Mobile Translator

En applikation för mobila Windows enheter

Mobile Translator

An application for Windows tablets

Niklas Salarp Gustafsson

Herman Othelius

Fakulteten för hälsa, natur- och teknikvetenskap

Datavetenskap

C-uppsats 15 hp

Handledare: Martin Blom

Examinator: Lothar Fritsch

Datum: 2016-06-07

Abstrakt

Företaget Saab AB i Karlstad har utvecklat produkten Paratus Pocket Translator för att hjälpa ambulanssjukvårdare kommunicera med patienter som inte talar samma språk. Eftersom denna produkt enbart finns på en utgående mobil plattform behöver Saab en omskrivning av denna produkt på en modern plattform.

Denna uppsats beskriver projektet Mobile Translator vilket även blev produktens arbetsnamn. Projektets huvudsakliga syfte var en ny version av den tidigare produkten Paratus Pocket Translator för Windows med bibehållen funktionalitet.

Projektet har resulterat i en Windows applikation baserad på moderna tekniker som tillfredställer Saabs behov och önskemål.

Abstract

The company Saab AB in Karlstad has developed the product Paratus Pocket Translator to help paramedics communicate with patients who do not speak the same language. This product is only available on an older mobile platform that has since been replaced. Saab is now in need of a rewrite of this product on a more modern platform.

This dissertation describes the project Mobile Translator, which also became the products working title. The project's main aim was a new version of the previous product Paratus Pocket Translator for Windows with maintained functionality.

The project has resulted in a Windows application based on modern technologies that satisfies Saab's needs and requests.

Förord

Denna uppsats är skriven vid Karlstads Universitet vårtterminen 2016. Projektet är utvecklat på Saab i Karlstad med tanke att inkluderas i deras nuvarande produktportfölj.

Vi vill framförallt tacka Martin Blom som varit vår handledare för uppsatsskrivandet samt Carl-Henric Smedman som varit vår handledare på Saab.

I övrigt vill vi även tacka personalen på Saab för deras stöd och intresse i projektet.

Innehåll

Figurer

Kapitel 1

Introduktion

När ambulanspersonal kommer fram till en patient i sitt hem eller på en olycksplats är det viktigt att man kan kommunicera med denna person och med eventuella anhöriga. Kommunikation är inte enbart viktig för att ställa frågor och få svar utan även för att meddela vad som sker för att minska oro hos patient och anhöriga. Om de man försöker kommunicera med varken pratar svenska eller engelska kan det vara svårt att få sagt det som behövs och för att säkerställa detta kan en tolk behövas. En tolk i detta fall kan vara en faktisk person som rings upp för att översätta vid behov men i dagens informationssamhälle kan det även ske med hjälp av informationsteknik. Saab AB har idag en produktlösning för handdatorer som bistår ambulanspersonal med enklare kommunikationsmöjligheter mellan flera språk genom förinspelade fraser. Handdatorer med denna plattform är idag svårtillgängliga då plattformen är utgående och Saab är därför i behov av en ny version av produkten för en modern plattform som passar ändamålet.

1.1 Summering

Projektet har resulterat i applikationen Mobile Translator som kommer inkluderas i Saabs produktfamilj Paratus. Alla krav som ställts på applikationen samt önskad funktionalitet är implementerad.

1.2 Disposition

Kapitel 1 Introduktion ger en kortare introduktion till projektet.

Kapitel 2 Bakgrund beskriver arbetsgivaren Saab, uppdraget, tillgängligt material och sist kommer en summering av verktyg och tekniker som använts i projektet.

Kapitel 3 Design beskriver hur applikationens användargränssnitt är utformat och jämförs med den tidigare produkten Paratus Pocket Translator.

Kapitel 4 Implementation beskriver tekniskt hur applikationen fungerar, kommunikation internt samt hjälpklasser och resurser som används av Mobile Translator.

Kapitel 5 Resultat och utvärdering beskriver kravlistan och kravuppfyllelse, den slutgiltiga produkten jämförs med vår planering och kapitlet avslutas med en utvärdering av verktyg och tekniker.

Kapitel 6 Utvärdering av projektet beskriver hur tiden har disponerats under projektets gång, arbetsmiljö samt möjligheter till vidareutveckling. Avslutas med en självvärdering av vår insats samt ett utlåtande från uppdragsgivaren.

Kapitel 2

Bakgrund

I detta kapitel beskrivs projektet Mobile Translator samt de tekniker och verktyg som har använts för att utveckla applikationen. För att få en bättre förståelse om projektets syfte beskrivs företaget Saab, dess lokala Karlstad kontor och produktfamiljen Paratus.

2.1 Saab

Saab är ett globalt företag med verksamhet i över 100 länder som tar fram produkter och tjänster inom både militärt försvar och civil säkerhet[15]. Företaget har ca 15.000 medarbetare i mer än 30 länder varav ca 12.000 på över 40 order i Sverige[16]. På kontoret i Karlstad verkar ett 10-tal medarbetare där de främst utvecklar produktfamiljen Paratus.

2.2 Paratus

Paratus är en produktfamilj som är speciellt framtagen för blåljusmyndigheter så som ambulanssjukvård, räddningstjänst och polis[17]. Paratus är latin för redo/beredd och rymmer allt från enklare handdatorer för ambulans och räddningstjänst till mer komplexa system som journalsystem i ambulans med momentan överföring till landstingets journalserver. Paratus Pocket Translator är en applikation som körs på en handburen enhet med pekskärm. Den används av ambulanssjukvårdare

när patienten inte pratar svenska eller engelska och ingen tolk finns tillgänglig. Applikationen hjälper till att identifiera patientens språk med hjälp av flaggor och text. När patientens språk är fastställt tillhandahåller applikationen flera inspelade frågor och instruktioner i patientens språk. Paratus Pocket Translator har stöd för 16 olika språk med 96 förinspelade fraser per språk.

2.3 Uppdraget

Uppdraget har varit att portera den befintliga applikationen Paratus Pocket Translator från den äldre mobila plattformen Windows Mobile till en fullfjädrad Windows applikation. Windows Mobile applikationen använder ramverket .NET Compact Framework som är en delmängd av ramverket .NET Framework. De grundläggande funktionella kraven för den porterade applikationen som fick arbetsnamnet Mobile Translator var bibehållen funktionalitet från Paratus Pocket Translator. I uppdraget fanns det en förfrågan om att utveckla applikationen på så sätt att i största möjliga utsträckning underlätta för eventuella framtida plattformsbyten. Applikationen skulle kunna köras på en generisk Windows 7 tablet eller högre med en pekskärmsvänlig design med dynamiskt skalbart användargränssnitt. Applikationen skulle kunna användas av både svensk- och engelsktalande personal, därmed skulle man kunna byta visningsspråk för användargränssnittet. Det var även önskvärt att det lätt skulle kunna gå att lägga till fler språk i framtiden.

2.4 Tillgängligt material

Vid uppstart av projektet fanns det tillgång till både källkod och körbar fil av applikationen Paratus Pocket Translator. I materialet fanns inspelningar med olika fraser som skulle återanvändas till den nya applikationen samt textfiler med översättningar till de olika språken.

2.5 Tekniker

I detta avsnitt presenteras de tekniker och designmönster som använts under projektets gång.

2.5.1 C# och .NET Framework

.NET är ett ramverk utvecklat av Microsoft som primärt körs i operativsystemet Windows[25]. De viktigaste komponenterna är ett omfattande klassbibliotek och CLR (Common Language Runtime)[21]. CLR är en virtualiserings-komponent som hanterar exekveringen av alla .NET program och tillhandahåller exempelvis minneshantering, felhantering och **garbage collection**.

C# är ett objektorienterat programspråk utvecklat av Microsoft[22]. Språket är baserat på C++ men har även likheter med programspråket Java.

Alla program skrivna i .NET oberoende av programmeringsspråk kan användas tillsammans då alla exekveras av CLR som kompilerar dessa till maskinkod m.h.a. en process som kallas just-in-time compilation[21].

2.5.2 WPF

WPF (Windows Presentation Foundation) är Microsofts senaste ramverk för att utveckla användargränssnitt och är inkluderat i .NET Framework[27]. WPF försöker ge en enhetlig programmeringsmodell för att bygga applikationer som separerar användargränssnitt från affärslogiken.

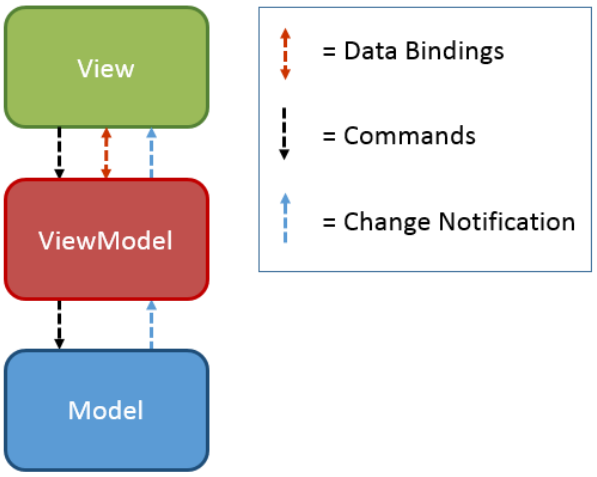
2.5.3 XAML

XAML (Extensible Application Markup Language) är ett XML-baserat (Extensible Markup Language) språk som vanligtvis används för att skapa ett programs användargränssnitt. Detta implementerades 2006 då WPF (Windows Presentation Foundation) infördes som en del av .NET Framework 3.0[28].

2.5.4 MVVM

MVVM (Model View ViewModel) är ett designmönster som separerar användargränssnitt och affärslogik[24]. Med hjälp av MVVM kan man skriva ett program och sedan byta ut komponenten View för att eventuellt byta plattform eller göra om designen av applikationen. I View undviker man därför att skriva

någon affärslogik, denna presenterar endast informationen som ViewModel tillhandahåller som i sin tur baseras på Model. Kommunikationen mellan de olika komponenterna visas i figur 2.1 nedan.



FIGUR 2.1: Kommunikation i MVVM

2.6 Verktyg

I denna del beskriver vi de verktyg vi har använt oss av under projektets gång.

2.6.1 Visual Studio 2015

Visual Studio är en utvecklingsmiljö från Microsoft som används för att utveckla applikationer till många olika plattformar. Det innehåller verktyg till att utveckla allt från användargränssnitt till logik. Visual Studio stödjer ett flertal olika programmeringsspråk och erbjuder möjligheten att testa det användaren utvecklar på flera olika sätt[11].

2.6.2 Git

Git är ett versionshanteringsprogram som kan användas i projekt av alla olika storlekar[2]. Versionshanteringen sker lokalt och varje användare jobbar på en klon av projektet, på detta sätt kan man testa olika funktioner utan att oroa sig över

vad andra medarbetare i projektet jobbar på. Sedan sammanfogar man de delar man vill inkludera i projektets delade repository.

2.6.3 TortoiseGit

Vi har använt oss av TortoiseGit[19] är ett komplement till Git i form av ett grafiskt användargränssnitt, detta underlättade arbetet då vi fått en bra översikt över projektets framgång.

2.6.4 Balsamiq

Balsamiq är ett program för att skapa en skiss av en applikations användargränssnitt. Programmet kan köras direkt i webbläsaren eller som en fristående applikation och erbjuder ett stort bibliotek av fördefinierade grafiska komponenter som kan användas genom drag- och släpprincipen[18].

2.6.5 Audacity

Audacity är ett ljudprogram som kan användas för efterbehandling av flera typer av ljudfiler. Efterbehandling såsom normalisering, tvättning från oljud och exportering till olika format är några exempel på vad Audacity kan göra[1].

2.6.6 Google Dokument

Google Dokument är ett ordbehandlingsprogram som körs direkt i webbläsaren där flera användare kan arbeta samtidigt[5]. Alla ändringar sparas direkt i molnet och genom en historik kan man se vad som ändrats i dokumentet och av vem. Funktionalitet för exportering till olika format som Pdf och Word finns inbyggt.

2.6.7 ShareLaTeX

ShareLaTeX är en LaTeX-editor som körs direkt i webbläsaren där flera användare kan arbeta samtidigt[14]. ShareLaTeX användargränssnitt är uppdelat i två fönster

där det ena fönstret innehåller LaTeX-editorn och det andra presenterar en kompilerad version av LaTeX-koden som en Pdf.

2.7 Sammanfattning

Detta kapitel har gett en bakgrundsbeskrivning till företaget Saab och dess produktfamilj Paratus. Arbetsnamnet Kapitlet innehåller även en beskrivning av uppdraget, förutsättningar samt tekniker som används under projektets gång.

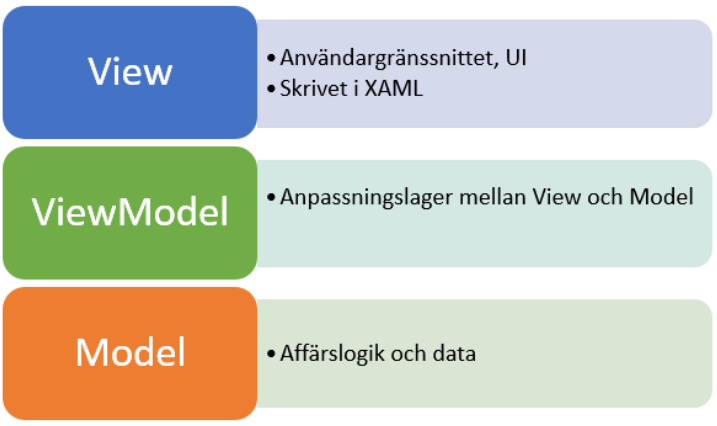
Kapitel 3

Design

I detta kapitel beskrivs designen av applikationen Mobile Translator och de designval som gjorts under arbetet. Jämförelser görs mellan den tidigare applikationen Paratus Pocket Translator och den nya Mobile Translator.

3.1 Applikationsdesign

Det som påverkat applikationens design mest är valet att följa MVVM-designmönstret. Valet att följa designmönstret härrör i önskemålet från arbetsgivaren att i största mån möjligt designa applikationen på ett sätt som underlättar eventuella plattformsbyten i framtiden. Då det huvudsakliga syftet för MVVM är ytterligare abstraktion av arkitekturen åtföljs detta. Att följa MVVM mönstret leder typiskt sett till mer kod, men med färre kontaktpunkter mellan objekt. Anledningen till att man strävar efter färre kontaktpunkter mellan objekt är för att det leder till mer flexibel kod som inte påverkar andra delar lika mycket. Att applikationen designats enligt MVVM-designmönstret innebär praktiskt att de flesta klasserna delas in i Views, Models och ViewModels för att särskilja på deras uppgifter vilket visas i figur 3.1.



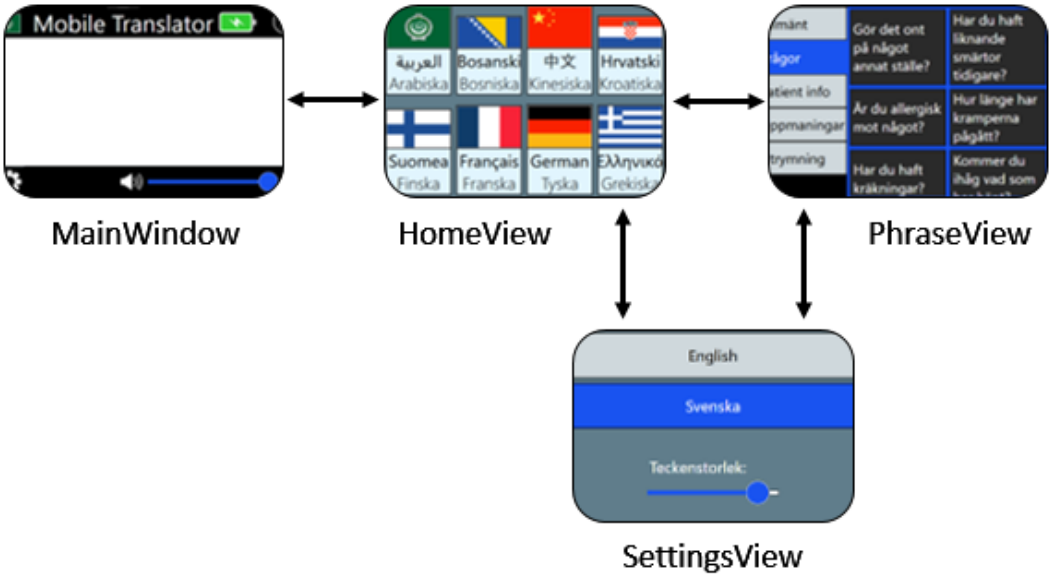
FIGUR 3.1: De olika klassernas uppgiftsområde i MVVM

Dessa kommer i fortsättningen vara hänvisade till som vyer, modeller och vymodeller i löptext. I projektet åtskiljs dessa tydligt i egna mappar samt med standardiserade namngivelser för att underlätta navigeringen.

3.2 Views

Applikationen består som tidigare nämnts av tre olika typer av klasser, vyer, vymodeller och modeller. Vyer är XAML filer, som likt HTML är ett markup-språk för att definiera grafiska användargränssnitt[12]. Modeller är klasser som hanterar datahantering och affärslogik. Vymodeller är de klasser som binder samman vyer och modeller.

Användargränssnittet i applikationen Mobile Translator består av ett fönster och tre olika vyer. Alla dessa kommer att beskrivas var för sig i detta kapitel. Designen på den nya applikationen Mobile Translator och den äldre Paratus Pocket Translator kommer jämföras med text och bilder. Ordningen sker efter Mobile Translators flöde vilket kan ses i figur 3.2.



FIGUR 3.2: Flödesschema för Mobile Translator

3.2.1 MainWindow

När applikationen Mobile Translator startas öppnas först detta fönster, se figur 3.3. Fönstret i applikationen är alltid synligt och har som uppgift att visa en vy och tillhandahålla navigeringstöd samt ett antal kontroller som skall finnas globalt i applikationen. I fönstret finns två mindre fasta fält definierat i den översta och understa delen och en större yta emellan dessa som är behållaren där en vy kommer visas.

I behållaren visas som standard HomeView vid uppstart av applikationen.

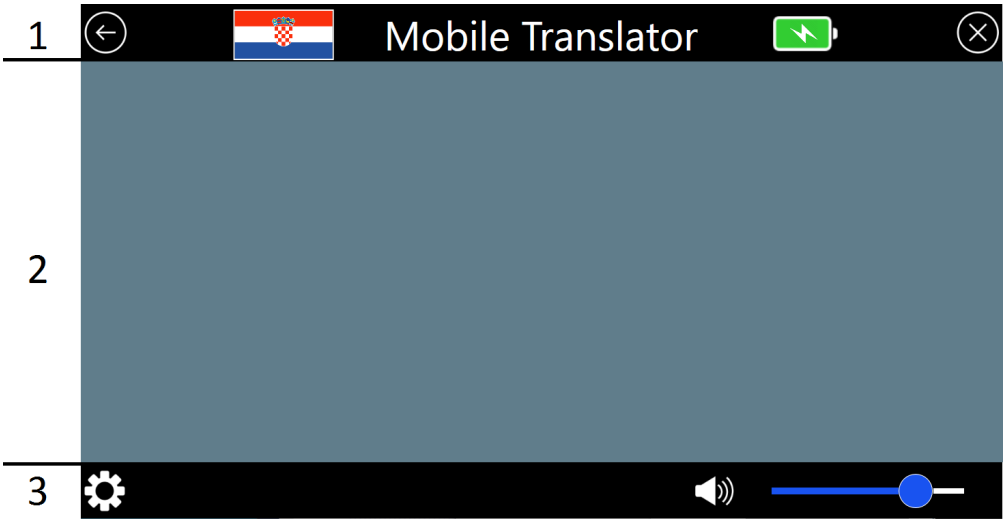
Det övre fältet innehåller tillbakanavigering, aktuellt valt språk i form av landets flagga, applikationens namn och knapp för att stänga applikationen. Det nedre fältet innehåller en knapp för att gå till generella inställningar samt volymkontroll.

Navigeringsknappen för att gå tillbaka visas enbart när det är möjligt att gå tillbaka vilket innebär hela tiden då man inte är i startvyn.

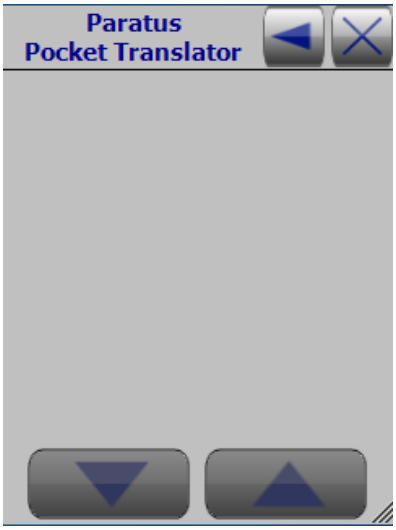
Flaggikonen som indikerar aktuellt valt språk i applikationen visas enbart när ett språk valts i startvyn.

Med volymkontrollen kan användaren justera applikationens volym, vilket är nödvändigt då inte alla enheter har fysiska kontroller för detta.

Skillnaderna mellan Mobile Translator och Pocket Translator som synes på figurerna nedan är utöver de tydliga storleksskillnaderna att Mobile Translator har ett liggande format och Paratus Pocket Translator har ett stående format. Anledningen till valet av formatet är att Mobile Translator är anpassat för en tablet och Paratus Pocket Translator för en mobilenhet.



FIGUR 3.3: Uppdelningen av Mobile Translators fönster



FIGUR 3.4: Paratus Pocket Translators fönster

Då arbetsytan inte varit ett problem har en batteri-ikon, en ljudkontroller och en användar-inställnings-knapp lagts till i Mobile Translator. Utöver detta navigerar man upp och ner med hjälp av en rullningslist istället för knappar i Mobile Translator och tillbakaknappen har flyttats till applikationens övre vänsterhörn.

3.2.2 HomeView

Detta är applikationens vy som visas vid uppstart, se figur 3.5. I denna vy presenteras samtliga språk som applikationen har stöd för med en flaggbild och två texter. De två texterna är språknamn i det egna språket samt språknamn i det inställda språket för användargränssnittet. Flaggbilden är en egen tryckbar knapp och de två språktexterna bildar tillsammans också en knapp. Dessa två knappar representerar tillsammans ett språk som stöds av applikationen, i fortsättningen refererat som ett språkobjekt. Språkobjekt är gjort med en mall som definierar hur det skall se ut för att dynamiskt kunna lägga till flera språk. Dessa språkobjekt presenteras och skalas automatiskt i ett rutnät med fem kolumner och skulle skärmytan inte räcka till för att visa alla objekt kommer en rullningslist visas på applikationens högersida för att indikera att användaren kan rulla ner för att visa fler språk. Att presentera varje språk på det här sättet var viktigt för att enkelt kunna lägga till fler språk i framtiden utan ändringar i applikationen. När användaren trycker på en flaggknapp så läses namnet på språket upp i respektive språk, om användaren trycker på textknappen under flaggan så navigeras man vidare till nästa vy.

De väsentliga skillnaderna mellan Mobile Translator och Pocket Translator som synes på figurerna nedan är att fler än fyra språkobjekt kan visas i Mobile Translator, vilket i normalfall är åtminstone 10. Något som inte syns på bilderna är att navigering och ljuduppspelningsfunktionerna är skiftade, det vill säga att texten numera navigerar och flaggan spelar upp ljud. Anledningen till att dessa skiftades var att vid tester av oss själva och andra medarbetare på Saab uppfattades detta som mer naturligt när texten inte längre såg ut som en länk.

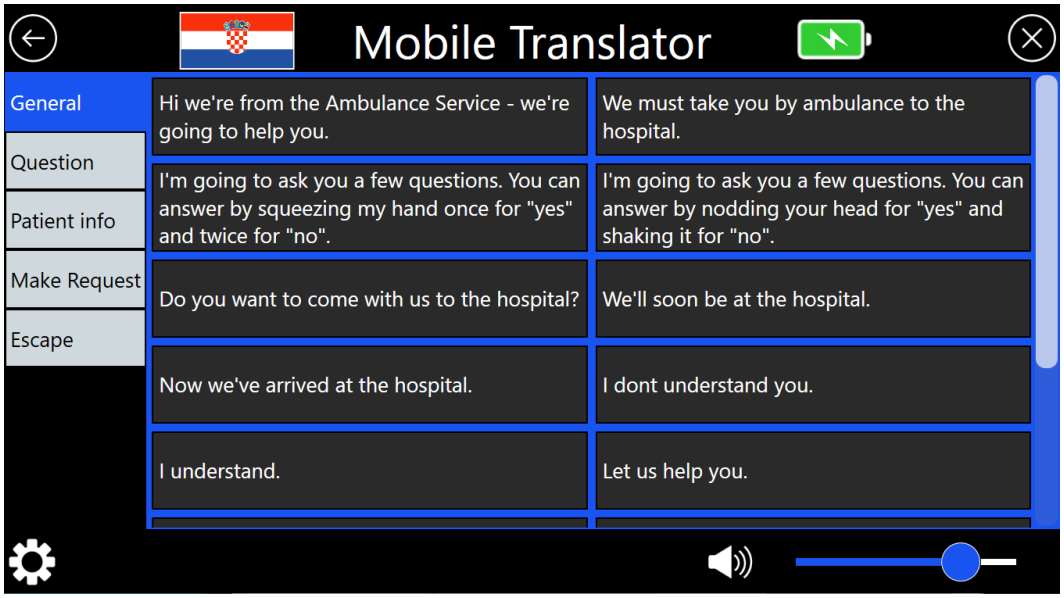
FIGUR 3.5: Mobile Translator uppstarts-vy, HomeView.

FIGUR 3.6: Paratus Pocket Translator uppstarts-vy

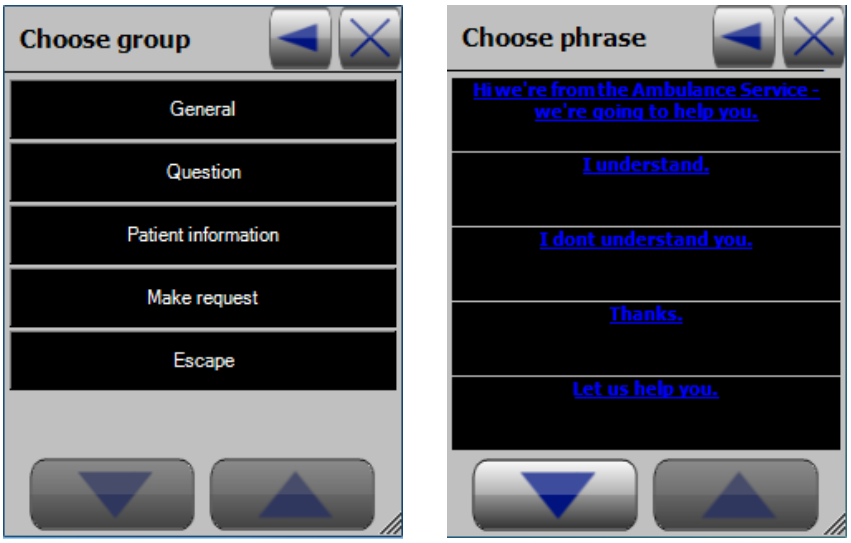
3.2.3 PhraseView

När användaren har tryckt på en flaggtext som beskrivet i föregående vy kommer man till denna vy, se figur 3.7. Det språk som valdes indikeras av språkets flagga uppe till vänster. Vyn består av fem vertikala flikar intill vänsterkanten och på övrig yta visas den valda kategorins innehåll. Kategorins innehåll består av ett antal knappar med text i som är uppdelade i två kolumner och med en rullningslist till höger om inte alla knappar får plats. När användaren valt en kategori visas detta tydligt med att kategorin ändrar färg och innehållet till höger uppdateras. Om användaren klickar på en knapp under en kategoris innehåll spelas den visade textfrasen upp i det språk som valts i föregående vy.

Skillnaderna mellan Mobile Translator och Pocket Translator som synes på figurerna nedan är även här att mer innehåll kan visas samtidigt. En viktig skillnad är också att antal tryck för att navigera runt i de olika kategorierna har kortats ner då två av Paratus Pocket Translators vyer har blivit en vy i Mobile Translator. Det är alltså inte längre nödvändigt att använda tillbakaknappen vid varje val av en kategori som i den äldre applikationen.



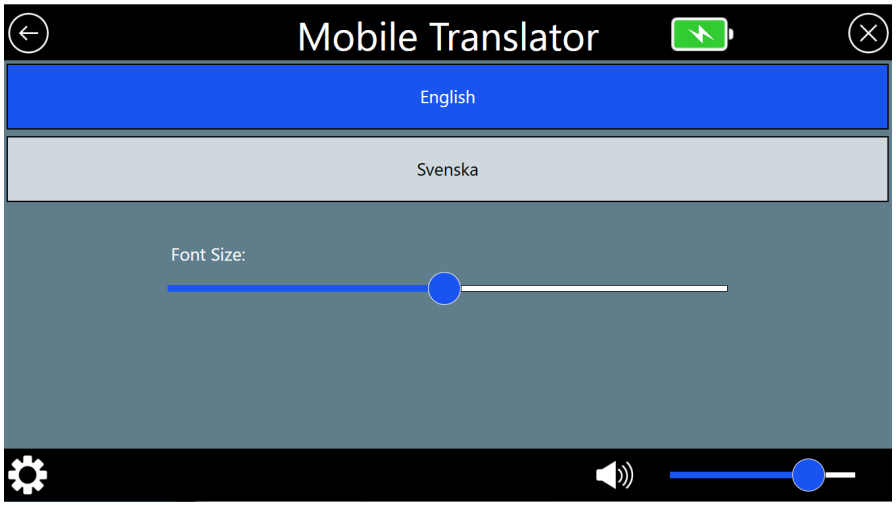
FIGUR 3.7: Mobile Translators kombinerade kategori- och fras-vy



FIGUR 3.8: Paratus Pocket Translator's kategori- och fras-vy

3.2.4 SettingsView

Denna vy kan nås genom kugghjulsknappen i applikationens nedre vänstra hörn, se figur 3.9. Eftersom denna knapp ligger i applikationens fönsterfält visas den oavsett vilken vy som visas för tillfället. Vyn innehåller samtliga inställningar för applikationen Mobile Translator som användaren skall kunna ändra. Här kan användaren ändra användargränssnittets visningsspråk mellan svenska och engelska samt justera storleken på applikationens teckensnitt med ett skjutreglage. Alla inställningar som ändras uppdateras i realtid vilket innebär att användaren ser exempelvis teckenstorleken ändras medan skjutreglaget rör sig. Alla användarinställningar sparas i applikationens inställningar så att de behålls vid omstart av applikationen.



FIGUR 3.9: Mobile Translators användarinställningar, SettingsView

Paratus Pocket Translator hade inga möjligheter att ändra inställningar direkt i användargränssnittet så här finns inget att jämföra med.

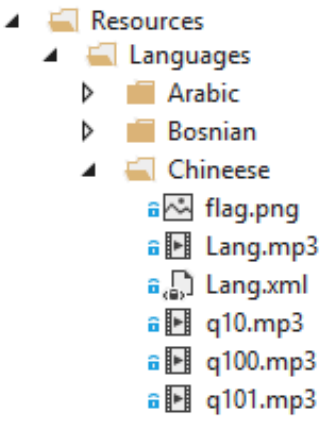
3.3 ViewModels

Vymodeller är de klasser som binder samman vyer och modeller och agerar som ett anpassningslager mellan dessa, se figur 2.1. Varje vy har en egen vymodell som tydligt presenteras i namngivelserna, exempelvis vyn HomeView och vymodellen HomeViewModel. Det är vymodellen som ansvarar för att vyn får tillgång till data ifrån modellen på ett presentabelt sätt för att vyn skall kunna rita upp detta för användaren. Vymodeller innehåller även någon typ av logik för att dess vy skall vara funktionell så som navigering eller händelser vid knapptryck vilket tas upp under kapitlet Implementation.

3.4 Models

Modeller är klasser som hanterar datahantering och affärslogik. I applikationen Mobile Translator finns enbart en modell som heter Language. Modellen tillhandahåller all information om ett visst språk så som flaggbild, språknamn i svenska och engelska samt i det egna språket. Modellen innehåller även en sökväg till en lokal mapp på enhetens hårddisk, härmed refererad som en resursmapp, se figur 3.10.

All läsning ifrån denna resursmapp sköts av modellen och den innehåller alla språk i en separat mapp med tillhörande ljudfiler och flaggbild. I resursmappen finns även en XML fil som modellen läser in där språknamn på det egna samt alla användargränssnittets visningsspråk finns. Anledningen till att inläsningen sker på detta sätt är att fler språk med tillhörande ljudfiler enkelt skall kunna läggas till utan att göra andra ändringar i applikationen.



FIGUR 3.10: Mappstruktur för resurser i Mobile Translator

3.5 Flerspråksstöd

Ett av kraven för applikationen var flerspråksstöd i användargränssnittet och de två visningsspråk som skulle stödjas initialt var svenska och engelska. Visual Studio hade inbyggd funktionalitet för att stödja flerspråksstöd vilket valdes att användas. Funktionaliteten innebar att varje visningspråk sparades i en egen resursfil som applikationen sedan kunde växla mellan. En resursfil i detta sammanhang är en typ av XML fil i Microsofts filformat .resx[10]. En resx-fil innehåller en tabell med ett namn och ett värde, se figur 3.11.

Name	Value
FontSize	Teckenstorlek:
G1	Allmänt
G2	Frågor
G3	Patient info
G4	Uppmaningar
G5	Utrymning

Name	Value
FontSize	Font Size:
G1	General
G2	Question
G3	Patient info
G4	Make Request
G5	Escape

FIGUR 3.11: Ett utdrag ur den svenska resurs-filen, Resources.sv.resx till vänster och den engelska, Resources.resx till höger.

Applikationen Mobile Translator har två resursfiler där alla fraser och texter i programmet är definierade, en på svenska och en på engelska. Språket bestäms av de två slutbokstäverna på dessa filer som identifierar språket med hjälp av landskoderna enligt standarden ISO 639-1[23]. Applikationens två resursfiler är Resources.resx vilket är en engelsk version av alla texter samt Resources.sv.resx som är den svenska versionen. Resources.resx behöver inte heta Resources.en.resx då denna är definierad som standard.

Applikationens vyer har ingen statisk text definierad utan alla texter hämtas direkt ifrån dess resursfiler genom att hänvisa till den aktuella resursfilens namnfält och resursfilen returnerar motsvarande text ifrån värdefältet. Exempelvis ifrån figur 3.11 ovan om namnet “G2” refererats i en vy hade värdet “Frågor” returnerats om valt språk varit svenska, annars “Question” om valt språk varit engelska.

3.5.1 Infralution Localization

En nackdel som påträffades med Visual Studios lösning för flerspråksstöd var att växlingen mellan applikationens resursfiler medan applikationen kördes krävde att applikationen startades om. För att undvika dessa omstarter valdes en påbyggnadslösning av Visual Studios resurshantering ifrån Infralution vid namn Localization[4]. Denna påbyggnadslösning innehöll modulen Resx Extension som gjorde det möjligt att byta användargränssnittets visningsspråk i realtid utan omstart.

3.6 Sammanfattning

Detta kapitel har gått igenom vad det innebär rent praktisk att applikationen skrivs enligt MVVM-designmönstret. Klasser delas upp efter deras uppgift som vyer, vymodeller eller modeller. Vyer är XAML filer, som likt HTML är ett markupspråk för att definiera grafiska användargränssnitt[12]. Modeller är klasser som hanterar datahantering och affärslogik. Vymodeller är de klasser som binder samman vyer och modeller. Varje vy ifrån applikationen Paratus Pocket Translator jämförs med sin motsvarighet i Mobile Translator med både figur och text. Utöver detta beskrivs även Mobile Translators vyer och hur flerspråksstöd designats.

Kapitel 4

Implementation

FIGUR 4.1: Implementations-kapitlets delar

Förra kapitlet beskrev de designval som skett under arbetets gång, designmönstret MVVM, vyer samt användargränssnittets språkstöd. I detta kapitel presenteras de delar som bygger upp användargränssnittet mer i detalj och på en mer teknisk nivå. Kapitlet börjar med att ta upp kommunikation i MVVM. Därefter tas applikationens huvuddelar upp, som är View (XAML), ViewModel och Model. Slutligen presenteras övriga hjälpklasser och komponenter som används utanför MVVM arkitekturen.

4.1 Kommunikation i MVVM

FIGUR 4.2: Översikt över Kommunikation i MVVM kapitlets innehåll enligt markeringar

Det har tidigare nämnts att det huvudsakliga syftet för designmönstret MVVM är ytterligare abstraktion av arkitekturen och att detta görs genom att separera användargränssnittet (vyn) från datan (modellen) genom en mellanliggande klass

(vymodellen). Utöver detta skall inte vymodellen ha någon referens till vyn direkt utan vyn skall enbart känna till vymodellen och inte det motsatta. För att inte bryta detta kan alltså inte vymodellens egenskaper (eng. properties) och metoder användas direkt. Funktionalitet för att uppnå dessa krav tillhandahåller WPF genom **bindings**, **events** och **commands**.

4.1.1 Binding och Event

En **binding** eller **data-binding** är ett sätt att uppdatera data mellan modellen och användargränssnittet. En **binding** binder samman en komponent i vyn med en egenskap i dess vymodell. Det är även möjligt att binda en samlingskomponent i vyn till exempelvis en lista i vymodellen, där listan i sin tur innehåller flera modeller med tillhörande egenskaper. När den bundna egenskapen uppdateras speglas detta i användargränssnittet. För att denna spegling skall fungera måste både användargräns-snittets **binding** och den bundna egenskapen tillhandahålla en uppdaterings notifikation genom ett event som heter **PropertyChanged**. **PropertyChanged** kommer ifrån gränssnittet **INotifyPropertyChanged** som inkluderar metoden **NotifyPropertyChanged**. Metoden kallas alltid på när en egenskap ändras så att användargränssnittet skall bli uppmärksammad om ändringen.

4.1.2 Command

Ett **command** är en alternativ lösning till den klassiska event-baserade lösningen som .NET ramverket alltid haft vid exempelvis ett knapptryck i användargränssnittet[7]. Problemet med den klassiska event-baserade lösningen är att alla intressenter av eventet måste tillhandahålla en egen event-handler vilket leder till att ett starkt beroende skapas mellan dessa. Systemet måste också hålla reda på alla event-handlers och det starka beroendet innebär att det kan förhindra **garbage collection**, vilket är .NET ramverkets sätt att frigöra minnesresurser som inte längre används[8]. Förhindrandet av **garbage collection** är en återkommande orsak för minnesläckor i .NET[7]. Ett följdproblem av det starka beroendet är att event som är deklarerade i en vy måste ha en event-handler i dess bakomliggande kod. Med bakomliggande kod menas den klassfil som alltid skapas för varje XAML fil i WPF. Om en event-handler inte finns i dess bakomliggande kod kommer det

resultera i kompileringsfel. Anledningen till att man undviker att skriva kod i vyns bakomliggande kodklass är att detta minskar applikationens testbarhet[6].

Lösningen på ovan nämnda problem är att använda ett **command** istället för den event-baserade lösningen. Ett **command** exponerar en enda event-handler som en vys komponenter kan binda till genom en **data-binding**. Eftersom en **data-binding** först utvärderas när applikationen körs kommer det inte resultera i några kompileringsfel och då de enbart har ett löst beroende innebär det att det inte finns risk för minnesläckor[7].

Ett **command** är en implementation av gränssnittet **ICommand** som är en del av Microsofts .NET ramverk. Gränssnittet specificerar tre delar:

Execute(object) anropas när ett **command** aktiveras. Metoden har en parameter som kan användas för att skicka med ytterligare information. Eftersom parametern är av typen objekt kan det vara nästan vad som helst vilket innebär att mottagarsidan måste ha logik för att läsa ut vilken datatyp objektet är i.

CanExecute(object) returnerar en boolean som indikerar om ett **command** kan köras. Parametern är densamma som för **Execute**. Om denna metod returnerar falskt kommer dess kopplade kontroller i användargränssnittet automatiskt att inaktiveras.

CanExecuteChanged är en event-handler som anropar **CanExecute** varje gång det bör omvärderas, vilket sköts automatiskt.

En stor nackdel med **ICommand** implementationen är att för varje **command** som läggs till krävs det att man skapar en ny klass som implementerar gränssnittet. För att kringgå denna begränsning används en lösning som heter **RelayCommand**.

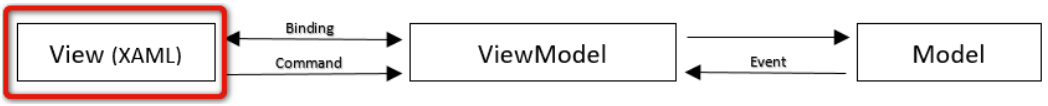
4.1.2.1 RelayCommand

RelayCommand är en klass som implementerar **ICommand** gränssnittet utan statiska **Execute** och **CanExecute** metoder[3]. Dessa metoder specificeras istället i klassens konstruktor. Det är då möjligt att skapa ett **command** med hjälp av att göra en ny instans av **RelayCommand** och ange två parametrar till konstruktorn vilket är metoderna som skall användas som **Execute** och **CanExecute**, se exempel nedan.

```
1 DoSomethingCommand = new RelayCommand(DoSomething, CanDoSomething);
```

Det är viktigt att metoderna som skickas in som parametrar finns i samma klass där man instansierar ett nytt **command** då det annars resulterar i att metoderna inte hittas.

4.2 Views



FIGUR 4.3: Översikt över View kapitlets innehåll enligt markering

Applikationens vyer är skrivna i språket XAML som tidigare beskrivits i 2.5.3 är inkluderat i WPF. XAML är en lösning som bygger på att åtskilja användargränssnittet från kod vilket innebär att en designer kan bygga det grafiska användargränssnittet utan programmeringskunskaper. En utvecklare kan sedan använda **databindings** och **commands** ifrån XAML koden för att koppla in funktionalitet i användargränssnittet. Ett enkelt exempel på hur man skapar några användargränssnitts-komponenter i XAML gentemot i C# kod visas nedan[29].

```
<StackPanel>
    <TextBlock Margin="20">Welcome to the World of XAML</TextBlock>
    <Button Margin="10" HorizontalAlignment="Right">OK</Button>
</StackPanel>
```

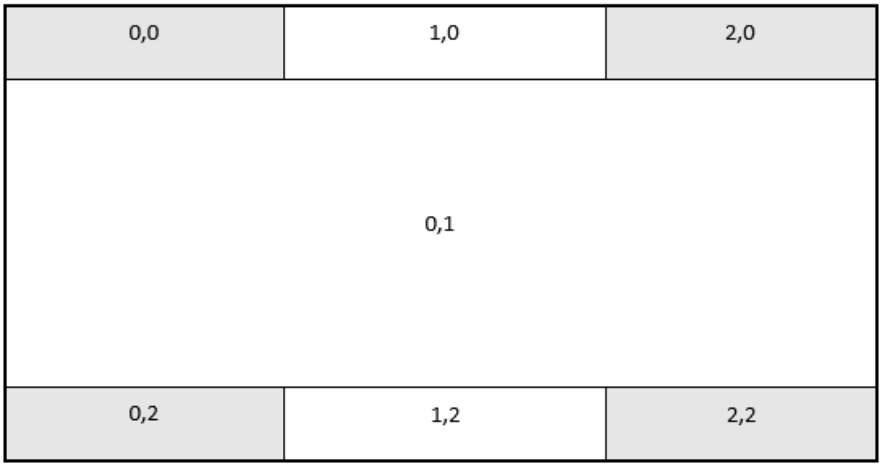
FIGUR 4.4: En StackPanel innehållande ett textblock och en knapp skapas i XAML

```
// Create the StackPanel
StackPanel stackPanel = new StackPanel();
this.Content = stackPanel;

// Create the TextBlock
TextBlock textBlock = new TextBlock();
textBlock.Margin = new Thickness(20);
textBlock.Text = "Welcome to the World of XAML";
stackPanel.Children.Add(textBlock);

// Create the Button
Button button = new Button();
button.Margin= new Thickness(10);
button.Content = "OK";
stackPanel.Children.Add(button);
```

FIGUR 4.5: En StackPanel innehållande ett textblock och en knapp skapas i C#

4.2.1 MainWindow

Detta fönster visas i figur 3.3 där man tydligt kan se uppdelningen av fönstret genom ett övre och undre fält samt en yta i mitten för att presentera andra vyer. I fönstrets XAML kod finns enbart en komponent av typen **Frame**. En **Frame** är en komponent som kan fyllas med såväl statiskt som dynamiskt innehåll och som tillhandahåller navigerings-funktionalitet för att byta dess dynamiska innehåll. Statiskt innehåll kan exempelvis vara som i Mobile Translators fall de övre och undre fälten med tillhörande kontroller. Dynamiskt innehåll är den vy som presenteras i mitten. Applikationens **Frame** är baserat på en **ControlTemplate** vilket är en mall som specificerar dess uppbyggnad genom andra underkomponenter. Denna mall är uppdelad i olika sektioner med ett rutnät (eng. **Grid**) där uppdelningen sker med rader och kolumner som visas i figur 4.6.

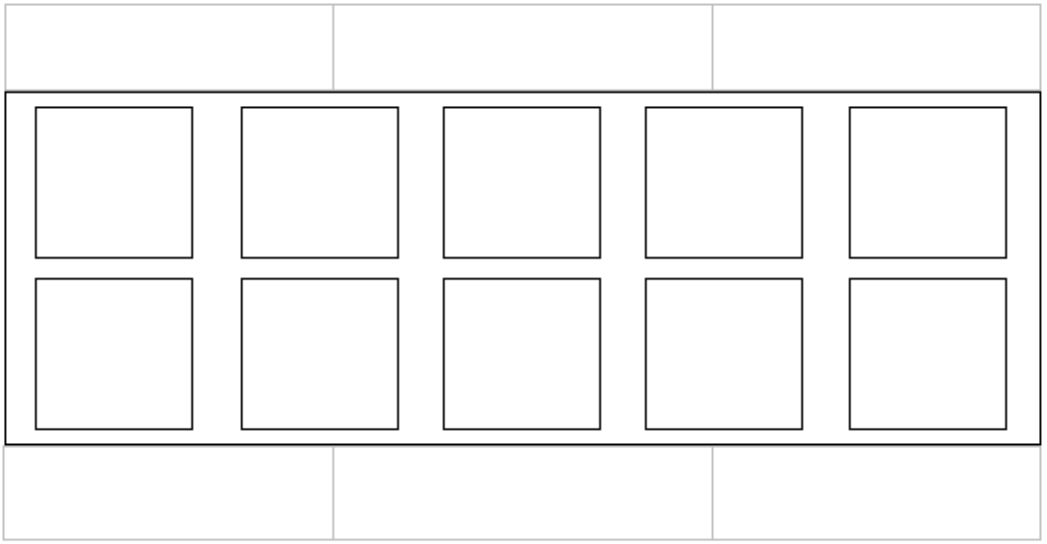


FIGUR 4.6: Rutnäts-uppdelning av MainWindow förtydligat genom ett koordinatsystem

Alla hörn sektioner som markerats i grått i figuren innehåller även en **DockPanel**, vilket är en komponent som har som uppgift att grupperna underliggande komponenter på ett sätt så att de inte kolliderar med varandra. Det är även möjligt att vänster respektive högerjustera de grupperade komponenterna så att de exempelvis alltid läggs nära applikationens fönsterkant.

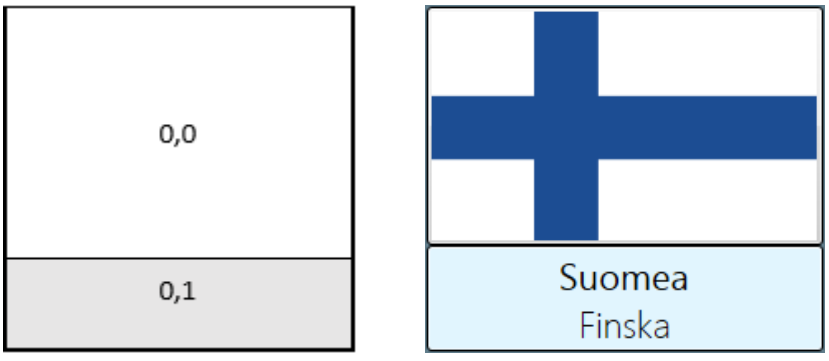
4.2.2 HomeView

HomeView som också är applikationens uppstarts-vy visas i figur 3.5. Vyn består av en komponent av typen **UniformGrid** som är ett utökat rutnät som ser till att all upp- och nedskalning av en komponent har bibehållet bildförhållande. Det bibehållna bildförhållandet medför att allt innehåll i komponenten också har sitt korrekta bildförhållande oavsett hur vyn har skalats för att passa på en specifik bildskärm. Antal kolumner definieras direkt i komponenten, vilket är fem för Mobile Translator och antal rader blir då dynamiskt beroende på dess innehåll.



FIGUR 4.7: HomeView's fem fasta kolumner, med två rader som exempel och med applikationens fönster utgråat runtomkring

Varje ruta i figur 4.7 ovan representerar ett language-objekt, vilket är vår modell som innehåller all information för att presentera ett språk så som flaggbild, språknamn i applikationens olika visningsspråk samt det egna språket. Anledningen till att vyn måste kunna presentera ett dynamiskt antal language-objekt är för att inläsning av alla applikationens språk också sker dynamiskt vid uppstart av applikationen. Lösningen på detta är att använda en komponent som heter **ItemsControl** i vyn. Denna komponent fylls genom en **data-binding** till dess vymodell. En **ItemsControl** definierar hur ett enstaka item, vilket i detta fall är ett language-objekt skall se ut genom en mall. I denna **ItemsControl** ligger en kontroll i form av en **trigger** som kontrollerar att flaggbilden i ett language-objekt existerar, om den inte existerar sätts "Flag missing" som flaggbild.



FIGUR 4.8: Mall för ett language-objekt till vänster med slutresultat till höger

Mallen som visas i figur 4.8 ovan specificerar två rader i dess ruta där det övre är betydligt större och innehåller en bild på språkflaggan. Den nedre rutan innehåller två staplade texter där den övre är språknamnet i det egna språket och den undre texten är språknamnet i det aktuella visningsspråket. De två rutorna är också vardera en egen knapp där flaggbilden är bundet till ett **command** för ljuduppspelning och texterna är bundet till ett navigerings-command.

4.2.3 PhraseView

PhraseView visas i figur 3.7 och innehåller en **TabControl** med fem olika flikar (eng. tabs). Varje flik är en kategori som innehåller ett visst antal knappar som är grupperade två och två bredvid varande genom att använda en **UniformGrid** med två kolumner. Varje knapp har en knapptext som överensstämmer med en fras, exempelvis "Vad är ditt namn?". Knapptexten hämtar dessa fras-texter i rätt visningsspråk genom en **binding** med ett text-id till biblioteket resx extension, (se 3.5.1). Varje knapp har också ett **command** angivet för ljuduppspelning. **Command** parametern är en enkel sträng med samma text-id som användes för att hämta knapptexten. Denna parameter motsvarar namnet på en ljudfil och används för att hitta den ljudfil med samma fras som texten på knappen, se figur 4.9.

```
<UniformGrid  Columns="2">
    <Button Content="{Resx Q10}"
        Command="{Binding PlaySoundCommand}" CommandParameter="Q10" />
    <Button Content="{Resx Q108}"
        Command="{Binding PlaySoundCommand}" CommandParameter="Q108" />
</UniformGrid>
```

FIGUR 4.9: Ett knapp-par på en rad skapats i XML med knapptext, command och command parameter

Detta **command** har som tidigare nämnts en metod som heter **CanExecute**, i detta fall kontrollerar metoden om den angivna ljudfilen existerar och om ljudfilen saknas inaktiverar den knappen. Denna inaktivering sker som tidigare beskrivits automatiskt om dess kopplade **CanExecute** metod returnerar falskt. En inaktiverad knapp får som standard ett ljusgrått utseende med liknande text för att indikera att den inte går att trycka på. För att ytterligare förtydliga detta används en **StyleTrigger** för att dölja knappen helt. En **StyleTrigger** är något som ändrar knappens utseende automatiskt när en annan angiven egenskap hos

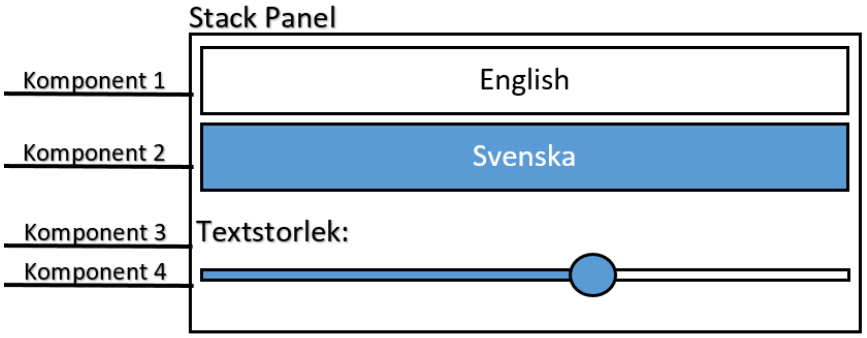
knappen ändras. I detta fall kontrolleras egenskapen om knappen är aktiverad eller inte och sätter synligheten på knappen utefter detta som visat nedan i figur 4.10.

```
<Style TargetType="Button" BasedOn="{StaticResource PhraseViewButton}">
    <Style.Triggers>
        <Trigger Property="IsEnabled" Value="False">
            <Setter Property="Visibility" Value="Hidden"/>
        </Trigger>
        <Trigger Property="IsEnabled" Value="True">
            <Setter Property="Visibility" Value="Visible"/>
        </Trigger>
    </Style.Triggers>
</Style>
```

FIGUR 4.10: Style trigger som används på alla knappar i PhraseView för att dölja dessa om de är inaktiverade

4.2.4 SettingsView

SettingsView visas i figur 3.9 och innehåller kontroller för att styra applikationens visningsspråk och teckenstorlek. Vyn är uppbyggd genom en *StackPanel* (Se figur 4.11) som innehåller först de två knapparna för val av visningsspråk följt av en textkomponent och skjutreglaget för teckenstorleken. En **StackPanel** fungerar genom att stapla dess innehåll vertikalt direkt på varandra. För att åstadkomma rymd mellan komponenterna används egendefinierade marginaler som kan sättas separat för varje sida (vänster, höger, över och under) på en komponent. De två språkknapparna binder till ett **command** i dess vymodell för att ändra visningsspråk. Detta **command** innehåller också en kontroll för vilket visningsspråk som är aktivt och detta visas i vyn genom att inaktivera motsvarande knapp. På samma sätt som tidigare beskrivits i under rubriken PhraseView används en **StyleTrigger** för att att ändra utseende på den inaktiverade kontrollern. I detta fall markeras knappen i en blå färg för att indikera att det är det aktiva valet. Texten för textstorlek hämtas som tidigare beskrivet i design kapitlet genom resx extension för att visas i rätt visningsspråk. Skjutreglaget använder en **two-way-binding** vilket innebär att den inte bara speglar en egenskap utan den kan också påverka denna egenskap direkt. Skjutreglaget binder direkt till egenskapen font size i applikationens globala inställningar vilket kan nås då vyn deklarerat inställningarna som en statisk resurs.



FIGUR 4.11: SettingsView's uppbyggnad

4.3 ViewModels

FIGUR 4.12: Översikt över ViewModel kapitlets innehåll enligt markering

En vymodell är en abstraktion av dess kopplade vy och exponerar egenskaper och **commands** för vyn. Varje vymodell har en konstruktor utan parametrar så att de kan instansieras direkt ifrån dess vy. En vymodell är skriven i C# som en vanlig klassfil (.cs). Kapitlet inleder med att beskriva basklassen ViewModelBase vilket samtliga vymodeller ärver ifrån och går sedan igenom alla vymodeller.

4.3.1 ViewModelBase

Klassen som alla vymodeller ärver ifrån och som innehåller generell funktionalitet för dessa. Denna klass implementerar gränssnittet **INotifyPropertyChanged** som inkluderar metoden **NotifyPropertyChanged** vars uppgift är att synkronisera vymodellen och det grafiska användargränssnittet när egenskaper ändras. Synkroniseringen görs genom metoden **SetField** som anropas i alla egenskapers **setter**.

```
protected bool SetField<T>(ref T field, T value, [CallerMemberName] string propertyName = null)
{
    if (EqualityComparer<T>.Default.Equals(field, value)) return false;
    field = value;
    NotifyPropertyChanged(propertyName);
    return true;
}
```

FIGUR 4.13: Metoden SetField

SetField är en metod som kontrollerar att värdet som skickas in inte är detsamma som föregående och om så inte är fallet sätter den detta till det nya värdet. Metoden använder även en ny funktionalitet ifrån .NET 4.5 ramverket som heter **CallerMemberName** som syns i figur 4.13. Det **CallerMemberName** gör är att sätta variabeln **propertyName** till samma namn som egenskapen vars **setter** kallade på metoden **SetField**. Om .NET versionen skulle behövas sänkas från 4.5 till exempelvis 4.0 kan man ta bort **CallerMemberName** och istället skicka in en fast sträng direkt i **NotifyPropertyChanged** metoden.

När det nya värdet sätts anropas metoden **NotifyPropertyChanged** så att alla intressenter av denna egenskap meddelas om ändringen. Detta kan även göras direkt i en egenskaps **setter** men skillnaden är att med **SetField** är en egenskaps **setter** betydligt mer kompakt än utan **SetField** vilket visas i figur 4.14.

```
public string GuiLanguage
{
    get { return _guiLanguage; }
    set { SetField(ref _guiLanguage, value); }
}
```

```
public string GuiLanguage
{
    get { return _guiLanguage; }
    set
    {
        if(_guiLanguage != value)
        {
            _guiLanguage = value;
            NotifyPropertyChanged();
        }
    }
}
```

FIGUR 4.14: En egenskap med SetField till vänster och utan till höger

Med metoden **SetField** minskar man varje egenskaps **setter** och undviker duplicerad kod som visats i figur 4.14. Med duplicerad kod menas alltså att inte varje egenskap behöver definiera samma kontroll om värdet ändrats samt att i så fall kalla på **NotifyPropertyChanged**.

4.3.2 MainWindowModel

Denna vymodell står för logiken bakom fönstret MainWindow. Den har tre egenskaper som är **Volume**, **BatteryPercent** och **ShowChargingIcon**.

Volume sätter och returnerar aktuell volym i operativsystemet med hjälp av Windows ljudbibliotek. Ljudbiblioteket nås genom att importera dll filen winmm.dll som finns i alla Windows-installationer sedan Windows 2000[9].

BatteryPercent returnerar systemets aktuella batteriprocent genom det inkluderade biblioteket *system information*. Batterinivåns värde hämtas två gånger i minuten genom ett eget tids-event.

ShowChargingIcon returnerar en boolean som indikerar om enheten är nätansluten. Denna egenskap uppdateras genom eventet **SystemEventsPowerModeChanged** som utlöser vid in och urkoppling av nätanslutning. Eventet kommer ifrån det inkluderade biblioteket *system information*. När eventet utlöser anropas den egna metoden **UpdatePowerStatus()** som uppdaterar **ShowChargingIcon** genom att läsa ut informationen ifrån eventet om enheten är nätansluten eller inte.

4.3.3 HomeViewModel

Denna vymodell står för logiken bakom vyn HomeView. Den har två egenskaper som är **Languages** och **CurrentLanguage**.

Languages är en lista **ObservableCollection** med Language-objekt. Denna lista innehåller alla språk som applikationen kan spela upp fraser i.

CurrentLanguage är ett Language-objekt som indikerar vilket språk som valts av användaren i vyn HomeView. **CurrentLanguage** sätts först då användaren navigerar ifrån vyn genom klassen Navigator, som hanterar all navigering i applikationen. I vymodellens konstruktor läses samtliga språkmappar in som finns i applikationens resursmapp. Sökvägen för varje mapp används för att skapa ett nytt Language-objekt och sparas i listan **Languages**.

Vymodellen implementerar designnmönstret Singleton vilket innebär att man begränsar antal instanser av klassen till enbart ett objekt[26]. Detta görs genom att göra vymodellens konstruktor privat och istället skapas en egenskap som håller en

referens till vymodellen. När egenskapens **getter** anropas skapas en instans av vymodellen om det inte redan finns en. Om det redan finns en instans så returneras en referens till denna. Detta medför att denna instans blir statisk och kan kommas åt direkt ifrån andra vymodeller vilket är viktigt då egenskaperna **Languages** och **CurrentLanguage** används ifrån andra vymodeller.

4.3.4 PhraseViewModel

Denna vymodell står för logiken bakom vyn PhraseView och har en egenskap **CurrentLanguage** och ett **command PlaySoundCommand**.

CurrentLanguage är precis som i HomeViewModel ett Language-objekt som indikerar vilket språk som valts av användaren i vyn HomeView. Denna egenskap sätts i klassens konstruktor till samma som HomeViewModels egenskap med samma namn.

PlaySoundCommand exponerar metoderna **PlaySound(object)** och **PlaySound(object)** där parametern är en unik textsträng som indikerar namnet på en ljudfil.

CanPlaySound(object) skapar en sökväg till en ljudfil på hårddisken med hjälp av **CurrentLanguage** och parametern. En kontroll görs om ljudfilen existerar och om så är fallet visas knappen i användargränssnittet. Om ljudfilen skulle saknas döljs knappen för användaren.

PlaySound(object) skapar på samma sätt som ovan en sökväg till ljudfilen och utför ytterligare en kontroll att ljudfilen finns och om så är fallet spelas denna upp genom klassen SoundPlayer (se 4.5.5), som är klassen som hanterar all ljuduppspelning.

4.3.5 SettingsViewModel

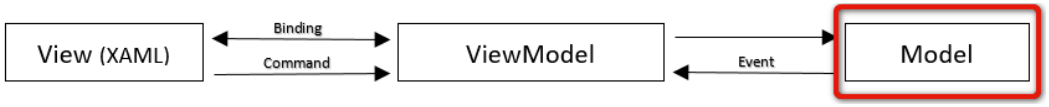
Denna vymodell står för logiken bakom vyn SettingsView. Inga egenskaper finns i denna vymodell utan här definieras enbart ett **command** som är **ChangeGuiLanguageCommand**. Detta command aktiveras när en användare trycker på någon av knapparna för att byta språk i applikationen. En parameter skickas med för att identifiera vilken språkknapp som tryckts på, exempelvis “sv” för svenska

eller "en" för engelska. **ChangeGuiLanguageCommand** består av två metoder **CanChangeGuiLanguage()** och **ChangeGuiLanguage()**.

CanChangeGuiLanguage() kontrollerar om knappen skall vara aktiverad och körs kontinuerligt i användargränssnittet på de språkvals-knapparna som finns. Kontrollen går ut på att jämföra knappens parameter gentemot applikationens inställningar som håller vilket det aktiva visningsspråket är i användargränssnittet. Om det aktiva visningsspråket skulle vara desamma som knappens parameter kommer knappen att inaktiveras samt markeras i en blå färg för att indikera att detta är det aktuella valet.

ChangeGuiLanguage() ändrar det befintliga visningsspråket i applikationens användargränssnitt. Valet utläses genom parametern och applikationens inställningar uppdateras med detta värde. Eftersom alla språknamn som visas under språkflaggan i HomeView behöver uppdateras baserat på detta val anropas ytterligare en metod som heter **SetLocalizedString()** för samtliga språkobjekt, vilket tas upp under kapitlet 4.4.2 Language.

4.4 Models

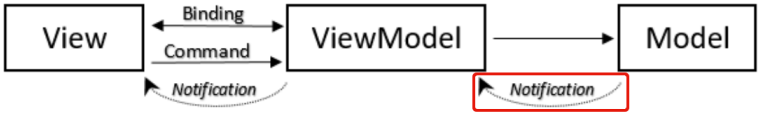


FIGUR 4.15: Översikt över Model kapitlets innehåll enligt markering

En modell representerar data och information och bör utformas på ett sådant sätt att den inte känner till någon vy eller vymodell. Detta innebär att modellen inte skall komma åt, referera eller interagera direkt med någon vy eller vymodell. Ett typiskt exempel på en modell är en kontakt i en kontaktlista där en kontakt innehåller egenskaper för namn, telefonnummer och adress. En modell är skriven i C# som en vanlig klassfil (.cs). Kapitlet inleder med att beskriva basklassen ModelBase vilket samtliga modeller ärver ifrån och går sedan igenom alla modeller.

4.4.1 ModelBase

Klassen som alla modellklasser ärver ifrån och som innehåller generell funktionalitet för dessa. På samma sätt som i ViewModelBase (se 4.3.1) implementeras gränssnittet **INotifyPropertyChanged** med tillhörande metod **NotifyPropertyChanged**. Klassen innehåller också metoden **SetField** som beskrivits i ViewModelBase. Den väsentliga skillnaden är här att synkroniseringen sker i två steg. När en egenskap i vymodellen ändrades meddelades vyn direkt om detta. Skillnaden här blir att vyn har bundit till en egenskap i vymodellen som innehåller en eller flera modell-objekt, exempelvis en lista. När ett enskilt modell-objekt ändras meddelar först modellen detta till vymodellen som sedan meddelar detta till vyn.



FIGUR 4.16: Det extra meddelandet som behövs mellan ViewModel och Model

4.4.2 Language

Denna klass representerar ett språk som applikationen har stöd för att spela upp fraser i. Klassen innehåller sex egenskaper som är **LanguagePath**, **LanguageNameNative**, **LanguageNameEn**, **LanguageNameSv**, **Flag** och **GuiLanguage**.

LanguagePath är sökvägen till en lokal mapp på enheten där all information kring det aktuella språket sparas. Denna sökväg tas emot i klassens konstruktor ifrån HomeViewModel (se 4.3.3).

LanguageNameSv, **LanguageNameEn** och **LanguageNameNative** är samtliga det aktuella språkets namn i de angivna språken svenska, engelska samt det egna språket. Alla dessa texter läses ut ifrån en XML fil kallad Lang.xml som finns i språkmappen.

Flag är sökvägen till en bild som skall representera språket. I första hand används en bild på landets flagga som förknippas med språket, exempelvis Finlands flagga för finska. Detta är inte alltid möjligt då vissa språk inte alltid förknippas med ett specifikt land och då används istället en flagga som på annat sätt hör till språket. Ett exempel är arabiska där Arabförbundets flagga används vilket är

en samarbetsorganisation med ett 20-tal medlemsländer som har arabiska som officiellt språk[20].

GuiLanguage är en egenskap som antingen är densamma som **LanguageNameSv** eller **LanguageNameEn**. Egenskapens uppgift är att returnera rätt text baserat på vilket som är det aktiva visningsspråket i användargränssnittet och detta hålls reda på med hjälp av metoden **SetLocalizedString()**.

SetLocalizedString() är en metod som körs varje gång visningsspråket ändras och som kontrollerar det aktuella visningsspråket gentemot denna egenskap. Anledningen till att **GuiLanguage** finns överhuvudtaget även om det egentligen rör sig om en duplicerad egenskap är att applikationens vyer behöver en gemensam egenskap de kan binda till. Denna egenskap måste kunna visa språkets namn i samtliga av applikationens visningsspråk, vilket för stunden är svenska och engelska. Egenskapen använder sig av metoden **NotifyPropertyChanged** vilket meddelar det grafiska användargränssnittet när egenskapen ändras så detta kan uppdateras för användaren.

4.5 Hjälpklasser

Hjälpklasser är de klasser som hamnar utanför MVVM arkitekturen och som kan användas överallt i projektet. Alla hjälpklasser är skrivna i C# som en vanlig klassfil (.cs). Kapitlet inleder med att beskriva CustomCommands, MultiValueConverter och Navigator som används av applikationens vyer. Därefter beskrivs klasserna SingleInstance och Soundplayer som används av applikationens vymodeller. Kapitlet avslutas med att redovisa för applikationens resurser så som ljudfilerna och hur dessa har bearbetats.

4.5.1 CustomCommands

Denna klass hanterar globala **commands** som skall finnas tillgängliga för flera vyer direkt i applikationen. Globala **commands** är möjliga genom att göra klassen statisk och varje vy kan definiera klassen som en statisk resurs. Anledningen till att göra dessa globala är för att slippa definiera dem i varje vys vymodell. Det finns tre **commands** definierade i denna klass: **PlaySoundCommand**, **NavigateCommand** och **CloseCommand**.

PlaySoundCommand exponerar metoden **PlaySound(object)** där parametern är en tuple innehållande ett language-objekt och en textsträng. Utifrån tuplen byggs en sökväg fram genom att ta language-objektets egenskap **LanguagePath** och lägger till textsträngen som är ett filnamn med filändelsen. Sökvägen kontrolleras så att den pekar på en existerande mp3-fil och om så är fallet spelas denna upp genom klassen SoundPlayer.

NavigateCommand exponerar metoden **PlaySound(object)** där parametern antingen kan vara en tuple av samma typ som ovan eller enbart en textsträng. Först kontrolleras om parametern är en tuple och om så är fallet är textsträngen namnet på den vyn som skall navigeras till och language-objekten skickas vidare som en parameter. Om inte navigate parametern är av typen tuple kontrolleras det istället om det är en textsträng och om så är fallet är detta vyn som skall navigeras till men ingen parameter skickas med. I båda fall anropas den statiska klassen Navigator som innehåller metoden **Navigate()** som beskrivs under 4.5.3. **CloseCommand** exponerar metoden **Close()** vars enda uppgift är att avsluta applikationen.

4.5.2 MultiValueConverter

Ett **command** kan enbart ta emot en parameter och inte flera. I vissa fall kan det däremot vara nödvändigt att skicka fler än en parameter till ett **command** ifrån en vy och för att lösa detta problem får man ha i åtanke att parametern till ett **command** är i form av ett objekt. Att parametern är ett objekt och inte en fast datatyp innebär att det är möjligt att använda en datatyp som hanterar mängder. En vy har inte samma begränsning som ett **command** vad gäller antalet parametrar utan har inbyggd funktionalitet för detta genom vad som kallas en **multibinding**. En **multibinding** möjliggör bindning av två eller flera egenskaper direkt ifrån vyn. För att lösa flera parametrar till ett **command** behövs alltså ett mellansteg mellan det att vyns **multibinding** skickar parametrarna och innan det att de når sitt **command**. Detta mellansteg är en MultiValueConverter, vars jobb är att ta emot flera parametrar ifrån en **multibinding** och paketera om dessa parametrar till en gemensam datatyp som sparar alla parametrar separat. När denna MultiValueConverter paketerat om parametrarna i en gemensam datatyp skickas de i form av ett objekt till det **command** det skall till. Detta **command** måste i sin tur veta vilken datatyp objektet skall läsas ut som.

4.5.3 Navigator

Denna statiska klass hanterar all navigering mellan vyer i applikationen och exponerar enbart metoden **Navigate()** som tar emot antingen en eller två parametrar. Den första parametern är ett vynamn i form av en textsträng och den andra parametern är ett objekt av typen Language. För att utföra navigeringen till en ny vy hämtas först applikations fönster MainWindow upp av metoden **GetMainWindow()**. När klassen har hittat fönstret används den inbyggda servicen NavigationService för att ersätta tidigare vy i fönstret med den nya.

4.5.4 SingleInstance

Denna klass implementerades för att förhindra att flera instanser av applikationen kördes samtidigt. Detta var viktigt då Mobile Translator skulle kunna startas av andra applikationer som körs i helskärmsläge. Problemet som kan uppstå här är att Mobile Translator inte stängs korrekt innan man kommer tillbaka till applikationen som startade den. Detta skulle innebära att nästa gång man startar Mobile Translator så startas en ny instans av applikationen och den gamla ligger också kvar och körs. Det man ville uppnå var istället att den äldre instansen av applikationen som redan körs istället visas och ingen ny instans startas. Lösningen bygger på klassen SingleInstance.cs som är skriven av Microsoft[13]. Mobile Translators startpunkt är klassen App.xaml och i dess bakomliggande kod i App.xaml.cs finns applikationens **Main()** metod som körs varje gång programmet startar. I **Main()** metoden görs först en kontroll så att inga andra instanser av applikationen körs och om ingen instans av applikationen redan körs så startas en ny. Om en instans redan körs så kommer ingen ny instans att skapas utan istället anropas den befintliga instansen av applikationen med parametrar från den nya som försöker starta. Detta görs genom att App.xaml.cs implementerar gränssnittet **ISIngleInstanceApp** som finns definierat i SingleInstance.cs. Detta gränssnitt innehåller enbart metoden bool **SignalExternalCommandLineArgs(IList<string>args)** vars uppgift är att aktivera den första instansen av applikationen med parametrar från den andra.

4.5.5 SoundPlayer

För att applikationen skall kunna spela upp ljudfiler i filformatet Mp3 importeras Windows ljudbibliotek genom filen Winmm.dll i klassen SoundPlayer. Här finns metodern **Play()**, **Open(file)** och **Stop()** som vardera skapar en intern instruktion som skickas till ljudbiblioteket. För att spela upp en ljudfil måste den först öppnas genom metoden **Open(file)** där parametern är sökvägen till filen. Efter att man öppnat en ljudfil kan den spelas upp med metoden **Play()**. Innan man kan spela upp nästa ljudfil måste metoden **Stop()** anropas även om ljudfilen spelats upp i sin helhet. För att underlätta detta skapades en till metod **Play(string file)**. I denna metod anropas först metoden **Stop()** följt av **Open(file)**. Slutligen skickas instruktionen för uppspelning till Windows ljudbibliotek. Att göra allt detta i en metod medför att man också kan avbryta pågående uppspelningar med en ny uppspelning då alltid metoden **Stop()** anropas på först.

4.6 Resurser

I projektet finns mappen Resources som innehåller alla resurser som används i applikationen så som språkmappar med tillhörande ljudfiler som visas i figur 3.10. Här beskrivs de olika filerna som representerar ett språk samt hur de bearbetats innan de använts i Mobile Translator.

4.6.1 Språkfiler

När applikationen byggs kopieras mappen Languages med och i denna mapp lagras alla språk som Mobile Translator stödjer att spela upp fraser i. Varje språk har en separat undermapp namngiven efter sitt språk som innehåller tre viktiga filer utöver alla ljudfiler med fraser. De tre viktiga filerna är flag.png, som är en flaggbild, Lang.mp3, som är en ljudfil som frågar om man pratar det berörda språket och slutligen Lang.xml som innehåller texter för språknamn på det egna samt alla användargränssnittets visningsspråk.

4.6.2 Ljudfiler

Ljudfilerna som har återanvänts från Paratus Pocket Translator var i filformatet Wav, vilket innebar att dessa tog upp totalt 250-300 MB (Megabyte) på disken. Programmet AudaCity användes för att konvertera dessa till filformatet Mp3 och ljudfilerna fick istället en total storlek på 40 MB med bibehållen ljudkvalité. Vissa ljudfiler hade ett par tysta sekunder både före och efter det att frasen hade lästs upp. De ljudfilerna klipptes till för att minska filstorleken ytterligare och för en bättre respons då ljudfilen läses upp direkt vid knapptryck. I ljudfilerna identifierades även brusljud, buller och högfrekventa ljud som lyckades filtreras bort med hjälp av AudaCity. Slutligen normaliserades alla ljudfiler, det vill säga att dess maxvolym angavs till samma värde så att den generella volymen uppfattas som lika. Språket turkiska som tidigare inte används i Pocket Translator då inspelningarna haft för dålig kvalité blev med alla ovan nämnda steg godtagbara och kunde läggas till i Mobile Translator.

4.7 Sammanfattning

Detta kapitel har gått igenom hur funktionaliteten i Mobile Translator är uppbyggd på en teknisk nivå. Först har kommunikationen internt med hjälp av **bindings**, **commands** och **events** förklarats följt av en genomgång av vardera vy. Efter detta har vymodellerna med dess egenskaper och metoder gåtts igenom utförligt. Även modellen Language har förklarats och slutligen har övriga hjälpklasser samt resurser som används i applikationen beskrivits.

Kapitel 5

Resultat och utvärdering

Detta kapitel inleds med en kravuppfyllelse där krav som uppfyllts och krav som inte uppfyllts beskrivs. Därefter följer en presentation av resultatet för projektet Mobile Translator där jämförelser görs mellan skisser och slutgiltigt resultat. Kapitlet avslutas med en utvärdering av de tekniker och verktyg som använts under projektets gång.

5.1 Kravuppfyllelse

I början av projektet fanns ingen färdig kravspecifikation utan denna arbetades ut tillsammans med arbetsgivaren. Kraven prioriterades aldrig upp i någon specifik ordning i början av projektet. Anledningen till detta var att det inte fanns så många krav och att dessa ansågs finnas tid att slutföra. Istället fanns det planer för vidareutveckling om projektets omfång skulle visa sig vara för litet. Ett exempel på en vidareutveckling var möjlighet för text to speech i applikationen vilket inte fanns tid för att genomföra.

Nedan listas de krav som utarbetades tillsammans med arbetsgivaren sorterat i funktionella och icke-funktionella krav med en beskrivning hur dessa uppfyllts.

5.1.1 Funktionella krav

1. Applikationen skall innehålla samma funktioner som tidigare applikation.	☑
2. Applikationen skall fungera utan internetåtkomst.	☑
3. Applikationen skall stödja från 800*600 i skärm-upplösning och uppåt.	☑
4. Applikationen skall ha flerspråksstöd för användargränssnittet.	☑
5. Applikationen bör kunna byta visningsspråk i runtime.	☑

Mobile Translator innehåller alla funktioner ifrån Paratus Pocket Translator utom en. Den funktionalitet som saknas är möjligheten att kunna ringa en tolk direkt ifrån applikationen. Denna funktionalitet valdes bort av två anledningar. Den ena anledningen var att det innebar administrativt arbete att hålla telefonlistor uppdaterade och den andra anledningen var att telefonmodul i hårdvaran inte kunde garanteras. Kravet anses därmed som uppfyllt av arbetsgivaren.

Mobile Translator ställer inga krav på internetåtkomst alls.

Mobile Translator har en testad minimi-upplösning på 800*600. En höjning av upplösningen innebär att komponenter automatiskt skalas upp och det finns därmed ingen specifik max-upplösning.

Mobile Translator har flerspråksstöd för användargränssnittet och detta kan även bytas i runtime. Visningsspråken som är inkluderade är svenska och engelska.

5.1.2 Icke-funktionella krav

Användbarhet

6. Likvärdig användarupplevelse som för nuvarande system. Exempelvis antal klick för att nå funktioner	— ☑
7. Pekskärmsanpassad med tydliga knappar och menyer.	☑

Förvaltningsbarhet

8. Systemet skall fungera på en Windows 7 tablet med .NET 4.5.	☑
9. I största mån möjligt utforma applikationen för lättare plattforms-byte i framtiden.	— ☑

Dokumentation

10. Användarmanual på svenska.	☑
11. Användarmanual på engelska.	☑

Mobile Translator har en likvärdig användarupplevelse som Pocket Translator med lika många eller färre antal klick för att nå funktioner.

Mobile Translator har tydliga knappar och menyer med möjlighet för användare att själv justera teckenstorlek i applikationen efter önskemål.

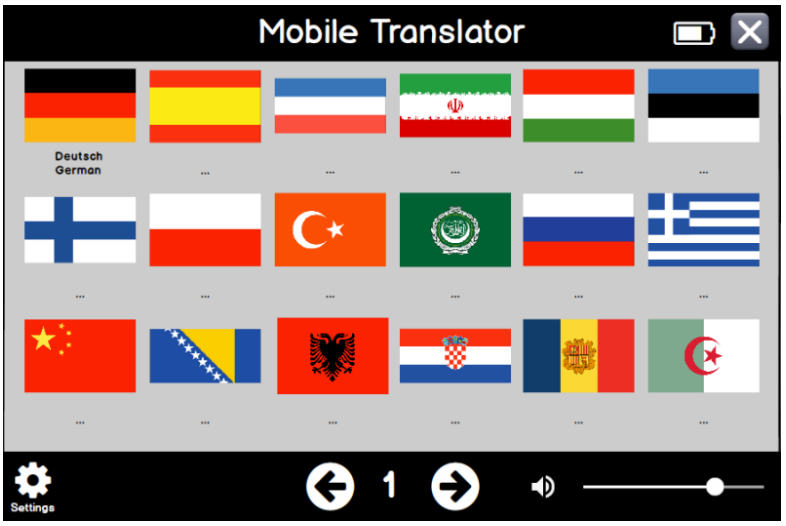
Mobile Translator kräver minst .NET 4.5 vilket är en rekommenderad uppdatering för Windows 7. Det är även möjligt att sänka detta till .NET 4.0 då det inte är många delar som använder den högre versionen av .NET 4.5.

Mobile Translator är skrivet enligt MVVM-mönstret i WPF vilket tidigare har beskrivits har som syfte att öka abstraktionen av arkitekturen och minska kontaktpunkter mellan objekt. Detta leder i sin tur till mer flexibel kod som är lättare att migrera till en annan plattform.

5.2 Resultat

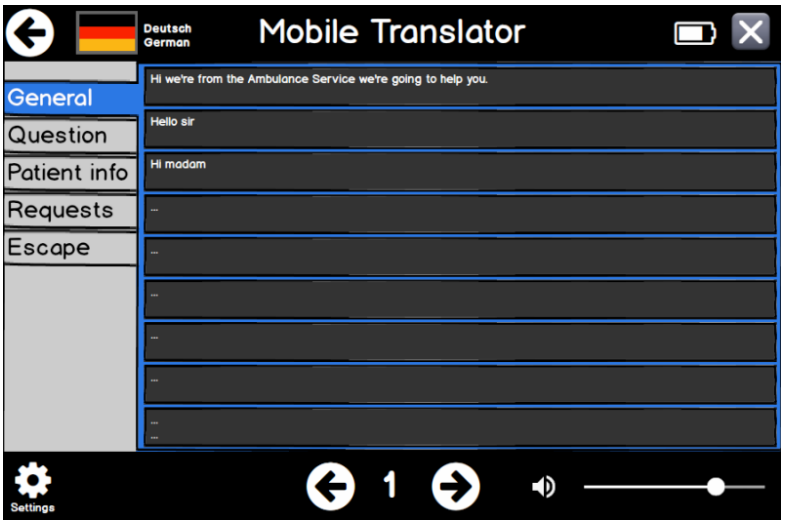
Projektet Mobile Translator har resulterat i en Windows applikation utformad för att ersätta arbetsgivarens tidigare produkt Paratus Pocket Translator som körts på den äldre mobila plattformen Windows Mobile. Mobile Translator underlättar för arbetsgivaren då applikationen körs på samma plattform som de flesta övriga produkter i Paratus-familjen.

Nedan presenteras tidiga skisser av användargränssnittet i Mobile Translator med beskrivning på vad som ändrats i det slutgiltiga resultatet.



FIGUR 5.1: Skiss av Mobile Translators uppstarts-vy (HomeView)

Skillnader mot figur 3.5: Tanken var först att ha vänster- och höger-pilar för att navigera i vyerna när innehållet inte fick plats. Dessa pilar ersattes istället av en rullningslist som visas på applikationens högersida. Antalet flaggikoner som visas i bredd blev ändrat till fem stycken istället för sex som på bilden. Anledningen till att vi sänkte antalet ikoner på bredden var att texterna under flaggikonerna blev för liten och svårläst.



FIGUR 5.2: Skiss av Mobile Translators fras-vy (PhraseView)

Skillnader mot figur 3.7: Flaggikonen uppe till vänster som indikerar vilket språk som valts i föregående vy innehåller enbart flaggbilden i Mobile Translator. Texterna bredvid flaggan valdes bort för att spara plats. Knapparna blev uppradade

i två kolumner istället för en som i bilden ovan. Anledningen till detta var att det bättre använde skärmytan då vissa texter kunde vara väldigt korta.

Settings-vyn fanns inga planer på i ett tidigt skede då det var osäkert på vad användare skulle kunna ändra. Slutresultatet blev att användaren kan styra fontstorlek genom ett skjutreglage och visningsspråk med knappar.

5.3 Tekniker och verktyg

I detta avsnitt utvärderas de tekniker och verktyg som används under projektets gång. För utveckling har Visual Studio 2015 använts tillsammans med ramverket .NET Framework 4.5. Ramverket .NET Frameworks delar WPF och XAML utvärderas tillsammans med utvecklingsspråket C#. Utöver dessa delar utvärderas designmönstret MVVM, versionshantering med Git och övriga verktyg som används vid utveckling eller skrivandet av denna rapport.

5.3.1 C# och .NET Framework

Mobile Translator har helt utvecklats i språket C# för .NET Framework 4.5. Inga speciella problem har uppstått under projektet med språket eller ramverket utan det har flutit på bra. De gånger frågor har uppstått har svar varit lätt att finna då det är ett populärt utvecklingsspråk. Användargränssnittet är framtaget med tekniken WPF som är en del av ramverket .NET Framework och som i sin tur antingen använder C# eller XAML. Valet här föll på att använda XAML efter inrådan från arbetsgivaren. Vi kan konstatera att vi gillar både språket C#, XAML och ramverket .NET Framework även om XAML tog lång tid att lära sig.

5.3.2 MVVM

Utvecklingen av Mobile Translator har helt formats efter designmönstret MVVM vilket tidigare beskrivits omfattande (se 3.1). Valet att följa detta designmönster var även det efter inrådan från arbetsgivaren. Vi gillar designmönstret även om det tillsammans med XAML är det som tagit mest tid att lära sig.

5.3.3 Visual Studio

All utveckling har skett i Visual Studio 2015 som är en programutvecklingsmiljö ifrån Microsoft. Visual Studio har en kodeditor som stödjer IntelliSense, vilket ger förslag på vad man försöker skriva efter att man bara skrivit in några få tecken. Andra integrerade verktyg i Visual Studio som använts flitigt är designern och debuggern.

Designern i Visual Studio är en funktion som visar upp hur användargränssnittet ser ut utan att behöva bygga och köra projektet varje gång något ändras. När användargränssnittets XAML kod uppdateras så speglas detta i designern direkt. Designern har underlättat utvecklingen mycket då det hade tagit mycket längre tid att testköra projektet varje gång man exempelvis flyttat någon grafisk komponent litetgrann.

Debuggern i Visual Studio pekar på vad som gått fel i en applikation när en krasch uppstår. Det är även möjligt att sätta upp breakpoints, vilket innebär att man säger till applikationen att pausa exekveringen på en specifik kodrad. Det är sedan möjligt att stega genom kodrad för kodrad och se vad som händer. Värden på variabler är även möjligt att se när man debuggar genom att hålla muspekaren över dessa.

Visual Studio är enligt oss en komplett och väldigt välfungerande utvecklingsmiljö.

5.3.4 Versionshantering

För versionshantering av Mobile Translator har vi använt oss av versionshanterings-programmet Git. Som Git-klient har vi använt både Visual Studios integrerade git-klient och TortoiseGit. Primärt användes TortoiseGit som integrerades i Windows utforskare då denna var snabbare och gav mer kontroll vid eventuella konflikter.

Git var nytt för oss då vi tidigare enbart använt snarlika verktyg som SVN (Apache Subversion) och TFS (Microsoft Team Foundation Server). Git hade däremot en stor förbättring emot dessa som vi fick uppleva under projektets gång. Denna förbättring är att man har en kopia av kodrepot lokalt som man kan spara sina ändringar i under arbetets gång. När man sedan vill kan man skicka alla dessa ändringar till det centraliserade kodrepot. Det som hände under projektet var att

den server där vårt centraliserade kodrepo låg fick en hårddisk-krasch. Eftersom vi hade en lokal kopia av kodrepot kunde vi utan svårigheter sätta upp ett nytt centraliserat kodrepo. Vi kan sammanfatta med att vi gillar Git även om vi enbart föredrar att köra med en grafisk git-klient och inte direkt ifrån terminalen. Anledningen till att vi föredrar en grafisk git-klient är att detta känns mer familjärt.

5.3.5 Balsamiq

För att rita skisser på användargränssnittet till Mobile Translator användes programmet Balsamiq. Detta program ger möjlighet att direkt från webbläsaren eller en applikation rita upp användargränssnitt med fördefinierade komponenter genom drag- och släpprincipen. Möjligheten fanns också att ladda upp egna bilder som kunde användas på samma sätt. Vi gillade att man kunde köra Balsamiq direkt ifrån webbläsaren och det gick snabbt att skapa en skiss med alla fördefinierade komponenter.

5.3.6 Audacity

Audacity är ett ljudredigeringsprogram som användes för att arbeta med applikationens ljudfiler. Genom Audacity kunde vi först markera de bitar i ljudfilen innan och efter att uppläsaren läste upp en fras för att identifiera oljud så som buller, brus och tjut. Efter att identifierat dessa oljud kunde dessa oljud tvättas bort ifrån den delen där uppläsaren pratade. Vidare kunde vi klippa bort dessa bitar innan och efter något sades. Slutligen normaliserades alla ljudfiler, vilket innebär att maxljudet i samtliga ljudfiler sattes till det samma så att de uppfattas som att det är samma volym på inspelningarna. Vi är mycket nöjda med detta program och det fanns enkla instruktioner till det.

5.3.7 Google Dokument

För rapportskrivning har Google Dokument använts via en webbläsare. Google Dokument valdes primärt för att det är något vi är väl bekanta med och det ger oss möjlighet att tillsammans skriva i samma dokument och se ändringar direkt. Man kan lätt se i Google Dokument vad den andra jobbar med genom markeringar

i texten vart denna har sin textmarkör. I det stora hela gillar vi Google Dokument men vi är missnöjda med rättstavningen då den inte håller tillräckligt hög standard som en rapportskrivning kräver.

5.3.8 ShareLaTeX

För att formatera vår rapportskrivning har vi använt oss av LaTeX genom tjänsten ShareLaTeX. ShareLaTeX fungerar likt Google Dokument direkt i webbläsaren där man tillsammans med andra kan jobba i samma dokument. I ShareLaTeX har man två fönster där det ena är en LaTeX-editor och det andra är en kompilerad PDF utifrån LaTeX koden. Vi har haft en del problem med att formatera vår rapport i ShareLaTeX där exempelvis placering av figurer har varit svårt. En figur i LaTeX är nämligen något som är flytande, vilket innebär att den försöker lägga in figuren i närheten av där vi specificerat men inte exakt beroende på vad LaTeX själv anser när den kompilerar rapporten. De problem vi haft är framförallt för att vi inte tidigare använt LaTeX och hade vi istället gjort det är vi säkra på att vi hade gillat ShareLaTeX då detta i sig inte tillfört några problem. ShareLaTeX är enligt oss därför en väldigt bra tjänst och det introducerar nya användare med flera mallar och väldokumenterade guider.

Kapitel 6

Utvärdering av projektet

I detta kapitel görs en utvärdering av projektet. Redogörelser som tidsåtgång och arbetsmiljö tas upp samt lärdomar och framtida vidareutveckling av projektet.

6.1 Tidsåtgång

Vi la mer tid på utvecklingen i början av projektet än uppsatsskrivande, så vi skulle ha något konkret att skriva om i uppsatsen. Vi ville vara säkra på att vi följde designmönstret först så att vi kunde dokumentera så nära slutprodukten som möjligt och inte redigera uppsatsen så fort vi gjorde en ändring i applikationen.

- Första månaden (februari) jobbade vi nästintill heltid på projektet, utvecklingen stod för ungefär 90% av tiden.
- Andra månaden (mars) jobbade vi halvtid och då jämnades tiderna ut till cirka 60% utveckling och 40% uppsatsskrivning.
- Tredje månaden (april) jobbade vi fortfarande halvtid och då gick cirka 70% av tiden till utveckling och 30% till uppsatsskrivning.
- Sista månaden (maj) gick 100% av tiden till uppsatsskrivning.

I och med att vi hade mer tid för projektet i början kunde vi försäkra oss om att vi fick en god start enligt planeringen. I april hade vi en demonstration av projektet för medarbetarna på Saab i Karlstad och utifrån återkoppling där behövdes

några ändringar göras. Ett exempel på en ändring var implementationen av SingleInstance, vilket förhindrade applikationen från att köras i fler instanser än en. I maj månad fick vi lägga all resterande tid åt uppsatsskrivningen för att komma ikapp då vi låg ett kapitel efter planeringen.

6.2 Arbetsmiljö

All utveckling har skett på Saabs kontor i Karlstad som ligger i stadsdelen Klara. Vi fick ett kontorsrum där vi hade tillgång till datorer med utvecklingsmiljöer och Windows-plattor för testning. Att sitta på företagets kontor underlättade då vi fick bra stöd gällande design, utveckling samt versionshantering. I och med att vi satt på ett lokalt kontor hade vi närhet till personal på Saab som utvecklade med samma tekniker vi använde oss av i projektet. Personalen hjälpte oss även testa produkten under utvecklingen och komma med synpunkter och råd. Uppsatsskrivningen skedde mestadels hemifrån enskilt och en del med kommunikation via Skype. Fördelen med att skriva enskilt har varit att fokusfaktorn blev generellt sätt högre med mindre distraheringar. Efter vi skrivit enskilt kunde vi gemensamt diskutera och renskriva det vi skrivit.

6.3 Kravspecifikation

I början av projektet fanns ingen färdig kravspecifikation utan denna arbetades ut tillsammans med arbetsgivaren. Kärnkravet var att all tidigare funktionalitet från Paratus Pocket Translator skulle finnas i den nya applikationen vilket inte innebar några problem då vi hade tillgång till en körbar version av produkten och dess källkod.

Vissa av de övriga kraven vi tagit fram insåg vi däremot vid genomgång av kravuppfyllelse att de inte var mätbara eller att de inte var helt tydliga. Vi fick gå igenom dessa krav igen med vår handledare på Saab i början på maj för att förtydliga dem. I slutändan hade vi uppnått ett resultat som förväntats utifrån kravspecifikationen med små justeringar av användargränssnittets skalning då det även skulle fungera på en enhet med 800x600 i upplösning.

6.4 Lärdomar

I projektet har vi stött på en del tekniker och verktyg som varit nya för oss. Dessa tekniker och verktyg var WPF, XAML, MVVM och Git vilket vi efter avslutat projekt känner att vi fått goda kunskaper inom.

Utöver de tekniska lärdomarna har vi även lärt oss vikten av att inte skynda in i utvecklingen utan en färdigställd kravspecifikation.

6.5 Möjligheter till vidareutveckling

I vår kravspecifikation så finns kravet ”I största mån möjligt utforma applikationen för lättare plattforms-byte i framtiden.”. Detta krav är inte speciellt mätbart. Här skulle man kunna försöka flytta projektet till Xamarin, vilket är ett verktyg som inkluderas i Visual Studio för plattformsöverskridande-utveckling.

Det fanns också önskemål från arbetsgivaren om att införa stöd för text-to-speech i applikationen. Detta skulle vara en god vidareutveckling då behovet av nya förinspelade fraser i applikationen skulle minska.

6.6 Avslutning

Vi är mycket nöjda med projektet hos Saab i Karlstad. Projektet har varit både roligt och givande och vi har fått mycket bra hjälp både från vår handledare och övriga medarbetare. Utöver detta har vi också fått erfarenhet i flera nya verktyg och tekniker.

Vår arbetsgivare är mycket nöjd med projektet och vi avslutar med ett citat från vår handledare Carl-Henric Smedman på Saab:

”I och med detta arbete kan vi lämna den utgående Windows Mobile plattformen bakom oss och säkerställa att även denna produkt kan användas på de plattformar som produktfloran numera använder. Produkten har samtidigt fått en mer hållbar design, vilket har gjort den mer framtidssäker och inbjudande att bygga ut med ytterligare funktionalitet i framtiden.” .

Referenser

[1] Audacity. *Audacity*. URL: http://www.audacityteam.org (hämtad 2016-05-11).

[2] Software Freedom Conservancy. *Git*. URL: https://git-scm.com (hämtad 2016-02-29).

[3] C-sharpcorner. *RelayCommand*. URL: http://www.c-sharpcorner.com/UploadFile/20c06b/icommand-and-relaycommand-in-wpf (hämtad 2016-04-19).

[4] Grant Frisken. *WPF Localization Using RESX Files*. URL: http://www.codeproject.com/Articles/35159/WPF-Localization-Using-RESX-Files (hämtad 2016-02-24).

[5] Google. *Google Dokument*. URL: https://www.google.com/intl/sv/docs/about (hämtad 2016-05-11).

[6] John Garland och Jeremy Likness Jeremy Likness. *Programming the Windows Runtime by Example*. Addison-Wesley Educational Publishers Inc, 2014, kapitel 9, sida 362.

[7] Microsoft. *Commands, RelayCommands and EventToCommand*. URL: https://msdn.microsoft.com/en-us/magazine/dn237302.aspx (hämtad 2016-04-28).

[8] Microsoft. *Garbage Collection*. URL: https://msdn.microsoft.com/en-us/library/0xy59wtx(v=vs.110).aspx (hämtad 2016-04-26).

[9] Microsoft. *PlaySound*. URL: https://msdn.microsoft.com/en-us/library/dd743680(VS.85).aspx (hämtad 2016-05-02).

[10] Microsoft. *.RESX*. URL: https://msdn.microsoft.com/en-us/library/ekyft91f(v=VS.90).aspx (hämtad 2016-03-31).

[11] Microsoft. *Visual Studio*. URL: https://www.visualstudio.com (hämtad 2016-02-24).

[12] Microsoft. *XAML*. URL: https://msdn.microsoft.com/en-us/library/cc295302.aspx (hämtad 2016-03-17).

[13] Arik Poznanski. *Singleinstance*. URL: http://blogs.microsoft.co.il/arik/2010/05/28/wpf-single-instance-application (hämtad 2016-04-21).

[14] ShareLaTeX. *ShareLaTeX*. URL: https://www.sharelatex.com (hämtad 2016-05-11).

[15] Saab Solutions. *About*. URL: http://saabgroup.com/about-company/company-in-brief (hämtad 2016-02-24).

[16] Saab Solutions. *Locations*. URL: http://saabgroup.com/career/people-philosophy/locations (hämtad 2016-02-29).

[17] Saab Solutions. *Paratus*. URL: http://saab.com/security/police-and-rescue/emergency-response/paratus (hämtad 2016-02-29).

[18] Balsamiq Studios. *Balsamiq*. URL: https://www.balsamiq.com (hämtad 2016-05-11).

[19] TortoiseGit. *TortoiseGit*. URL: https://www.tortoisegit.org (hämtad 2016-06-13).

[20] Wikipedia. *Arabförbundet*. URL: https://sv.wikipedia.org/wiki/Arabf%C3%B6rbundet (hämtad 2016-04-19).

[21] Wikipedia. *CLR*. URL: https://en.wikipedia.org/wiki/Common_Language_Runtime (hämtad 2016-03-01).

[22] Wikipedia. *C-sharp*. URL: https://sv.wikipedia.org/wiki/C-sharp (hämtad 2016-06-13).

[23] Wikipedia. *ISO 639-1*. URL: https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes (hämtad 2016-03-17).

[24] Wikipedia. *MVVM*. URL: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93viewmodel (hämtad 2016-02-24).

[25] Wikipedia. *.NET*. URL: https://sv.wikipedia.org/wiki/.NET_Framework (hämtad 2016-03-01).

[26] Wikipedia. *Singleton*. URL: https://sv.wikipedia.org/wiki/Singleton (hämtad 2016-04-21).

[27] Wikipedia. *WPF*. URL: https://en.wikipedia.org/wiki/Windows_Presentation_Foundation (hämtad 2016-02-24).

[28] Wikipedia. *XAML*. URL: https://en.wikipedia.org/wiki/Extensible_Application_Markup_Language (hämtad 2016-03-01).

[29] WPFtutorial. *Introduction to WPF*. URL: http://www.wpftutorial.net/XAML.html (hämtad 2016-05-02).