alltså komprimera och packa upp en fil hur många gånger som helst utan att det påverkar den ursprungliga filens information. Exempel på format som är icke-destruktiva är t.ex. ZIP, TAR och RAR, vilka också är andra arkiveringsformat. Om du däremot applicerar destruktiv komprimering (eller förstörande komprimering) på din fil, så kommer filen att ha ändrats efter komprimeringen för att ta mindre plats och denna ändring av filinformationen kan inte återskapas utan är i och med den destruktiva komprimeringen borta. När du sedan extraherat utdatafilen igen, så kommer du inte att ha samma fil som indatafilen. Destruktiv komprimering används vid exempelvis bild-, film- och ljudkomprimering. Bildformatet JPEG är ett exempel på destruktiv komprimering. Likaså formaten MP3 (för ljud) och MPEG (för film). Här kan vi alltså inte återskapa den ursrungliga filen efter att vi har komprimerat den, eftersom vi för varje gång vi komprimerar filen tappar lite information. Därför blir kvalitén sämre och sämre för varje gång man sparar om en JPEG-bild eller ett MP3-ljudspår. Destruktiv komprimering av filer används nästan helt uteslutande då det är människor, som är slutanvändare. En människa har en begränsad uppfattningsförmåga och om vissa toner i ett ljudspår (eller färger i en bild/film), som är mindre vesäntliga tas bort, kommer människan förmodligen inte att höra (eller se) någon skillnad. Om man däremot använder en för hög kompressionsgrad, kan människan uppfatta detta som att ljudet blir lite ”metalliskt” och defekt (eller som att bilden/filmen får en annan färgnyans än vad man förväntat sig). Ibland kan man dock ha överseende med detta, då storleken reduceras kraftigt. Om man har en exekverbar fil, kan man inte komprimera den destruktivt, eftersom ett enda bitfel kan göra filen icke körbar. Men eftersom vi vill att de filer vi komprimerar ska återskapas exakt till det skick de var i innan vi komprimerade dem, så kommer vi enbart att använda icke-destruktiv komprimering.

3. Grafiska användargränssnitt

Eftersom uppdragsgivaren vill ha ett grafiskt användargränssnitt till sin algoritm, och vi aldrig sysslat med grafisk programmering tidigare, började vi med att undersöka några viktiga delar i utformningen av ett sådant. Avsnitt 3.1. förklarar närmare hur händelsedriven programmering fungerar, och varför det lämpar sig väl för grafiska användargränssnitt. Därefter undersöker vi i avsnitt 3.2 hur man designar ett bra användargränssnitt, vilka regler man bör följa, samt hur man undviker fallgropar. I sista delen av kapitlet, avsnitt 3.3, tittar vi närmare på hur ramverket wxWidgets fungerar, och hur det är uppbyggt.

3.1. Händelsedriven programmering

Innan de grafiska användargränssnitten gjorde sitt intåg på marknaden, dominerade s.k. *batch-programmering*. Batch-programmering innebär att applikationens flöde är definierat direkt i källkoden. Här nedan följer ett exempel i pseudokod på ett enkelt program skrivet med ett batchflöde:

```
a[0] = GetNumberFromKeyboard();
a[1] = GetNumberFromKeyboard();
Print(a[0] + a[1]);
```

Programmet läser in två tal, och skriver ut dem. Det kommer att stanna vid rad 1 och rad 2 för att vänta på att användaren skriver in sina tal. Batchprogrammering är dock inte särskilt väl lämpat för grafiska gränssnitt och interaktiva applikationer. Ponera att vi har ett program som antingen adderar tal, som exemplet ovan, eller som skriver ut två bokstäver. Pseudokoden kan se ut så här:

```
integer Choice = GetNumberFromKeyboard();
if(Choice == 0)
{
	a[0] = GetNumberFromKeyboard();
	a[1] = GetNumberFromKeyboard();
	Print(a[0] + a[1]);
}
if(Choice == 1)
{
	a[0] = GetCharacterFromKeyboard();
	a[1] = GetCharacterFromKeyboard();
	Print(a[0] + a[1]);
}
```

Det märks redan nu att antalet instruktioner växer snabbt när vi bygger ut funktionaliteten. Dessutom tvingas användaren välja i början av programmet om denne vill arbeta med tal eller bokstäver, och är sedan låst vid det valet resten av exekveringen.
Betrakta följande pseudokod:

```
integer i = 0;
Repeat
{
	OnNumberEntered(number)
	{
		a[i] = number;
		i = i + 1;
	}
	if(i == 2)
	{
		print(a[0] + a[1]);
	}
}
```

Programmet gör samma sak som den första versionen av batch-programmet, men är skrivet enligt ett annat paradigm. Detta är ett *händelsedrivet (eng. event driven) program*. Med händelsedrivet menas att programmet väntar på att en händelse ska inträffa, och utför olika instruktioner beroende på vad den inträffade händelsen var. I det här fallet går programmet i en loop tills användaren skriver in ett tal, då den utför den första satsen. När denna sats körts två gånger, kommer även den andra satsen att köras.
Vid första anblicken kan det här programmet verka krångligare än batch-versionen, men den andra varianten kan avsevärt mycket lättare generaliseras och byggas ut med annan funktionalitet. Vi skulle exempelvis kunna lägga till kod för att hantera våra bokstäver från det andra batch-exemplet. Det kan se ut så här:

```
int i = 0
int j = 0
Repeat
{
	OnKeyPress(key)
	{
		if(isNumber(key))
		{
			a[i] = key
			i = i + 1
		}
		if(isChar(key))
		{
			b[j] = key
```

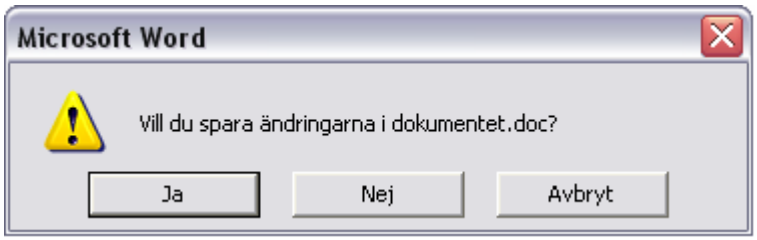
```
                j = j + 1
            }
        }
        if(i == 2)
        {
            print(a[0] + a[1])
            i = 0
        }
        if(j == 2)
        {
            print(b[0] + b[1])
            j = 0
        }
    }
```

För ett så här enkelt exempel blir den händelsedrivna koden omständligare än batch-koden, men har som sagt fördelen att den lätt byggs ut med ytterligare funktionalitet.
Loopen som utgör programmet kallas för en *händelseloop* (eng *event loop*), och koden som hanterar händelser kallas *händelsehanterare* (eng. *event handlers*). Händelseloopen kan vi utöka med ytterligare hanterare för andra händelser, exempelvis musklick, skapandet av en fil, signaler från hårdvara, o s v. När en händelse inträffar innan den föregående händelsen är omhändertagen, läggs den i en *händelsekö* (eng. *event queue*), där den sparas tills det finns tid att ta hand om den. Denna programmeringsstil passar utmärkt för grafiska användargränssnitt, som bör kunna hantera ett stort antal olika händelser, såsom tryck på olika knappar, musklick och annat, som kan hända i obestämd ordning.

3.2. GUI-design i allmänhet

Det finns ett stort antal uppsättningar av regler och riktlinjer att följa när det gäller utformningen av en applikation. Dessa samlingar är uppsatta av många olika författare och är därmed inte alltid överensstämmande med varandra.
För vårt projekt har vi valt att fokusera på de riktlinjer som sätts upp i en rapport vi namn ”*Grafiska gränssnitt - vad har användbarheten för betydelse?*”, skriven vid Göteborgs universitet, av Camilla Emring och Anna Funke [2]. I rapporten diskuterar de hur grafiska gränssnitt bör utformas för att främja användbarheten och minimera risken för fel. Författarna kommer fram till ett antal punkter att beakta vid design av grafiska gränssnitt, och de vi ansåg var viktigast för vår applikation listas här nedan: [2]

- *Feedback:* En applikation bör alltid ge feedback till användaren om vad som pågår, och inom rimliga tidsramar. Det får inte gå för lång tid mellan en utförd handling och feedback på densamma. Generellt gäller dessutom att om en åtgärd tar under en sekund, behövs ingen feedback, om inte åtgärden medfört någon större förändring av systemet. Om åtgärden exempelvis innebär att man komprimerar ett antal filer, bör någon form av feedback ges, så att användaren får reda på om komprimeringen lyckades eller inte. Är åtgärden istället inläsning av filer till arkivet, räcker det med att listan av filer uppdateras för att användaren ska förstå vad som hänt.
  Tar en åtgärd längre än tio sekunder krävs det mer än ett meddelande när det är klart. En förloppsindikator kan vara en bra lösning.[3]
- *Konsekvens och standard:* En applikation bör vara konsekvent utformad, och placering av knappar ska helst följa någon form av standardisering. Om ångerknappen ligger till höger om OK-knappen i de tre första dialogrutorna användaren träffar på, bör inte det mönstret ändras i den fjärde dialogrutan. Utöver detta kan man tänka på att försöka följa de standarder som gäller generellt för applikationer. De flesta program har exempelvis ett kryss i övre högra hörnet för att avsluta programmet, och en meny i överkant på fönstret. Menyn brukar dessutom ofta innehålla rubriker som ”Arkiv”, ”Redigera” och ”Hjälp”. Följer man denna konvention, inser troligtvis en presumtiv användare var de hittar en viss funktion.
- *Återhämtning från misstag:* Fel kommer alltid att inträffa, så en robust mekanism för att låta användare återhämta sig från dem är ett måste. Ett bra felmeddelande följer dessa fyra grundregler:
  1. De är skrivna på användarens språk, och undviker koder.
  2. De ska beskriva felet så precist som möjligt, och inte bara konstatera att något är fel.
  3. De bör beskriva en eller flera lösningar på problemet.
  4. De ska inte förnedra användaren om denne gjort fel, utan vara vänligt och konstruktivt utformade.
- *Förhindra misstag:* Det är alltid bättre att förhindra misstag, än att återhämta sig från dem i efterhand. Därför är det bra att ha bekräftelserutor, så att användaren får chans att ångra sig innan en åtgärd utförs.
  Bekräftelserutan här nedan kommer upp när man försöker stänga Microsoft Word utan att först ha sparat sitt dokument.



Figur 3.1: Bekräftelserutan som visas i MS Word vid stängning av ett osparat dokument

- *Skärmdisposition:* Under utvecklingen av en applikation bör man fundera över vilken information användaren kan behöva vid varje enskild åtgärd, och se till att den finns synlig, eller i alla fall tydligt visa var denna information finns att tillgå. Vidare bör man dela upp den information som visas på arbetsytan, i naturliga komponenter med de informationsbitar som hör ihop. Som exempel kan nämnas att knappar för att lägga till och ta bort filer ur en lista, bör ligga i närheten av representationen av denna lista. Användaren kommer att se dessa komponenter som enheter, och referera till dem som sådana.

  Det gäller också att se till att avstånden mellan komponenter som ska användas i sekvens är korta, så att rörelsen blir rent motoriskt lättare att utföra. Man ska exempelvis inte lägga en komponent i nedre högra hörnet, och nästföljande komponent i arbetsflödet i det övre vänstra hörnet.
- *Färgteman, ljud och typsnitt:* Antalet färger, ljud och typsnitt måste hållas nere, annars blir användaren snabbt ofokuserad och får svårt att koncentrera sig. Dessutom kan programmet bli rörigt, och svårt att navigera i. Dock kan intelligent användning av färger och ljud hjälpa användaren på traven, och dra uppmärksamheten till rätt komponent.

Vi kommer att ha dessa riktlinjer i åtanke när vi utformar vår applikation, och följa dem så gott det går.

3.3. GUI-programmering med wxWidgets

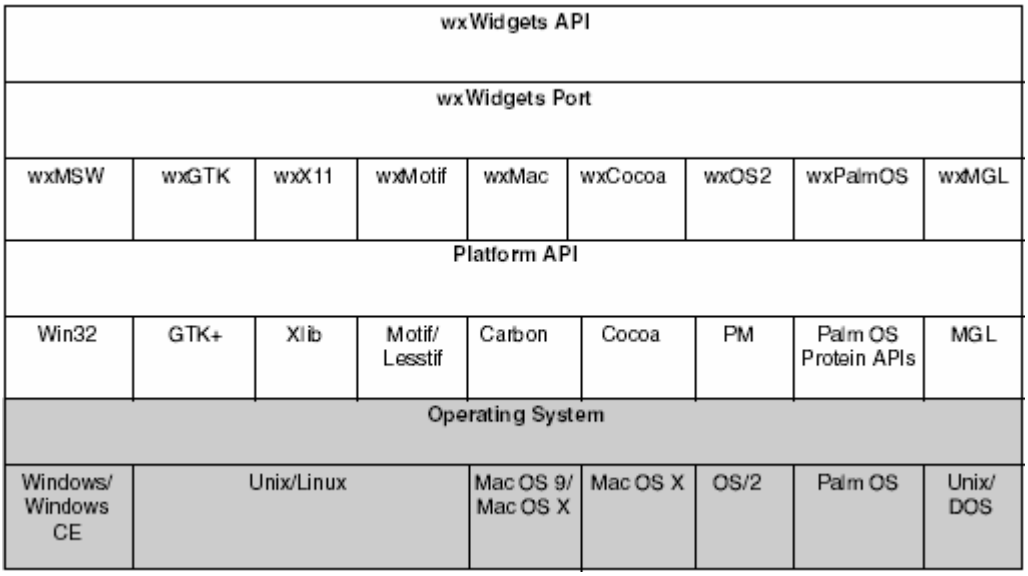
3.3.1. Vad är wxWidgets? [1]

wxWidgets är ett ramverk för att skriva portningsbara applikationer med grafiska användargränssnitt. Det sköter det grundläggande arbetet med hur applikationens grafiska komponenter ritas upp och beter sig. wxWidgets tillhandahåller ett stort antal klasser, varav de flesta är grafiska komponenter, s.k. *widgets*. Dessa kan programmeraren använda och ställa in på

egen hand. En grafisk applikation består typiskt av ett huvudfönster, med ett antal widgets i form av knappar, listor, textfält, et c., samt kanske några bilder och annan grafik. wxWidgets gör skapandet och inställningen av dessa enklare, genom att tillhandahålla speciella klasser för knappar och andra s.k. *widgets*. För att göra wxWidgets-applikationer lättare att porta till andra plattformar (även bärbara sådana, som exempelvis Palm pilots), finns inte bara grafiska klasser definierade, utan också stränghantering, process-till-process-kommunikation, filhantering, multithreading och mycket annat.

3.3.2. En kort historiebeskrivning [1]

wxWidgets-projeket startades 1992 vid universitetet i Edinburgh, av Julian Smart. Han höll på att utveckla ett verktyg kallat Hardy, för att skapa och redigera diagram. Julian ville inte välja mellan att implementera sitt program på antingen Sun-arbetsstationer eller PC:ar, så han bestämde sig för att använda ett ramverk som stödde flera plattformar. Eftersom utbudet av sådana ramverk var begränsat, och institutionens budget ändå inte tillät inköp av ett, återstod att skriva ett eget. I september samma år var wxWidgets version 1.0 klar, och laddades upp på institutionens FTP-server, varpå andra programmerare började utnyttja koden. Version 1.0 stödde XView och MFC 1.0, men Borland C++-användare klagade på MFC-kravet. Följaktligen skrevs wxWidgets om så att det enbart använde Win32-API:t. Eftersom Motif höll på att ta över efter XView, utvecklades snabbt stöd för just Motif. Bidrag från användare runt om i världen strömmade in, och stöd för ett flertal andra plattformar lades till. År 1997 kom en GTK-variant, och året efter kom också en Mac OS-version. Med det nya millenniet kom wxUniversal, ett widget-bibliotek för plattformar som inte hade något eget sådant bibliotek.
År 2002 lades wxX11 till, och denna portning använde wxUniversal, vilket gjorde att den kunde användas i vilken UNIX-miljö som helst.
I juli 2003 blev Windows CE-varianten klar, och året efter fick wxWidgets sitt nuvarande namn. wxWidgets hette nämligen wxWindows innan 2004, och det gillade inte Microsoft, som hävdade att det var varumärkesintrång. Samma år kom också en rejäl ansiktslyftning av wxMac för OS X, samt en minimal portning till Palm OS 6. Version 2.6 av wxWidgets släpptes i april 2005, med uppdateringar för alla de större portningarna. Figuren här nedan visar den övergripande strukturen för wxWidgets 2.6, och dess förhållande till de olika plattformarnas API:er.



Figur 3.2: Översikt över wxWidgets placering i förhållande till varje plattforms API

I december 2006 släpptes version 2.8, innehållande mindre uppdateringar och korrektioner. I skrivande stund är 2.8.3 den senast släppta versionen. Nästa stora release är planerad till slutet av 2007, med version 3. Denna version kommer bland annat att ha bättre stöd för C++ STL-bibliotek.[5]

3.3.3. Fördelar med wxWidgets [1]

Under inledningsfasen av det här projektet hade vi flera ramverk och verktyg i åtanke. Bland dessa fanns Qt, som är utvecklat av det norska företaget Trolltech. Deras lösning bygger på att emulera alla widgets som behövs, istället för att som wxWidgets använda de som finns i exempelvis Win32 API. Qt har, i likhet med wxWidgets, också klasser för exempelvis networking och multithreading. Det som fick oss att välja bort Qt var priset; det kostar 5260 EUR per licens, och vi skulle behöva två, vilket blir ungefär 100 000 SEK. Detta var uteslutet, så vi letade vidare och hittade då wxWidgets. wxWidgets är gratis och har öppen källkod, vilket passade oss utmärkt. Att källkoden är öppen innebär också att det finns mycket resurser på Internet, och många användare lägger också ut lösningen på olika problem, kanske till och med det man för tillfället vill ha hjälp med. Dessutom finns inga patentproblem med i bilden, eftersom allt är öppet. wxWidgets användarvillkor gör också att vi inte heller behöver göra vår egen källkod tillgänglig. Utöver Qt och wxWidgets finns säkert många fler plattformsoberoende ramverk, men dessa var de vi stötte på. Eftersom ett av delmålen var att porta applikationen till fler plattformar än Windows, ville vi undvika plattformsspecifika API:n. Därför valde vi wxWidgets.

3.3.4. Nackdelar med wxWidgets

Det finns inte bara fördelar med wxWidgets, utan också en del nackdelar. Hjälpfiler och annan dokumentation kan ibland vara för gammal eller till och med direkt felaktig. Detta irriterar en aning, eftersom det tar viktig tid från projektet att behöva testa funktioner och tolka källkod. wxWidgets har också en begränsad samling av grafiska komponenter. Som exempel kan nämnas att det finns en knapp som kan ha en bild, och en som kan ha text, men ingen som kan ha båda. Det går naturligtvis att kombinera olika klasser för att få önskat beteende, exempelvis kan man rita en bild på en knapp med text, och på så sätt uppnå sitt mål. Detta tar tyvärr tid från viktigare delar av projektet, men kan ibland vara nödvändigt.

4. Kravspecifikation

Kravspecifikationen i det här kapitlet är en sammanfattning av den kravspecifikation vi fått av vår uppdragsgivare Martin Larsson, och listar de viktigaste kraven på vår prototyp.
Avsnitt 4.1. beskriver kraven på den grafiska delen av vårt användargränssnitt, hur programmet ska se ut och hur filhantering ska gå till.
I avsnitt 4.2. specificerar vi mer programmässiga krav, som exempelvis vilka kompressionsstandarder som ska stödjas.

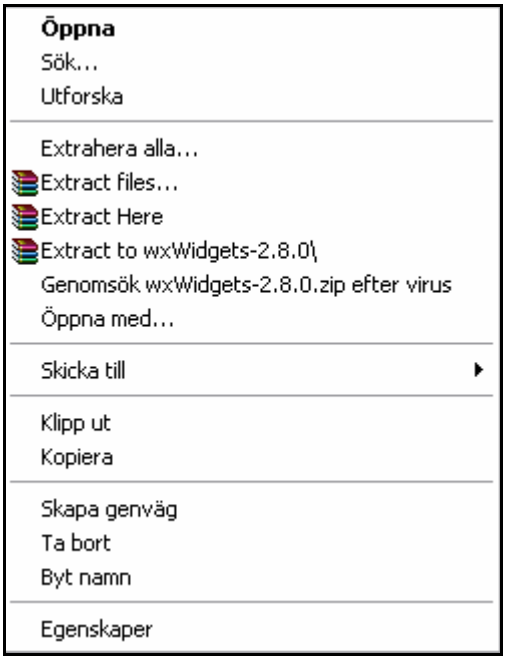
4.1. Krav på användargränssnittet

Ett av de viktigaste kraven på gränssnittet är att det ska vara så enkelt att använda, att det inte finns behov av att konsultera en hjälpfil. Det ska vara så uppenbart som möjligt hur programmet fungerar.
Filhantering ska fungera som det gör i andra applikationer under Windows, med standardiserade dialogrutor och visning av filer liknande den i Utforskaren.
Drag & Drop ska implementeras för att främja användbarheten, och underlätta för slutanvändare. De kan exempelvis ha en instans av fönstret uppe på skrivbordet hela tiden, och bara dra in filer i det när de behöver komprimera eller extrahera något.

4.2. Krav på applikationen i övrigt

Programmet ska i framtiden stödja de flesta algoritmer för komprimering och extrahering som finns på marknaden idag, men vår prototyp behöver enbart kunna stödja ZIP, och, för extrahering, RAR. Komprimering med RAR är licensbelagd, så det får vänta tills applikationen genererar pengar för att kunna betala dessa licenskostnader.
Applikationen bör också integreras i operativsystemet på samma sätt som WinRar och WinZip är. Om användaren högerklickar på en fil, kan denne få upp en lista med komprimerings- eller extraheringsalternativ för filen. Detta kan bli svårt att hinna med inom projektets tidsramar, och kommer troligtvis inte att implementeras i prototypen.

Figur 4.1: Popupmenyn vid högerklick på vanliga filer under Windows, med WinRAR installerat



Figur 4.2: Popupmenyn vid högerklick på arkiv under Windows, med WinRAR installerat

Slutligen ska, om tid finnes, en prototyp av PAZ-algoritmen implementeras. Detta är dock ett sekundärt mål, och huvudsyftet med projektet är att implementera det grafiska användargränssnittet.

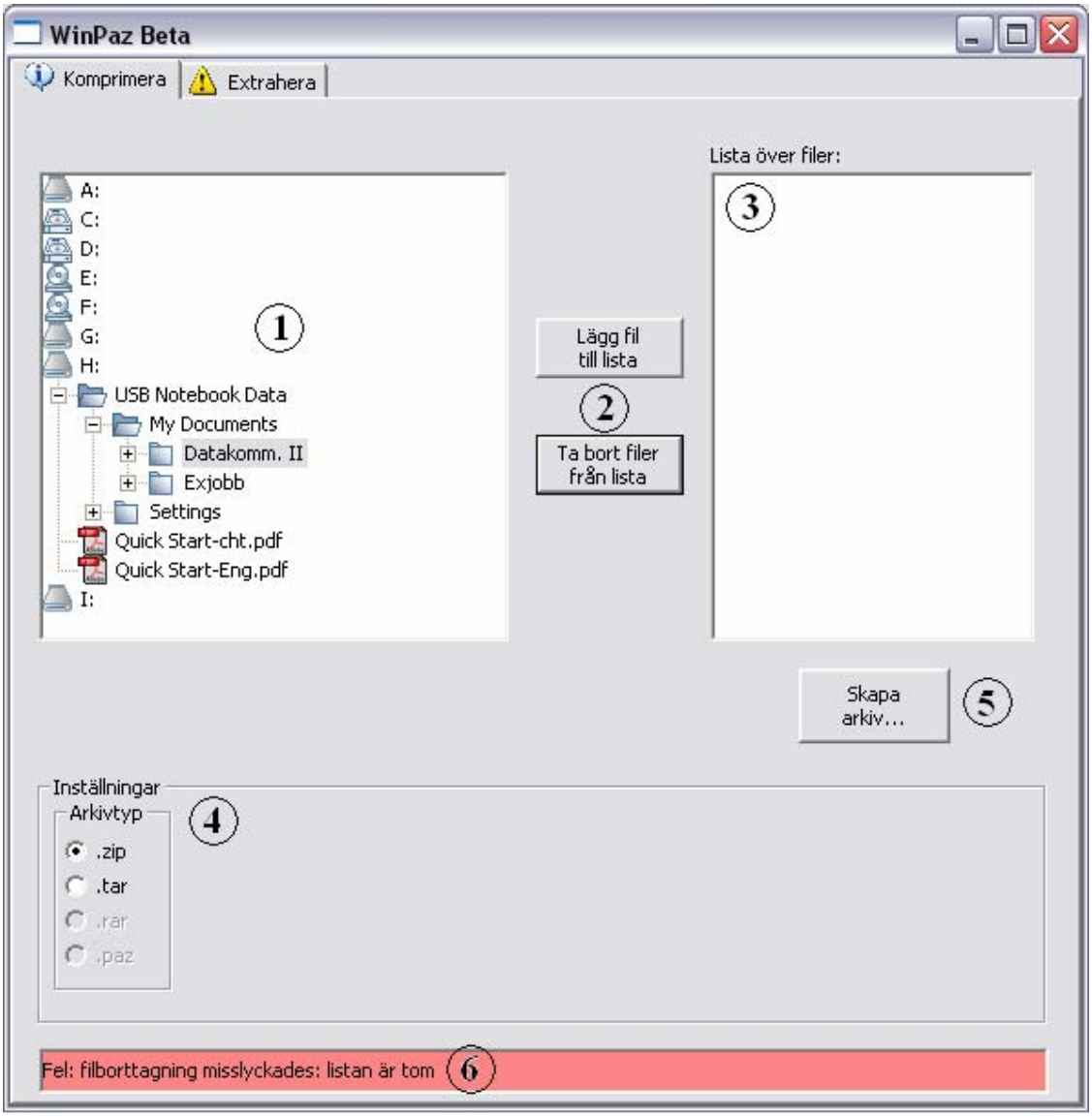
5. Tester

Innan vi började med den slutgiltiga prototypen gjorde vi två testversioner, där vi försökte implementera olika lösningar på diverse designproblem. Dessa två versioner skickades ut bland vänner och bekanta, samt bland lärare på avdelningen för datavetenskap på Karlstads universitet, för att få lite idéer och förslag på förbättringar. De som genomförde testerna av prototyperna var användare med god datorvana, och kunskap om vad komprimering är. Följande kapitel är en beskrivning av testversionerna och resultatet av undersökningen.

5.1. Beskrivning av version 1

I version 1 av programmet utnyttjar vi en widget kallad wxGenericDirCtrl för att hämta in filer för komprimering och extrahering. wxGenericDirCtrl är tyvärr behäftad med problem, varav det största är att den saknar stöd för flera markeringar samtidigt. Detta gör det jobbigt och icke-intuitivt att hämta in filer. Dock finns möjlighet att hämta det underliggande wxTreeCtrl-objektet, för vilken man eventuellt kan ställa in stöd för flera markeringar. Detta är något som kommer att undersökas närmare.

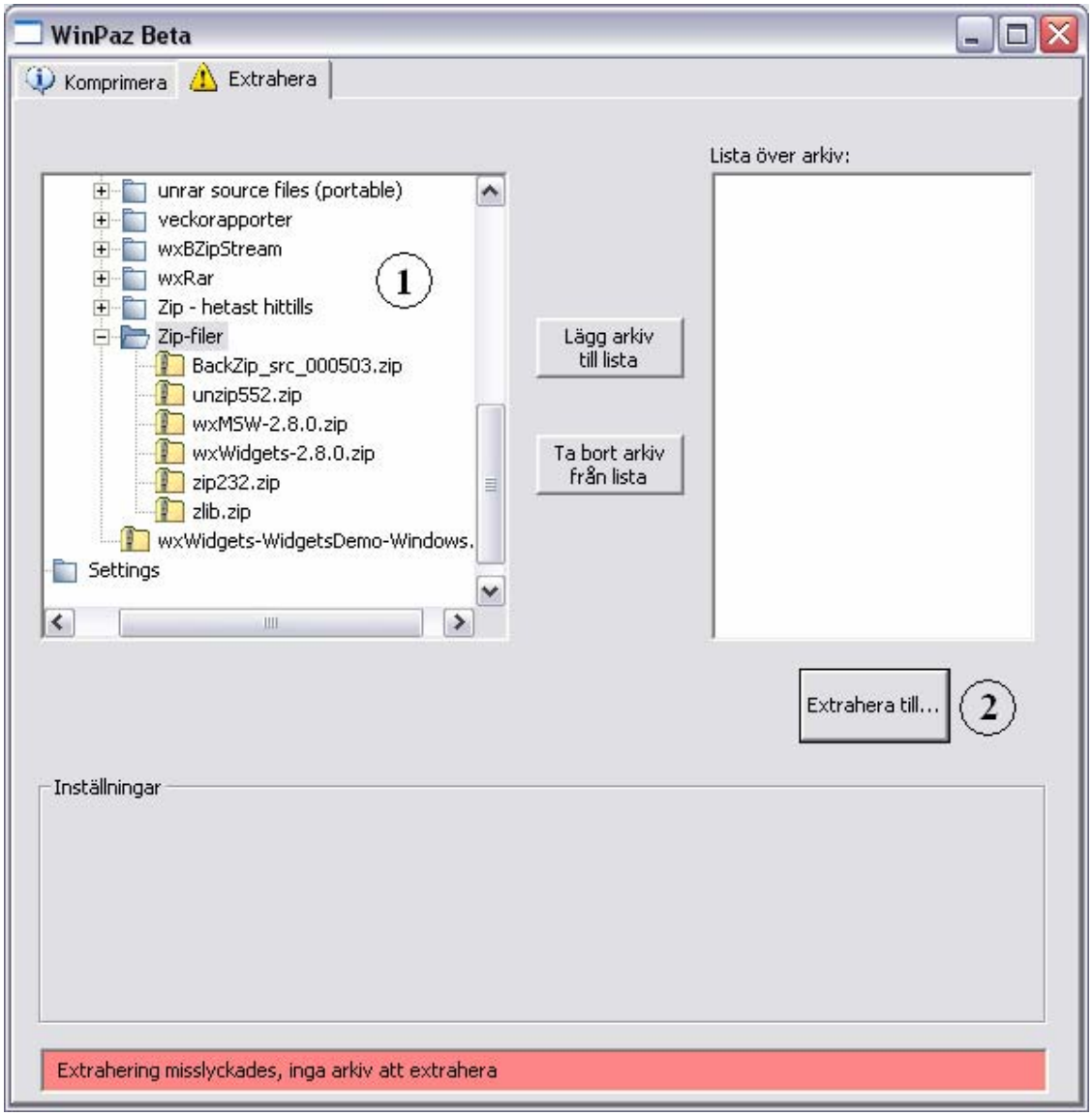
Här nedan visas två skärmdumpar från version 1, med tillhörande beskrivningar:



Figur 5.1: Testversion 1, Komprimera-fliken

1. *Filträdet:* Detta är wxGenericDirCtrl-objektet som diskuterades ovan. Här väljer användaren sina filer att komprimera, och lägger dem till listan till höger.
2. *Lägg till/Ta bort filer till/från lista:* Dessa knappar är användarens sätt att lägga filer till listan, och ta bort oönskade filer från densamma.
3. *Lista över filer:* Här listas alla filer som användaren läst in.
4. *Inställningar:* I denna ruta ska olika inställningar för komprimering kunna göras, men i testversionen kan endast en av två komprimeringsmetoder väljas, och inget annat.
5. *Skapa arkiv:* när användaren fyllt sin lista med de filer denne vill komprimera, trycker han eller hon på den här knappen. Då öppnas standarddialogrutan för att spara filer, och användaren får möjlighet att välja plats och namn för sin komprimerade fil. När valen gjorts, och användaren tryckt på ok, börjar komprimeringen.

6. *Meddelanden:* Här nere visas meddelanden om vad som händer i programmet, exempelvis om komprimeringen lyckades, eller om – som här – en åtgärd misslyckades. Raden blir gulgrön om åtgärden istället lyckas.



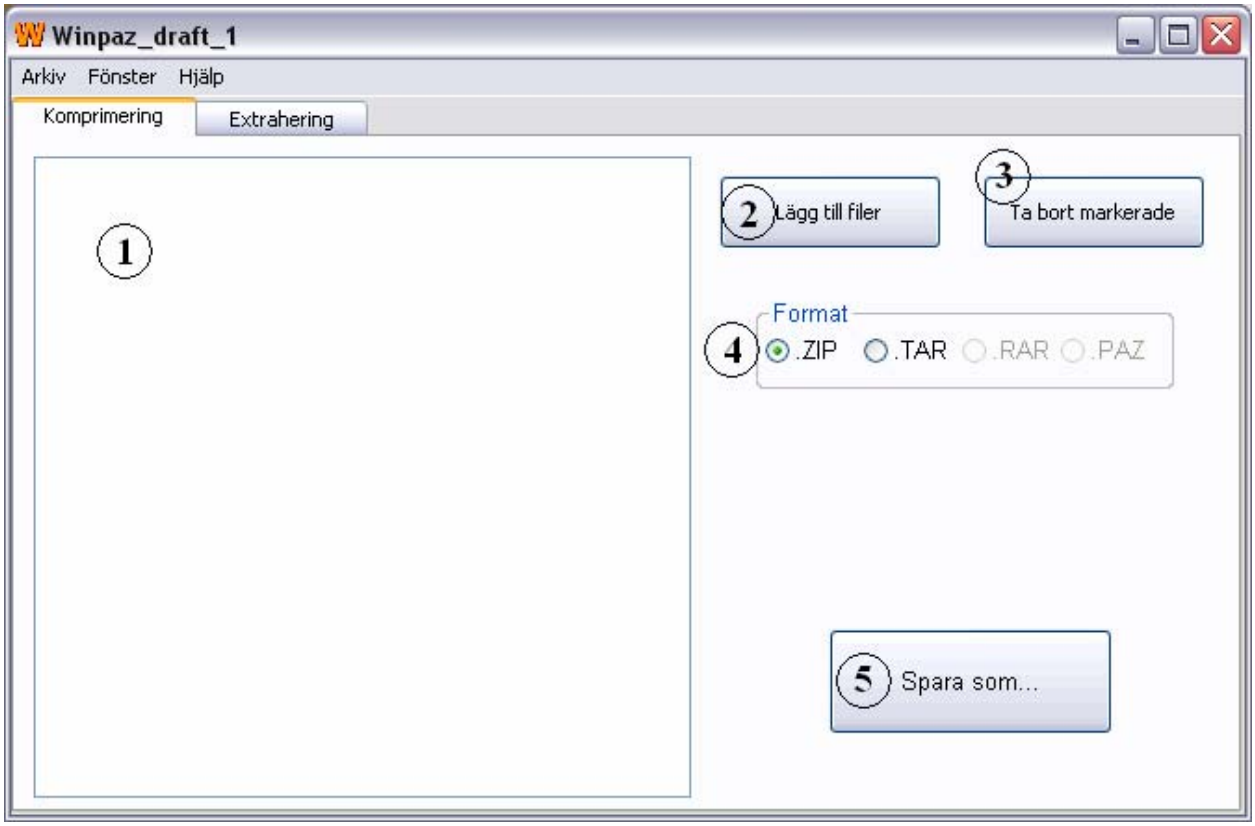
Figur 5.2: Testversion 1 – Extrahera-fliken

1. *Filträdet:* Trädet för filer att extrahera är låst till att enbart visa komprimerade filer av exempelvis zip- eller tar-typ.
2. *Extrahera till:* När användaren valt vilka arkiv som ska packas upp, trycker han eller hon här, och öppnar därmed en mappväljare, för att välja en målmapp, eller för att skapa en sådan.

I övrigt är de två flikarna lika.

5.2. Beskrivning av version 2

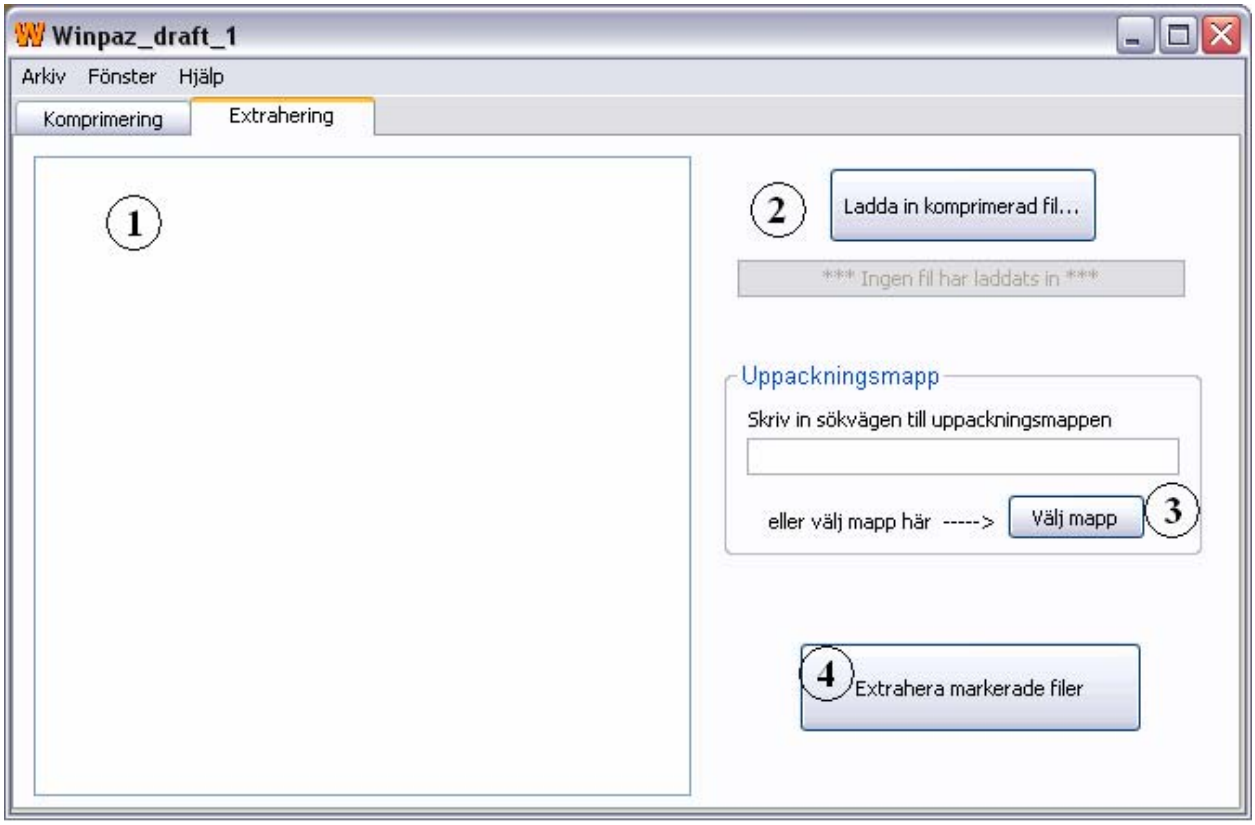
I den här versionen valde vi att gå ifrån filträdet från version 1, och istället satsa på standardiserade dialogrutor för mappväljare och för att spara arkiven. I denna version laddas filerna in och sparas via operativsystemets standardiserade dialogrutor. När ett antal filer har lästs in, så hamnar deras filnamn och filändelse i den vänstra listboxen. Här är två skärmdumpar från vår andra version, från respektive flik:



Figur 5.3: Testversion 2 – Komprimera-fliken

1. *Förhandsgranskningen:* När man har läst in en eller flera filer till programmet, hamnar dessa filers namn och filändelse i denna listbox.
2. *Lägg till filer:* Här läser man in filer via operativsystemets standardiserade dialogruta för att öppna filer. När man trycker på ”Öppna”, hamnar de valda filerna i ovan nämda lista.
3. *Ta bort markerade:* Med hjälp av denna knapp kan du ta bort de filer, som markerats i listan. Filerna tas bara bort ur listan, inte från datorn. Om inga filer markerats får man ett felmeddelande i form av en pop-up-ruta.
4. *Filformat:* Här väljs i vilket format man vill komprimera sina valda filer. Det förvalda alternativet är i denna version ”.zip”.

5. *Spara som:* Detta är den avslutande knappen, som skapar ett arkiv, med det av användaren inmatade namnet, innehållande de markerade filerna från listan. Om komprimeringen går bra, fås en bekräftelse i form av ett pop-up-meddelande. Dialogen som kommer upp när man trycker på knappen är en spara-fil-dialog, som endast visar upp mappar, och filer med ändelsen av det valda formatet. Alltså, väljer man att komprimera i formatet zip, så visas endast mappar och filer med ändelsen .zip upp.



Figur 5.4: Testversion 2 – Extrahera-fliken

1. *Listan av arkiv:* Här har vi en lista med de inladdade arkiven, som i denna version har låsts till ett enda arkiv. Då är denna lista onödig, men i framtida versioner är det tänkt att det inladdade arkivets innehåll ska placeras i listan och man ska kunna välja vad man vill extrahera ur det valda arkivet. Men hittills visas alltså bara arkivets namn och filändelse.
2. *Ladda in fil:* Denna knapp fungerar som inläsningen av filer på komprimeringsfliken, men med skillnaden att endast mappar och filer med ändelsen .zip och .tar visas, och att man bara kan läsa in en fil i taget.
3. *Målmappväljaren:* Här väljer man vart man vill extrahera sina filer. Man kan skriva in sökvägen till mappen, dit man vill extrahera filerna, direkt i textrutan. Men om man hellre vill söka upp mappen via ett standardiserat filträd, kan man trycka på knappen ”Välj mapp”.

4. *Extrahera:* Med denna knapp extraherar man det i listan valda arkivets innehåll till den valda målmappen.

5.3. Genomförande och testresultat

För att få en fingervisning om vilka funktioner användare ville ha i ett komprimeringsprogram, arrangerade vi ett litet test med våra två betaversioner. Försökspersoner i testet var vänner och bekanta till oss, samt ett antal lärare på avdelningen för datavetenskap vid Karlstads Universitet. Ambitionen med vårt försök var inte att få ett statistiskt underlag för vad som borde finnas i applikationen, utan att fastställa någon form av generell riktning för designen, exempelvis knappars placering, textfälts storlek eller programmets reaktioner på användarhändelser. För att strukturera testresultaten författade vi ett testformulär. Detta formulär innehöll följande sex frågor:

1. Vad tyckte du om navigeringen för att hämta in filer?
2. Hur bra var knappar och dialogrutor placerade, kändes det logiskt?
3. Var programmet rörigt, d.v.s. fanns det för många textfält, knappar, et c.?
4. Vad tyckte du om felmeddelandena, visades de på ett bra sätt?
5. Vad tyckte du om systemet med olika flikar för extrahering/komprimering?

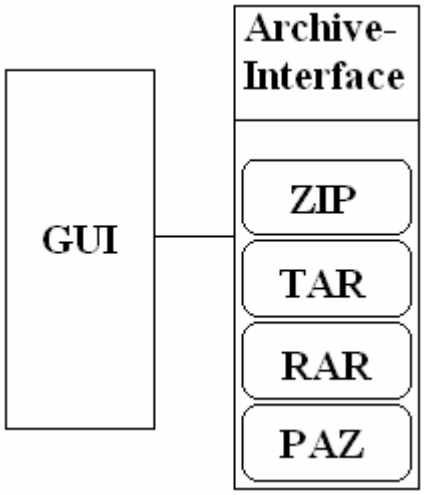
Dessa frågor besvarades med ett kryss i en av de fem rutor som fanns under varje fråga. Alternativen var: *Mycket bra*, *Bra*, *Ok*, *Sämre* och *Dåligt*. Sist i formuläret lade vi till en punkt för egna tankar och idéer om vad försökspersonen skulle vilja se i programmet.
De slutsatser vi dragit av svaren på de olika frågorna är:

1. Generellt sett tyckte försökspersonerna att filinläsning fungerade bra på båda sätten, alltså var både filträdet och standarddialogen rimliga och logiska. Den slutgiltiga prototypen kommer antagligen att implementera en kombination av de båda, men med tanke på problemen med trädets tekniska begränsningar, kan det bli svårt att lösa trädnavigeringen inom projektets tidsramar.
2. Knappar och textrutors placering och inbördes ordning var enligt försökspersonerna uteslutande bättre i version två, men kommentarerna under fråga 6 angående programmens utformning ger anledning till eftertanke. En av kommentarerna var att flödet på knapparna var bättre i version 2, de låg i en bättre och mer logisk ordning. Dock fick den stora tomma textrutan i nr 2 kritik, den kändes som ett slöseri med utrymme.

3. Se ovan.
4. Version 2 hade enligt testresultatet det bästa systemet för att visa felmeddelanden, nämligen meddelanderutor. Dock gillade några användare den andra varianten, med en remsa i nederkant som blev röd och visade ett felmeddelande om något gick fel. Den visar också ett meddelande om vad som händer i programmet under körning, och byter då färg till gulgrön. En kombination av de båda kanske blir bäst, med meddelanderutor vid större fel, som exempelvis korrupta arkiv, och en remsa med feedback om användarens åtgärder och mindre fel, som borttagning av filer som inte finns.
5. Flikuppdelningen var något som testpersonerna uteslutande tyckte om, så den lösningen kommer att implementeras också i slutversionen.

6. Konstruktionslösning

I det här kapitlet förklarar vi hur de olika delarna av programmet relaterar till varandra, och hur vi valt att konstruera vårt grafiska gränssnitt. Vi har försökt dela upp applikationen i naturliga moduler, och har kommit fram till en uppdelning enligt figuren nedan:



Figur 6.1: Översikt över programmodulerna i applikationen

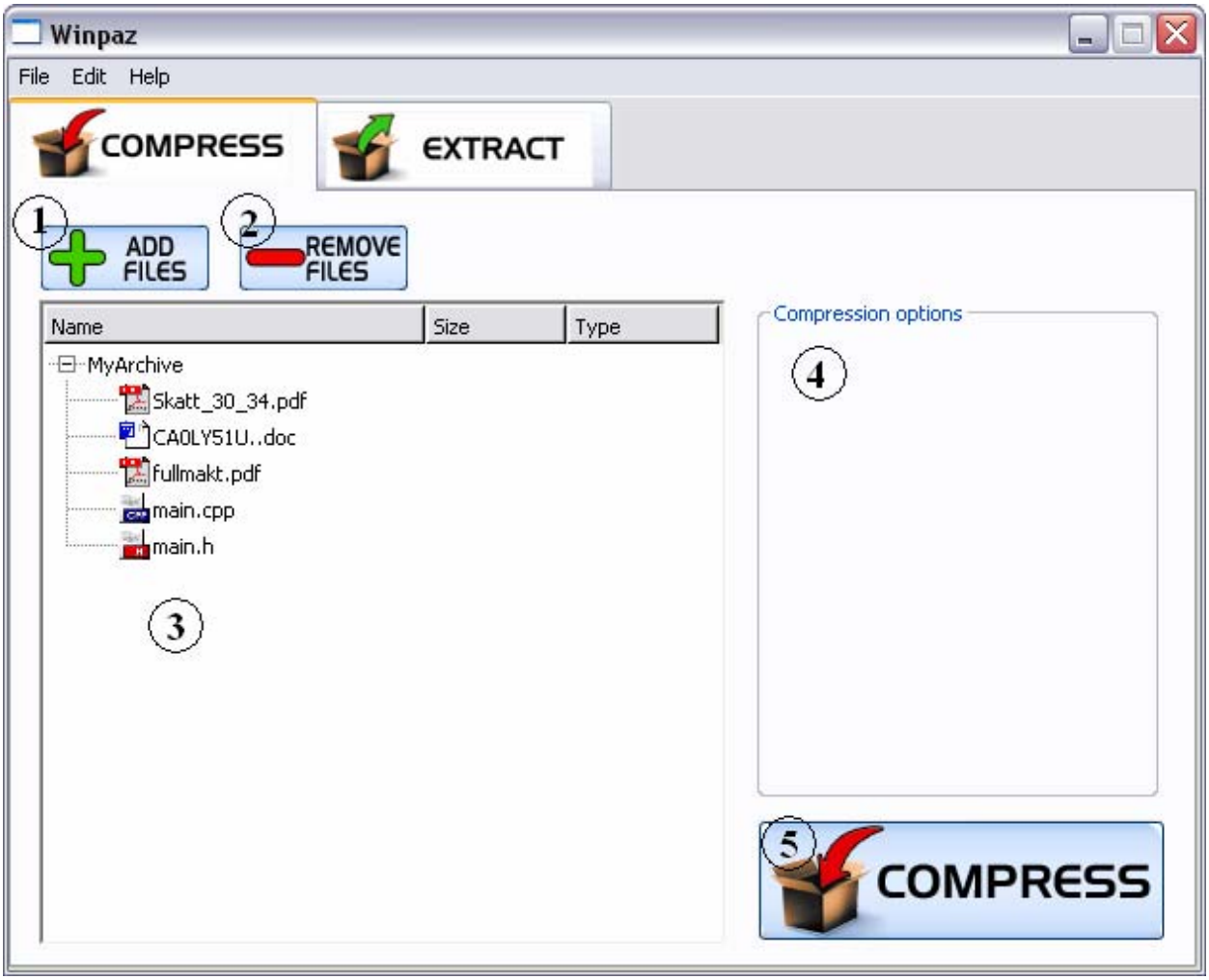
6.1. Programmatisk konstruktionslösning

Eftersom wxWidgetsbiblioteket är konstruerat efter det objektorienterade paradigmet, är också vår lösning objektorienterad. Detta ger oss även en naturlig uppdelning av vår applikation i moduler, vilket gör att det blir förhållandevis lätt att lägga till nya komprimerings- och extraheringsalgoritmer. Det blir också lättare att underhålla koden.

6.2. GUI-konstruktionslösning

Applikationen är uppdelad i två huvudfönster, som hanterar extrahering och komprimering oberoende av varandra. Denna uppdelning var något som fungerade bra i våra testversioner, och därför finns med i vår konstruktionslösning.

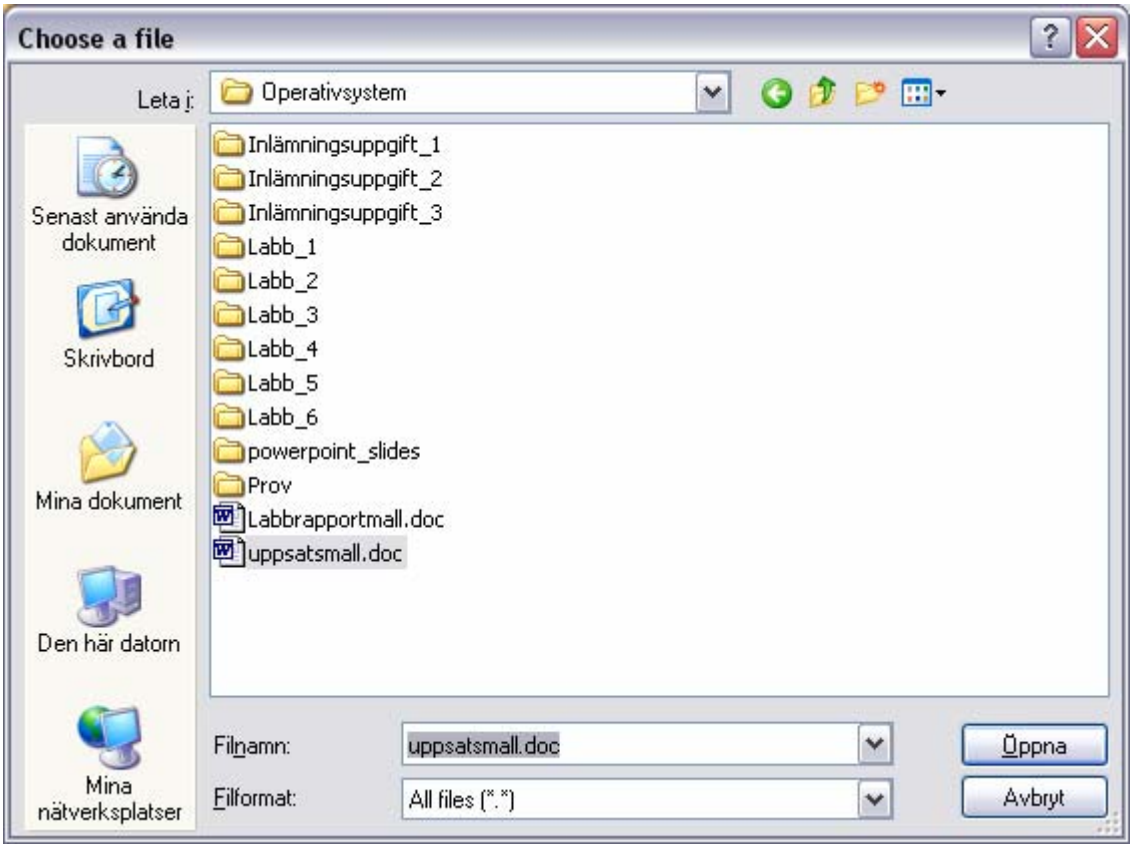
6.2.1. Compress-fliken



Figur 6.2: Winpaz – Compress-fliken

1. *Add files-knappen:* Den här knappen öppnar en standarddialogruta för inläsning av filer. När användaren valt sina filer och tryckt på *öppna*, läggs alla filerna till i filträdet.
2. *Remove-knappen:* Trycker användaren här, tas alla filer som är markerade i filträdet bort.
3. *Filträdet:* Det här är filträdet, där information om alla filer kommer att visas när användaren läst in dem.
4. *Compression options:* Här är det tänkt att diverse valmöjligheter ska presenteras för användaren, exempelvis kompressionsgrad. Dock blir den här versionen av programmet endast en prototyp, så denna ruta är för tillfället tom.
5. *Compress-knappen:* Trycker användaren här, öppnas en standarddialogruta för att spara filer, och användaren får välja namn, målmapp och arkivtyp för sitt arkiv. När alla val bekräftats, komprimeras alla filer i trädet.

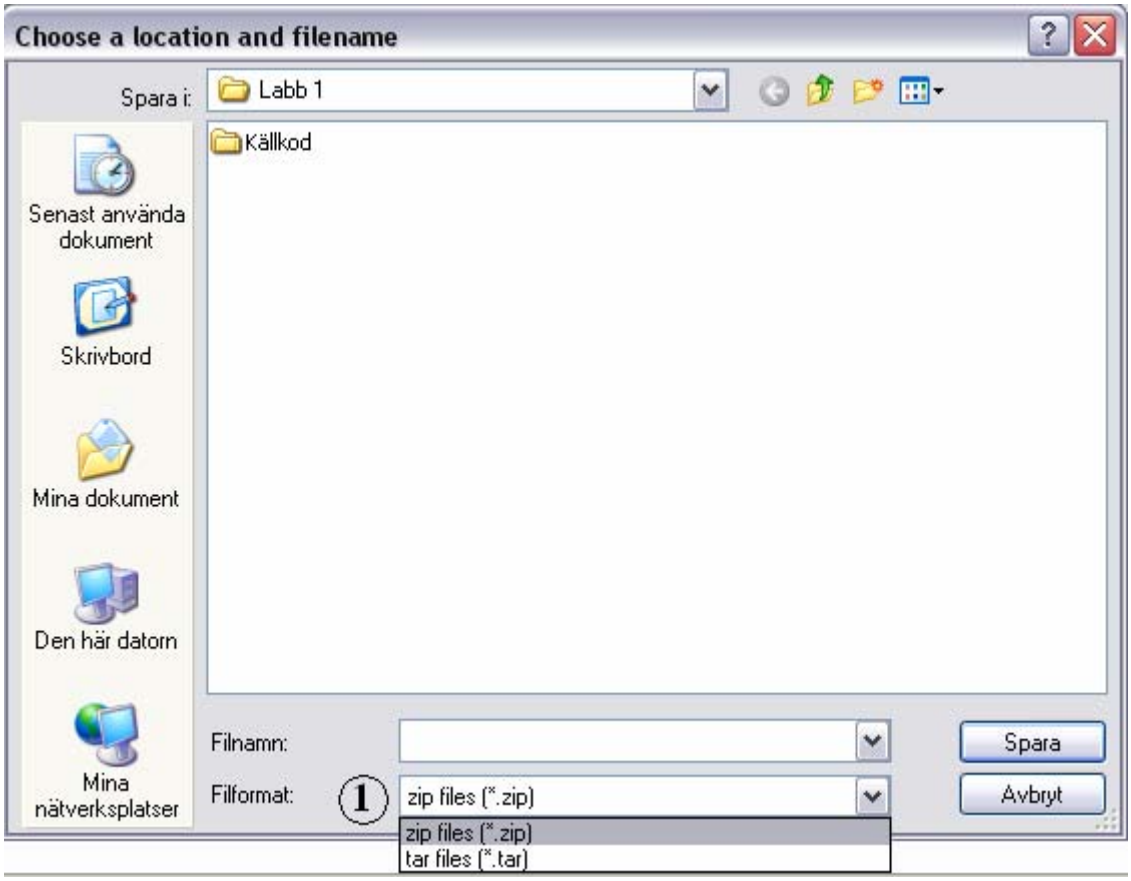
6.2.2. Add files-dialogen



Figur 6.3: Add files-dialogen

Det här är en standarddialogruta för öppning och inläsning av filer. Här kan användaren välja vilka filer denne vill komprimera. När valet bekräftas med Öppna, läggs alla filer till filträdet.

6.2.3. Compress-dialogen

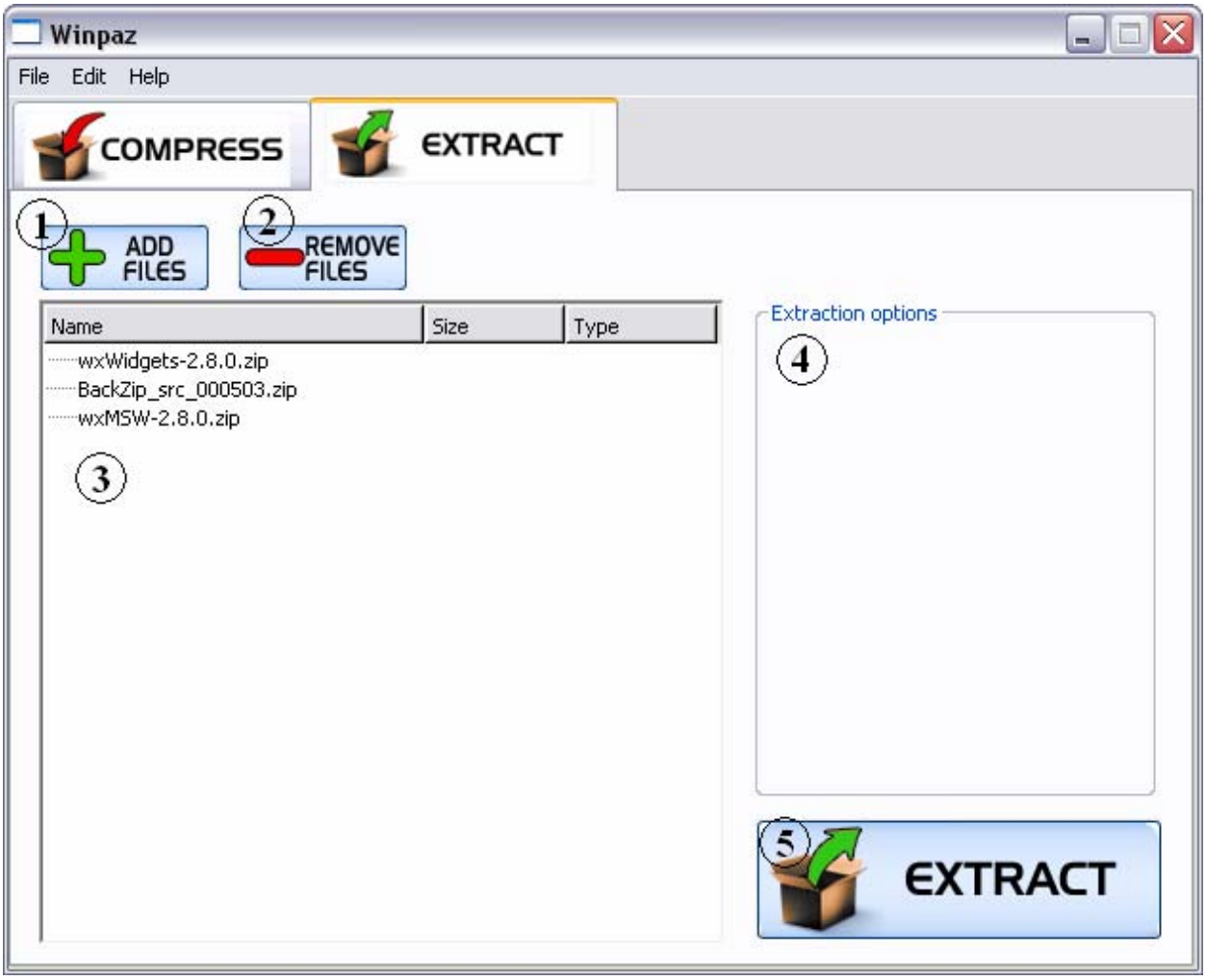


Figur 6.4: Compress-dialogen

Denna dialogruta låter användaren välja målmapp, namn och arkivtyp för sitt arkiv.

1. Detta är en lista av så kallade *wild cards*, som avgör vilka filer som visas i dialogrutan. Den här listan av wild cards är låst till de två filtyperna ZIP och TAR. (Eftersom vi endast implementerat stöd för dessa två komprimeringsalgoritmer i vår prototyp, så har vi nöjt oss med dessa två wild cards. Men om ytterligare algoritmer implementeras, exempelvis PAZ, så utökas givetvis listan.)

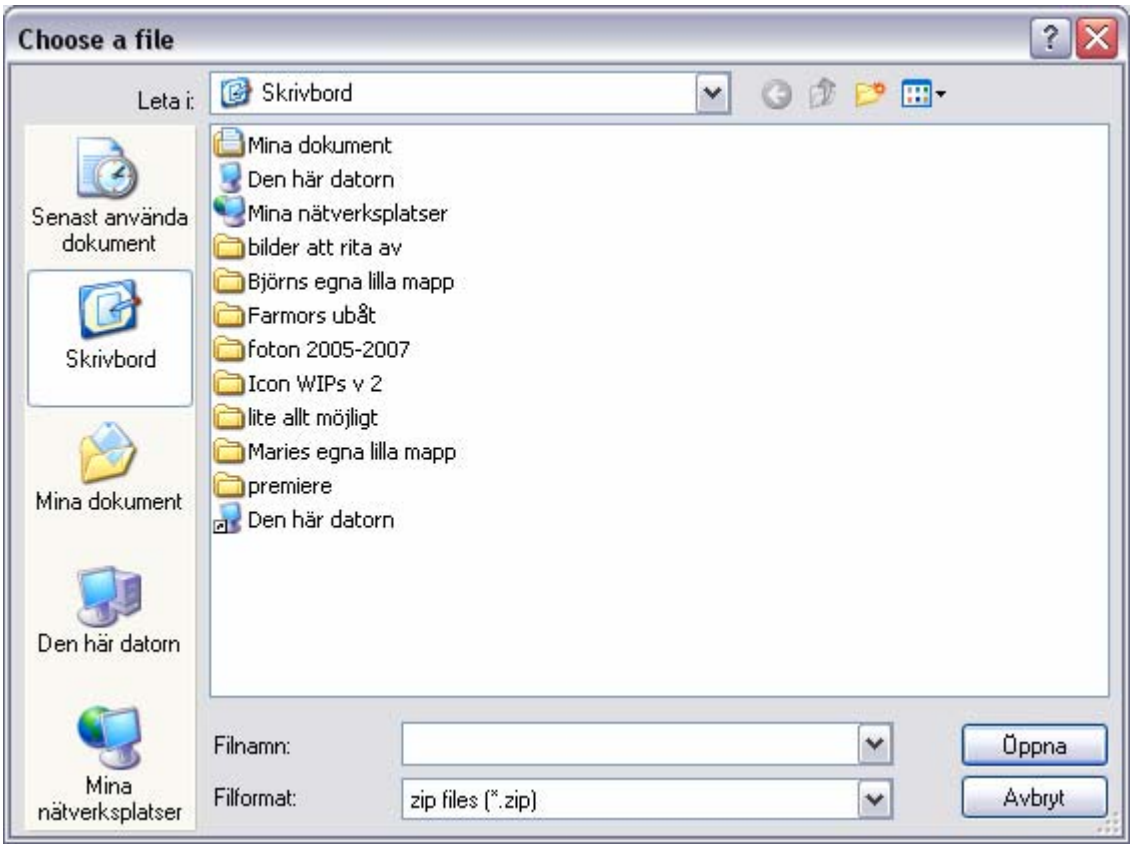
6.2.4. Extractfliken



Figur 6.5: Winpaz – Extract-fliken

1. *Add archives-knappen:* Här kan användaren läsa in arkiv till listan över arkiv, med hjälp av standarddialogrutan för inläsning av filer.
2. *Remove archives:* Den här knappen tar bort de arkiv i arkivträdet som är markerade.
3. *Arkivträdet:* Här visas information om de arkiv användaren har läst in.
4. *Extraction options:* Här är det tänkt att diverse valmöjligheter ska presenteras för användaren, exempelvis om man vill bevara mapphierarki eller inte. Dock blir den här versionen av programmet endast en prototyp, så denna ruta är för tillfället tom.
5. *Extract.knappen:* När användaren läst in alla de arkiv denne vill extrahera, och trycker på den här knappen, öppnas en standarddialogruta för att välja målmapp.

6.2.5. Add archives-dialogen

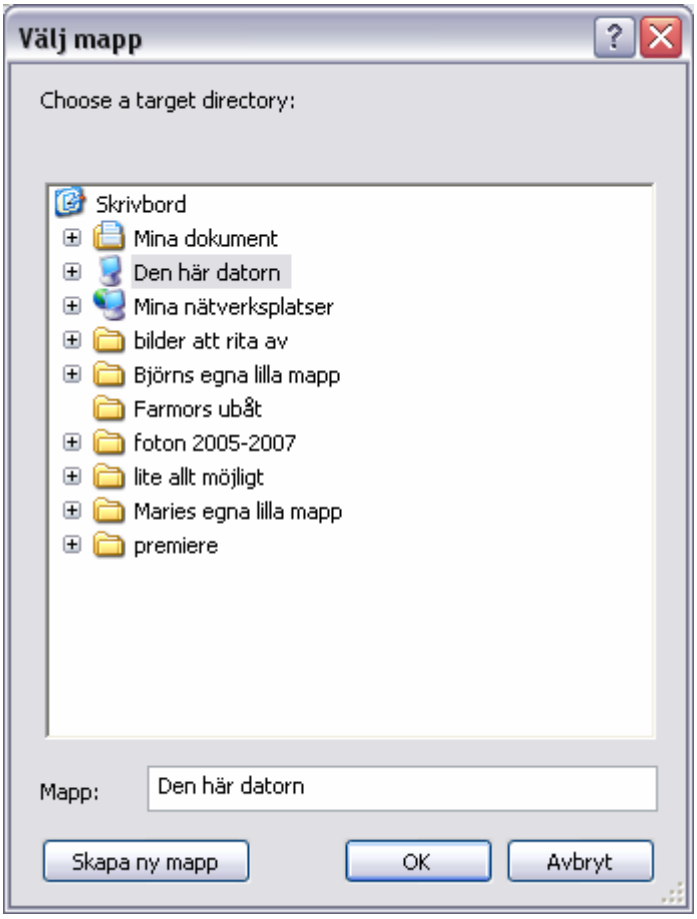


Figur 6.6: Add archives-dialogen

Om användaren har tryckt på Add archives-knappen i huvudfönstret, öppnas denna dialogruta, där man kan välja vilka arkiv man vill lägga till i arkivträdet.

1. Den här listan av wild cards är låst till de två filtyperna ZIP och TAR. (Även här gäller att vi skulle utöka listan av wild cards vid ytterligare algoritmimplementationer.)

6.2.6. Extract-dialogen

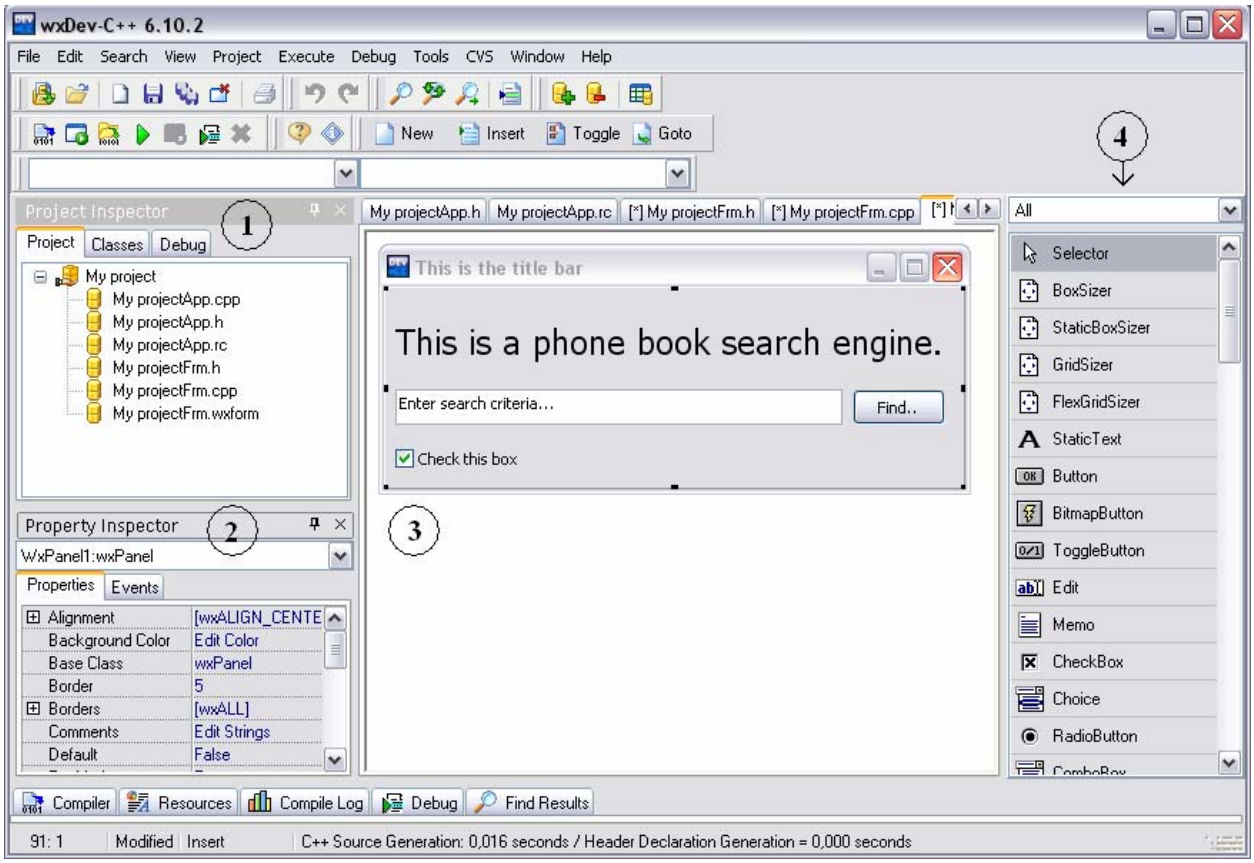


Figur 6.7: Extract-dialogen

Det här är standarddialogrutan för att välja målmapp. Den öppnas om användaren tryckt på Extract-knappen i huvudfönstret, och här kan man välja målmapp, eller skapa helt nya mappar.

7. Implementation

Vi har utvecklat vår applikation med ett verktyg kallat wxDev-cpp. wxDev-cpp är en utökning av utvecklingsmiljön Dev-cpp, som är ett utmärkt, och gratis, verktyg för implementation av applikationer i C++. Skillnaden mellan Dev-cpp och wxDev-cpp är att den senare också har en dialog-editor baserad på wxWidgets inbyggd, och ett antal mallar för att snabbt få grundläggande funktionalitet i en wxWidgets-applikation. Den mall vi använt genererar kod för ett huvudfönster, med systemmeny i övre vänstra hörnet, och stäng-, minimera- och maximeraknappar i det övre högra hörnet. I detta enkla fönster kan man sedan placera sina widgets, som man väljer från en lista i dialog-editorn.



Figur 7.1: Översikt över wxDev-Cpp

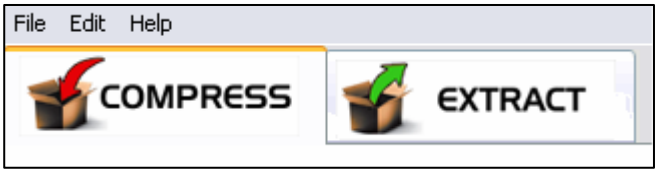
1. *Project inspector:* Här kan man undersöka vilka filer man har i sitt projekt, och vilka klasser som finns definierade. Man kan också skapa virtuella mappar här, för att gruppera sina filer bättre.
2. *Property inspector:* Det är här man ställer in sina widgets med avseende på exempelvis utseende, stil, färg, typsnitt på text, och annat viktigt.
3. *Workspace:* Här skapar man sina fönster, som alla widgets ska hamna i.

4. *Widget selector:* Detta är en lista med alla tillgängliga widgets, standarddialogrutor och textkontroller. Man väljer en widget ur listan, och klickar och drar ut den i fönstret.

Avsnitt 7.1. ger en översiktlig beskrivning av det grafiska gränssnittet. I avsnitt 7.2 går vi igenom vilka klasser vi använt, och beskriver syfte och alla metoder för de klasser vi definierat själva. Det sista avsnittet, 7.3., är en kort sammanfattning av hur implementationen har förlöpt.

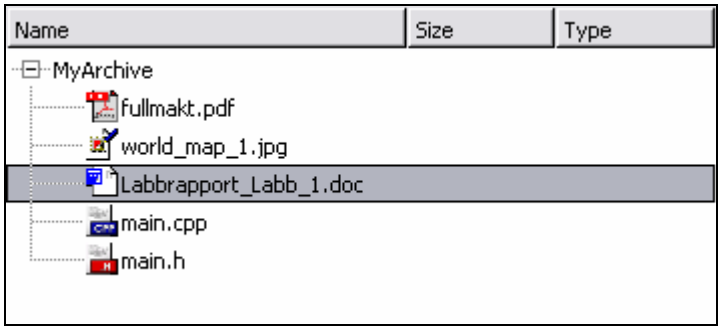
7.1. Översiktlig implementationsbeskrivning av det grafiska gränssnittet

Efter sammanställning av resultatet från vår testsession (se kapitel 5) bestämde vi oss för att dela upp vår applikation i två huvudfönster, Extract och Compress. Detta realiserades med en instans av en så kallad wxNotebook. Denna widget tillhandahåller funktionalitet för att ordna ett antal paneler i ett fliksystem.



Figur 7.1: Fliksystemet i Winpaz

Listorna över de filer och/eller arkiv man läst in har implementerats med en wxTreeListCtrl, vilket egentligen är en widget för att visa noder i en trädstruktur, fast med funktionalitet för kolumner. wxTreeListCtrl är fortfarande i utvecklingsstadiet, och har varit behäftad med en del problem. De flesta av dessa har vi dock lyckats lösa, så den nuvarande versionen är tillräckligt felfri för den funktionalitet vi behöver i prototypen.



Figur 7.2: Filträdet i Winpaz

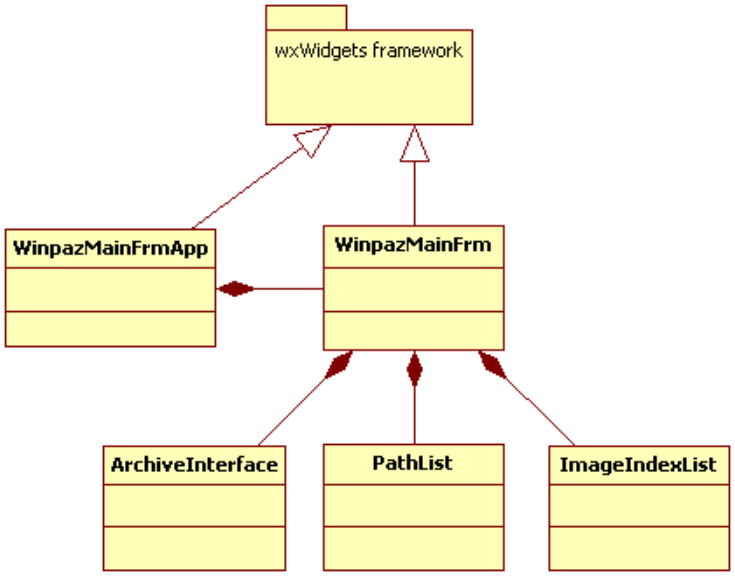
För att minimera mängden kodarbete har vi använt fördefinierade standarddialogrutor för inläsning av filer, sparning av arkiv och val av målmapp för extrahering. Detta medför dessutom att användarna kommer att känna igen sig i dessa dialoger, eftersom de flesta program använder samma dialogrutor för liknande funktioner. Som exempel kan nämnas Microsoft Word, vars öppna dokument-dialog ser nästan likadan ut som den dialog vi använder för att öppna arkiv. Ytterligare en fördel med att använda standarddialogerna är att de kommer att matcha respektive plattformars dialoger, eftersom wxWidgets använder varje plattforms API, istället för att emulera dialoger och widgets, som Qt gör.

7.2. Klasser

Vi har definierat egna klasser för en del funktionalitet, som exempelvis PathList-klassen för att spara sökvägar. wxWidgets-biblioteket innehåller visserligen en hel del klasser, men just den funktionalitet vi var ute efter krävde mer specifik hantering av datamedlemmar. Under avsnitt 7.2.3 finns denna containerklass beskriven mer i detalj.

7.2.1. Klassdiagram

Figuren här nedan visar förhållandet mellan de klasser vi definierat, och hur de hänger ihop med varandra. Objektet *wxWidgets framework* i diagrammet är ett objekt som innehåller alla wxWidgets-bibliotekets klasser, och är en abstraktion för att göra diagrammet mer överskådligt.



Figur 7.3: Förenklat klassdiagram över Winpaz

WinpazMainfrmApp ärver från en klass wxApp i wxWigets-biblioteket. Klassen måste implementera en virtuell metod, kallad OnInit(). Denna metod är den som utför allt förarbete som kan behövas före körning av applikationen, exempelvis behandling av argument som kan ha skickats till applikationen, och skapandet av ett fönster. Om metoden returnerar **true,** kommer wxWidgets att starta händelseloopen, och börja hantera händelser. Om den returnerar **false**, kommer wxWidgets istället avsluta applikationen, efter att ha städat upp minnesutrymmet efter sig.
WinpazMainFrm är härledd från en klass wxFrame i wxWidgets-biblioteket. Detta är det fönster som startas i OnInit()-metoden. Det är också i denna klass alla händelsehanterare för applikationen är definierade. WinpazMainFrm har också två instanser av containerklassen PathList, som hanterar sökvägar till filer och arkiv (se avsnitt 7.2.3.), samt två instanser av klassen ImageIndexList (se avsnitt 7.2.4.), som tillhandahåller par av indexnummer och filändelser till en lista av bilder som hör till wxTreeListCtrl-kontrollerna. Denna bildlista lagrar ikoner som läggs till varje nod.

7.2.2. ArchiveInterface-klassen

För att följa det objektorienterade paradigmet, valde vi att göra en klass för extrahering och komprimering. Den har ett programmeringsgränssnitt, ett *API*, som programmeraren kan använda för att tillämpa komprimering och extrahering i sitt program. Det är denna klass som är tänkt att kunna utökas med fler algoritmer i framtiden. För tillfället fungerar detta upplägg hjälpligt, men måste göras om helt vid en eventuell utökning av prototypen. Det är inte bra att man måste in i de privata metoderna för att lägga till nya algoritmer, utan detta borde kunna skötas med pekare till objekt av en basklass, som alla implementationer av nya algoritmer ärver från. Då kan klassen fungera som ett generellt gränssnitt, och nya algoritmer kan pluggas in genom att bara länka in en ny klass som implementerar algoritmen.
Här nedan följer en lista av alla metoder i klassen, tillsammans med en kort beskrivning av varje metod:

Publika metoder:

- **bool** Extract**(wxArrayString** *pathsToArchives***, wxString** *destination***)**
  Extract tar som argument en lista av sökvägar till de arkiv man vill packa upp (*pathsToArchives*), och en sökväg till mappen där man vill att de extraherade filerna ska hamna (*destination*). Extract anropar den privata metoden *DetermineTypeOfArchive*().

En **switch**-sats skickar sedan alla sökvägar till motsvarande privata extraheringsmetod, en i taget.

- **bool** Compress(**wxArrayString** *filepaths*, **wxString** *destination*)
  Compress tar samma argument som Extract, men *destination* är sökvägen till och namnet på det arkiv man vill skapa, med en filändelse som motsvarar vilken algoritm man vill använda. Därefter skickas listan med filsökvägar och destinationssökvägen till rätt privata metod för komprimering.
- **bool** ExtractEntries(**wxArrayString** *entryNames*, **wxString** *pathToArchive*)
  entryNames är en lista av kompletta filnamn man vill extrahera ur arkivet som specificerats av *pathToArchive*.
- **wxArrayString** GetEntryNames(**wxString** *pathToArchive*)
  GetEntryNames returnerar en lista över alla filer i arkivet som specificerats av *pathToArchive*.
- **int** GetEntrySize(**wxString** *pathToArchive*, **wxString** *entryName*)
- GetEntrySize returnerar storleken för filen *entryName* i arkivet som specificerats av *pathToArchive*.
- **int** GetArchiveSize(**wxString** *pathToArchive*)
  GetArchiveSize returnerar storleken för arkivet som specificerats av *pathToArchive.*
- **wxDateTime** GetDateTime(**wxString** *pathToArchive*)
  returvärdet är ett wxDateTime-objekt, ur vilket man kan hämta en tidsstämpel för när arkivet skapades

Privata metoder:

- **int** DetermineTypeOfArchive(**wxString** *fullName*)
  returvärdet är ett ID-nummer, som identifierar vilken arkivtyp filen *fullName* tillhör.
- **bool** CompressWithZip(**wxArrayString** *filepaths*, **wxString** *destination*)
  CompressWithZip komprimerar alla filer i listan *filepaths* till *destination*, med zip.
- **bool** CompressWithTar(**wxArrayString** *filepaths*, **wxString** *destination*)
  CompressWithTar komprimerar alla filer i listan *filepaths* till *destination*, med tar.
- **bool** CompressWithRar(**wxArrayString** *filepaths*, **wxString** *destination*)
  CompressWithRar komprimerar alla filer i listan *filepaths* till *destination*, med rar.
  Denna metod kommer inte att implementeras i prototypen. Den finns med i den här listan för eventuell framtida utökning.
- **bool** CompressWithPaz(**wxArrayString** *filepaths*, **wxString** *destination*)

CompressWithPaz komprimerar alla filer i listan *filepaths* till *destination*, med paz. Inte heller denna metod kommer inte att implementeras i prototypen. Den finns med i den här listan för eventuell framtida utökning.

- **bool** ExtractWithZip(**wxFileName** *filepath*, **wxString** *destination*)
  ExtractWithZip extraherar filen på platsen *filepath* till platsen *destination*, med zip.
- **bool** ExtractWithTar(**wxFileName** *filepath*, **wxString** *destination*)
  ExtractWithTar extraherar filen på platsen *filepath* till platsen *destination*, med tar.
- **bool** ExtractWithRar(**wxFileName** *filepath*, **wxString** *destination*)
  ExtractWithRar extraherar filen på platsen *filepath* till platsen *destination*, med rar.
- **bool** ExtractWithPaz(**wxFileName** *filepath*, **wxString** *destination*)
  ExtractWithPaz extraherar filen på platsen *filepath* till platsen *destination*, med paz. Inte heller denna metod kommer inte att implementeras i prototypen. Den finns med i den här listan för eventuell framtida utökning.

7.2.3. PathList-klassen

Vi behövde ett bra och intuitivt sätt att matcha sökvägar till alla filer som hanteras av programmet, med motsvarande nod i filträden. För detta ändamål utvecklade vi en containerklass med två datamedlemmar; en sträng med sökvägen, och en instans av klassen wxTreeItemId, som sparar ID-numret för en nod i ett träd. Nedan följer en lista över de metoder klassen tillhandahåller, samt en kort beskrivning av dem:

Publika metoder:

- **wxString** GetPath(**wxTreeItemId** *item*)
  Returnerar sökvägen som matchar *item*.
- **wxArrayString** GetPaths(**wxArrayTreeItemIds** *items*)
  Returnerar en lista av sökvägar som matchar *items*.
- **wxArrayString** GetAll()
  Returnerar en lista av alla sökvägar som finns i listan.
- **bool** Delete(**wxTreeItemId** *item*)
  Tar bort den nod som matchar *item*.
- **bool** Append(**wxTreeItemId** *item*, **wxString** *path*)
  Lägger en nod med ID-nummer *item* och sökvägen *path*, till listan.
- **bool** Exists(**wxTreeItemId** *item*)

  Exists returnerar *true* om det finns en nod i listan med id-nummer *item*.
- **bool** Exists(**wxArrayTreeItemIds** *items*)

Denna version av Exists returnerar true om det finns noder som matchar alla id-nummer i *items.*

- **int** GetSize()
  GetSize returnerar storleken på, alltså antalet element i listan.

7.2.3. ImageIndexList-klassen

Vi upptäckte i slutskedet av projektet att listan med de ikoner som läggs till alla filer i fil- och arkivträden växer väldigt snabbt, eftersom det för varje ny fil lades till en ikon i listan. Detta kan bli ett problem om programmet körs länge, och många filer läggs till. Det är inte ovanligt att man vill komprimera 1 000-tals filer i taget, och listan blir då väldigt stor. Därför utvecklade vi denna klass, som hanterar par av index till ikonlistan och filändelser, så att om tre Word-dokument läggs till i programmet, kommer det bara att läggas till en ikon. Man inser snabbt att om vi stället hade läst in 2 000 delar av en videofil, så hade varje del genererat en ny ikon. Nu blir det istället endast en ikon i listan.
Här nedan följer en lista över alla metoder i klassen:

Publika metoder:

- **void** Append(**const int imageIndex**, **const wxString** *extension*)
  Denna metod lägger till ett index/filändelse-par i listan.
- **int** GetIndex(**const wxString** *extension*)
  Den här metoden returnerar det index som hör till *extension*.
- **bool** Exists(**const wxString** *extension*)
  Med den här metoden kan man testa om en viss filändelse redan finns i listan.

7.3. Sammanfattning

Generellt sett har implementationen av applikationen gått bra, och utan större problem. wxWidgets är ett väldigt kraftfullt bibliotek att använda när man utvecklar grafiska applikationer, och har varit oerhört lätt att arbeta med. I och med att vi använde en utvecklingsmiljö som wxDev-cpp, har även mängden egenskriven källkod blivit mindre än befarat, eftersom wxDev-cpp genererar kodskelett för exempelvis alla händelsehanterare, och nästintill fullständig kod för alla widgets.
Det mest problematiska har varit de egendefinierade klasserna ArchiveInterface och PathList, men detta har mest berott på otillräckliga kunskaper i C++, och dålig planering av klassernas

funktionalitet. Frikopplingen av algoritmerna från ArchiveInteface-klassen fanns det som väntat ingen tid att implementera.

8. Resultat och rekommendationer

Av de mål vi satte upp i början av projektet (se kapitel 1.2.), har vi lyckats nå ungefär hälften. Vi har gjort en version för Windows, byggt in stöd för komprimering och extrahering med ZIP och TAR, samt implementerat drag & drop-funktionalitet. Vi trodde i början av projektet att det skulle vara lätt att använda extraheringskoden som WinRar tillhandahåller på sin hemsida, men den visade sig vara mycket mer komplicerad än vi anade. Därför uppnådde vi inte målet att implementera RAR-extrahering. Vårt arkiveringsgränssnitt är inte heller så lätt att bygga ut med fler algoritmer som vi ville. Vi hann inte med att göra Linux- och Mac-versioner, men vi valde att bygga vår lösning på wxWidgets-biblioteket, så vi har förberett för framtida portningar. Det sekundära målet, att implementera en enkel version av paz-algoritmen för testsyften, hanns inte heller med.

Så här i efterhand är vi förhållandevis nöjda med slutprodukten, trots att vår planering kunde ha varit avsevärt mycket bättre. Vikten av planering kan inte nog understrykas, och om vi sätts på ett liknande projekt i framtiden, kommer vi att satsa mycket tid i början av projekttiden till att planera.

Testversionerna var något vi lyckades väldigt bra med, och resultatet från testerna hjälpte oss oerhört mycket när vi utformade den slutgiltiga versionen. Att skriva testprototyper var ett smart drag, eftersom det hjälpte oss att se var eventuella problem kunde uppstå, och hur vi skulle lösa dem.

Vi rekommenderar att man till varje projekt bör avsätta en ordentlig portion tid i början, för planering och tester. Vi hade tillräckligt med tid till våra tester, men alldeles för lite tid för planering. Mer tid borde i början ha lagts på att göra diagram över klasser och även ett antal Use Case-diagram, så att vi hade vetat på förhand vilken funktionalitet som skulle behövas. Det hade nog också varit nyttigt att konsultera lärare angående olika programmatiska lösningar, som exempelvis vårt arkiveringsgränssnitt, och hur man skapar en bra bakomliggande struktur för sin applikation.

9. Summering

Här summerar vi vad vi lärt oss av projektet, vilka problem vi stött på under projektets gång, hur tiden disponerats, samt vilka lärdomar vi har kunnat dra av projektets utfall.

9.1. Nya kunskaper

Det enskilt viktigaste området vi har utökat våra kunskaper i var hur grafiska gränssnitt är uppbyggda, och hur de fungerar. Detta var något vi inte hade några kunskaper alls om innan det här projektet. Vi har även befäst våra färdigheter i C++-programmering, och skaffat oss erfarenheter i wxWidgets omfattande bibliotek, och hur det är uppbyggt.
Den viktigaste lärdomen vi har dragit av projektet är nyttan av att ha en bra planering. Utan planering hamnar man i ”Happy hacker”-träsket, och enda sättet att ta sig ur det är att gå tillbaka till början och göra en riktig planering.

9.2. Problem

Vi har stött på en hel del problem under projektets gång, både programmeringsrelaterade och rent administrativa. Bland de programmeringsrelaterade problemen kan nämnas svårigheten i att få tag i enkla moduler för komprimering och extrahering, med förståeliga API:er. När det gällde ZIP- och TAR-algoritmen fanns det stöd för dem i wxWidgets, så dessa två kunde implementeras förhållandevis enkelt. För RAR-algoritmen var det värre. Det fanns visserligen en klass i wxWidgets som utförde RAR-extrahering, som en tredje part bidragit med, men den gick inte att kompilera. Det var synd, för dess API fungerade precis som de för ZIP och TAR.
Det har också varit svårt att kontakta vår uppdragsgivare vid vissa tillfällen, och viktiga dokument som kravspecifikation har vi fått tillgång till först i slutskedet av projektet. Vi har dock haft möten löpande under projektets gång, och har då fått den information vi behövt för att kunna utveckla applikationen i enlighet med kravspecifikationen ändå.
Inga av dessa problem har varit oöverstigligt svåra att hantera, och på det stora hela har projektet fallit väl ut till slut.

9.3. Tidsåtgång

Av den tid vi allokerat till projektet har nog ungefär två tredjedelar använts till programmering och planering, medan den sista tredjedelen har gått till uppsatsskrivning. Detta är en inte helt orimlig uppdelning, men planeringen borde ha fått mer tid. Tid som man lägger på planering går

oftast att spara in på programmeringen istället, eftersom det blir mindre funderingar över hur man ska lösa diverse problem; man har ju så att säga redan löst dem vid ritbordet.

9.4. Lärdomar

En av de viktigaste lärdomarna vi dragit utifrån resultatet av projektet är att man måste planera sitt projekt bra från början, så att man kan undvika nödlösningar. Med bättre planering hade nog strukturen på vår applikation blivit bättre, och vi hade kanske haft tid att designa snyggare ikoner till knapparna.
Vi skulle också vara mycket mer aktiva i våra kontakter med uppdragsgivaren, för att kunna få feedback på hur projektet går, och om det går i rätt riktning. Dessutom skulle vi då kunna upplysa denne om olika svårigheter vi råkat på under arbetet, och kanske få hjälp med en del av dem.

10. Referenslista

[1] **Cross-platform GUI Programming with wxWidgets**

J. Smart och K. Hock, med S. Csomor. Prentice Hall PTR. 2006. 2 utg. ISBN 0-13-147381-6.

[2] **Grafiska gränssnitt - vad har användbarheten för betydelse?**

C. Emring och A. Funke. Handelshögskolan vid Göteborgs universitet, Institutionen för informatik. 1999.

[3] **The Essential Guide To User Interface Design**

W. O. Galitz. John Wiley & Sons, Inc. 2002. ISBN 0-471-084646

[4] **Wikipedia – den fria encyklopedin**

http://sv.wikipedia.org/wiki/Datakompression

Måndag den 19 mars 2007 kl. 11:57

[5] **WxWidgets roadmap**

http://wxwidgets.org/develop/roadmap.htm

Tisdag den 10 april 2007 kl. 13:30