CHALMERS

Refaktorering och vidareutveckling av bokläsarapplikation

Att sätta sig in i och förbättra ett främmande projekt

Examensarbete inom högskoleingenjörsprogrammet Dataingenjör

JOHAN ÖRN
JONATAN STRÖMSTEN

Institutionen för data- och informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige, 2012

Sammanfattning

I dagens samhälle där applikationer och användande av datorrelaterade produkter ökar, ökar också strävan efter att göra framtidssäkra och strukturellt hållbara produkter. Programmerare förväntas ofta ta sig an ett större projektarbete och att leverera under en kort tid då tekonologier är en färskvara. Med denna stress blir ofta programmet därefter och en del lösningar kan fungera, men inte alls vara genomtänkta eller framtidssäkra. Projektet utgörs av modernisering av en produkt med syfte att göra den mer tilltalande och användbar. Produkten är en windows applikation som kan läsa elektroniska böcker, där det just nu finns stöd för vanlig läsning samt ljuduppspelningen av meningar. Innan programmet betraktas som färdigt behöver ett antal nya funktioner implementeras, vilka innefattar bland annat bokmärken, videouppspelning och en sökfunktion. Utöver detta så finns även behov av en ny design och bättre layout. Innan de nya funktionerna kan implementeras så görs en analys av den nuvarande kodbasen, innefattande en genomgång av applikationens arkitektur och hållbarhet inför implementering av nya funktioner. En genomgång av befattningen god kodstandard diskuteras och hur problemen för den här applikationen identifierades genom en omfattande analys och senare förändrades.

Abstract

In todays society where applications and computer-related products increases, striving to make future safe and structurally further development of marketable products is at the essence. Programmers are often expected to take on major projects and deliver the solution in a short period of time since technologies is perishable. With this stress in mind the solution does work, but is not thourogly bethought for the future of the program. The project consists of a modernization of a product to make it more appealing and useful. The product is a Windows application that can read electronic books, which originally has support for regular reading and audio playback of sentences. Before the program is considered to be completed a number of new features have to be implemented including bookmarks, video playback and a search function. In addition there is also need for a new design and improved layout. Before the new features can be implemented an analysis of the current code base will be done, including a review of the applications architecture and sustainability for the implementation of new features. A review of the concept good code standard will be discussed and how the problems for this application was identified by a comprehensive analysis and subsequently changed.

Innehållsförteckning

Figurförteckning

1. Inledning

1.1 Bakgrund

Företaget har utvecklat en applikation som kan läsa digitala böcker. Applikationen riktar sig främst till personer som har någon form av läs-eller inlärningssvårighet, men är även tänkt att kunna användas för att läsa vanliga böcker.

Programmet har just nu stöd för grundläggande funktioner så som sidbläddring, kapitelbläddring, bildvisning och olika textstilar. Utöver detta finns även en funktion för direktuppläsning av text där användaren kan klicka på en del i texten för att få den uppläst. Företaget vill att applikationen ska uppdateras med nya funktioner och användarmöjligheter som konfigurerbart färgschema, videospelning, animationer, bokmärken, sökfunktion och annoteringar. En uppdatering och modernisering av gränssnittet är också önskad.

1.2 Problem

- Kan den nuvarande strukturen bära upp nya funktioner?
- Är det fördelaktigt att strukturera om inför implementering av nya funktioner?
- Bidrar en omstrukturering till applikationens framtidssäkerhet och skalbarhet?

1.3 Syfte

Syftet är att vidareutveckla applikationen, vilket kommer att innebära implementering av nya funktioner. Målet är att se till så att programmet är stabilt för framtiden och att det blir möjligt att införa nya funktioner även senare utan att behöva göra större omskrivningar av programmet.

1.4 Mål

- Att göra applikationen framtidssäker.
- Att uppdatera applikationens funktionalitet.

1.5 Avgränsningar

Det kommer inte att göras en djup analys av hur funktionerna i programmet är uppbyggda, utan enbart en analys för att skapa skalbarhet och för att kunna implementera nya funktioner. Det finns även en applikation där böckerna kan skapas och redigeras, ändringar på detta program kommer inte att göras.

1.6 Disposition

Rapporten inleds med definitioner av vanligt förekommande ord och begrepp, därefter följer ett kapitel som ämnar ge en teoretisk bakgrund. Vidare beskrivs metoden följt av genomförandet som är uppdelat i
analys, förslag på ändringar, refaktorering och implementering av nya funktioner. Rapporten avslutas sedan med en beskrivning av resultatet efter analysen och ändringarna följt av en slutsats.

2. Definitioner

Användargränssnitt Ett (grafiskt) användargränssnitt utgör den grafiska presentationen av en applikation, till exempel fönster och knappar. [12]

C# Ett programspråk som utvecklades av Microsoft tillsammans med .NET plattformen. [1]

Dataladdare En dataladdare är en komponent som har i uppgift att tolka komplex data och leverera simplare klasser som går att använda i övriga delar av programkoden. [29]

Domänmodell Domänmodellen utgörs av entiteter som beskriver datadefinitionen för ett problem. Domänen brukar separeras från logiken som bearbetar data och även från hur datan lagras, detta medför att entiteter inte känner till hur de lagras eller används. [35]

DropBox Är ett enkelt versionshanteringssystem som används för att dela filer och mappar. [6]

Enhetstest Programkod som testar en isolerad funktionalitet av ett program. [34]

Entitet En entitet är en beskrivning av ett objekt. Entiteten *Bil* skulle till exempel kunna ha flera olika attribut som ett *namn*, en *färg* och en *årsmodell*. [33]

Funktion Funktioner används i programmering för att dela upp program i mindre bitar så att koden blir mer överskådlig och lätthanterlig. Funktioner gör också att samma funktionalitet inte behöver skrivas på flera platser. [32]

Git Är ett versionshanteringssystem som tillåter utvecklare att arbeta, dokumentera och dela ändringar på samma projekt på ett smidigt sätt. [5]

Inkrement Utgörs av flera iterationer och definierar ett tydligt delmål i projektet. [20]

Iteration Den tid som löper under ett delmåls utförande. [20]

Klass Är en mall för objekt. En klass kan innehålla tillstånd, attribut och funktioner. Tillståndet beskriver oftast objektets nuvarande läge och värden på olika attribut, när en funktion hos ett objekt anropas kan detta ändra tillstånd. En klass som beskriver objektet *Bil* kan innehålla attribut för *färg* och status *nuvarande färg*, vidare kan det finnas en funktion *ändra färg till x* som ändrar statuset *nuvarande färg*. [30]

Konverterare En konverterare används för att konvertera från en klasstyp till en annan. [21]

Kryptering Kryptering är en teknik som används för att dölja känslig information när den på något sätt kan vara åtkomlig för obehöriga. Informationen ska inte vara tillgänglig i klartext utan en dekrypteringsnyckel. [13]

Mockningsverktyg Används för att simulera delar av programkoden. [19]

Modul	En modul är en isolerad del i en applikation som exponerar utvalda funktioner, ofta relaterade. [10]
MVC	Ett designmönster som används vid strukturering av användargränssnitt. [26]
MVVM	Ett designmönster som utvecklades samtidigt som WPF och som rekommenderas av Microsoft vid utveckling av bland annat WPF-applikationer. [9]
Plattform	En mjukvaruplattform är en applikation som kan handhava och köra andra applikationer. [14]
Ramverk	Är en större samling av moduler och verktyg som kan användas vid utveckling av applikationer eller andra ramverk. [31]
Textobjekt	Objekt som visas tillsammans med texten när man visar en bok i applikationen.
Versionshantering	Versionshanteringssystem låter användare välja filer eller mappar och spara en version tillsammans med en beskrivning vid en utvald tidpunkt. Det går sedan att nå de olika versionerna av filerna och mapparna genom systemet. [11]
WPF	Ett ramverk som ingår i .NET och används för att utveckla applikationer till Windows. [4]
XML	Är ett språk som används för att beskriva och definiera data. [2]
.NET	En plattform och ett ramverk som Microsoft har utvecklat för Windows. I .NET ingår verktyg för utveckling av flertalet olika sorters applikationer. [3]
ZIP-arkiv	Ett filformat där flera logiska filer kan lagras i en och samma fysiska fil på disk. [25]

3. Teori

3.1 Gränssnitt

Ett gränssnitt *(eng. Interface)* är ett sätt för en modul eller klass att uppfylla ett kontrakt. Kontraktet definierar utvalda funktioner som kontrakttagaren måste erbjuda men specifierar inga implementeringsdetaljer. Om en komponent är i behov av funktionerna som ett bestämt kontrakt exponerar kan denna komponent arbeta enbart mot kontraktet utan att vara medveten om eller beroende av hur funktionerna är implementerade. Ponera följande kontrakt för *bil*:

- En bil ska ha en färg.
- En bil ska gå att starta.
- En bil ska kunna köras.
- En bil ska kunna stanna.

När en komponent är i behov av en *bil* är information om *hur* bilen startar eller *varifrån* bilens färg kommer i sammanhanget inte väsentligt utan komponenten är enbart intresserad av denna utvalda funktionalitet. Kontraktet kan uppfyllas och implementeras på flera olika sätt och av många olika sorters *bilar*. Det är till exempel tänkbart att en biluthyrningsverksamhet hyr ut bilar där kunden enbart begär en körbar bil i en viss färg, valet av bilsort görs då istället av uthyrningsverksamheten. [28]

3.2 Designmönster

Ett designmönster är ett vedertaget sätt att lösa ett vanligt förekommande problem inom mjukvaruutveckling. Ofta definierar designmönster bestämda, men inte nödvändigtvis strikta, sätt att arrangera olika moduler och funktionaliteten inom dessa. Tanken med detta är att programkoden ska bli mer modulär, skalbar och lättförståelig genom en minskad *koppling* mellan komponenter och ökad *kohesion* inom dem.

Ett vanligt förekommande problem inom mjukvaruutveckling är att skapa separation mellan ett grafiskt användargränssnitts utseende och logiken som utgör applikationens funktionalitet. För att lösa detta problem på ett skalbart sätt finns ett antal olika designmönster, däribland *MVC* och *MVVM* [9]. Applikationer som är i behov av mer avancerad datahantering och lagring brukar använda designmönster som *Treskiktsmodellen* och *Datamagasin* för att separera hur data *används* från hur den *lagras* [26].

3.3 Koppling

Koppling är nära relaterat till *Kohesion* och beskriver *till vilken grad* en komponent är beroende av andra komponenter. Låg koppling mellan komponenter och moduler eftersträvas då detta medför att:

- En modul kan testas i isolation – de andra modulerna behöver inte inkluderas för att testerna ska kunna genomföras.
- Utbyte eller uppdatering av en modul kan resultera i små eller obefintliga ändringar i andra moduler. I motsats så krävs vid hög koppling istället stora ändringar i flertalet moduler.

[18]

Det finns många olika sätt att strukturera programkod för att ge en låg koppling mellan komponenter, oftast tillämpas designmönster för att lösa problem då hög koppling uppstår. Vid mätning av *kopplingsgrad* är flera variabler av betydelse:

d_i Antalet datarelaterade parametrar som förs in i modulen.
c_i Antalet kontrollparametrar som förs in i modulen.
d_o Antalet datarelaterade parametrar som modulen returnerar som resultat av operationerna den utför.
c_o Antalet kontrollparametrar som modulen returnerar.
g_d Antalet globala variabler som används för att lagra data.
g_c Antalet globala kontrollvariabler.
w Antalet moduler som kallas av modulen vars *kopplingsgrad* ska beräknas.
r Antalet moduler som kallar modulen (motsatt beroende jämfört med *w*).
[27]

$$Koppling = 1 - \frac{1}{d_i + 2c_i + d_0 + 2c_0 + g_d + 2g_c + w + r}$$

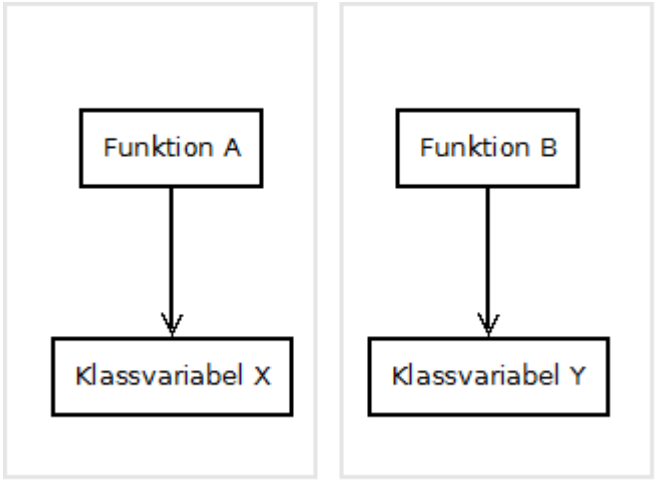
Figur 1: Formel för att mäta grad av koppling

Där resultatet är ett värde mellan 0.67 och 1.0, ett högre värde innebär att modulen har hög koppling. Ett sätt att minska detta värde vore till exempel att hitta en lösning på problemet som inte kräver användande av globala variabler. [15][16]

3.4 Kohesion

Medan *Koppling* mäter hur självstående en modul är så mäter *Kohesion* hur relaterade funktionerna inom modulen är till varandra. En väldesignad modul har funktioner som utför liknande arbete, detta medför att moduler har tydliga arbetsuppgifter *(från eng. Separation of Concerns)*. Ett exempel skulle kunna vara komponenten *Bil* som förväntas ha operationer för körning. Om en *Bil* också skulle behöva leverera temperaturdata är risken att bilen frångår sin centrala uppgift om den direkt implementerar denna funktionalitet. Problem liknande detta brukar lösas genom bland annat *Beroendeinvertering*. [23]

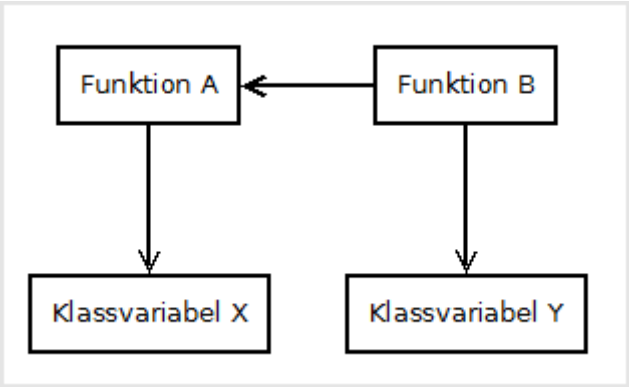
Kohesion kan mätas genom olika *LCOM*-metoder *(från eng. ”Lack of Cohesion of Methods”)*, som i princip mäter avsaknad av koppling mellan funktioner. Två funktioner anses vara kopplade till varandra om de båda använder samma klassvariabel eller om den ena funktionen använder den andra. *LCOM4*-metoden mäter kohesion genom att räkna antalet funktionsgrupper där en grupp innehåller funktioner och variabler som inte är kopplade till någon annan grupp.



Figur 2: Låg kohesion mellan komponenter enligt LCOM4

I fallet då två funktioner använder varsin klassvariabel men inte varandra skapas två grupper och *LCOM4* ger därmed ett värde på 2 medan värdet helst ska vara 1. En lösning på detta vore att

antingen dela upp funktionerna och variablerna i separata klasser med tydligare uppgifter eller skapa en icke-trivial koppling mellan grupperna:

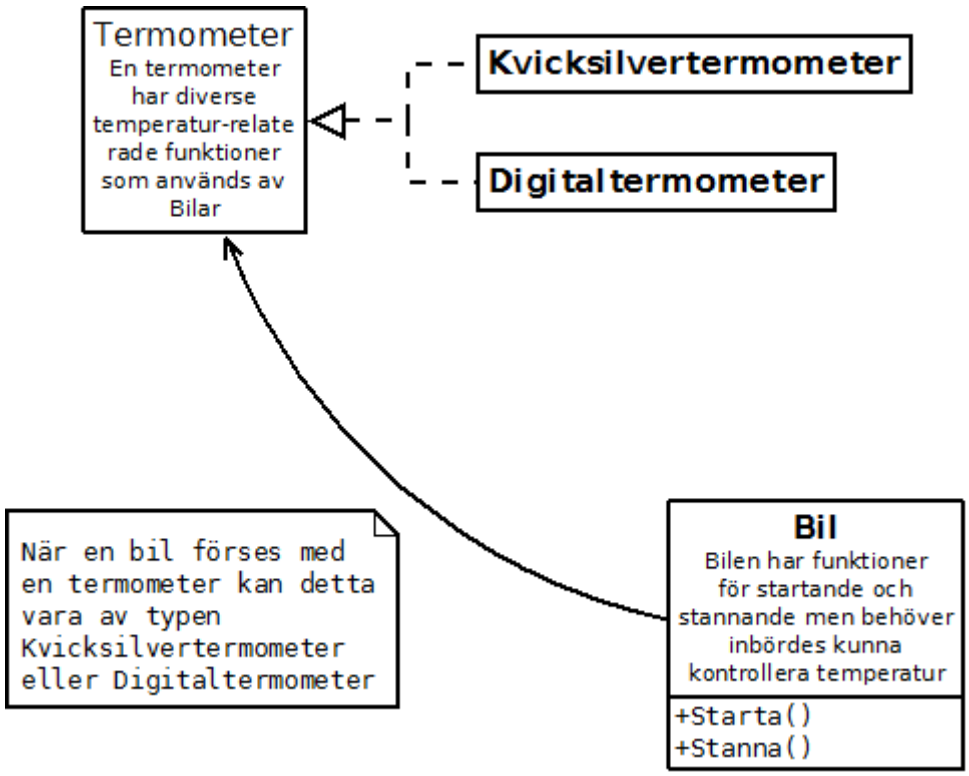


Figur 3: Hög kohesion mellan komponenter enligt LCOM4

På detta sätt skapas istället en ensam grupp då *Funktion B* sammanbinder grupperna genom att använda *Funktion A*. Det finns även andra metriker för mätning av kohesion, vilka innefattar *LCOM1, LCOM2, LCOM3* och *TCC och LCC*. [17]

3.5 Beroendeinvertering

Från *eng. Inversion of Control*. Detta är ett designmönster som har utvecklats för att flytta funktionalitet från klasser där funktionaliteten inte anses tillhöra klassens ansvarsområde. Ofta förses en klass med en implementering av ett gränssnitt utan att känna till hur den underliggande implementeringen ser ut och anropar sedan funktioner som exponeras av detta gränssnitt. Till exempel kan en *bil* vara i behov av att kontrollera utomhustemperaturen men behöver inte veta *hur* temperaturen kontrolleras, samtidigt är det inte en del av bilens ansvarsområde. Detta beroende kan omvändas genom att *bilen* förses med en klass som implementerar ett gränssnitt, *termometer,* som exponerar vissa bestämda funktioner för temperaturkontroller. Därefter anropas funktioner hos gränssnittet och den implementerande klassen skulle då kunna representera en kvicksilver- eller digitaltermometer.



Figur 4: Beroendeinvertering

Resultatet av detta är att *bilen* är oberoende av implementeringsspecifika detaljer, men även att den underliggande metoden kan bytas ut utan att *bilen* behöver känna till det. Det finns olika designmönster som implementerar beroendeinvertering, se Beroendeinjicering för en genomgång av ett av dem. [23]

3.6 Beroendeinjicering

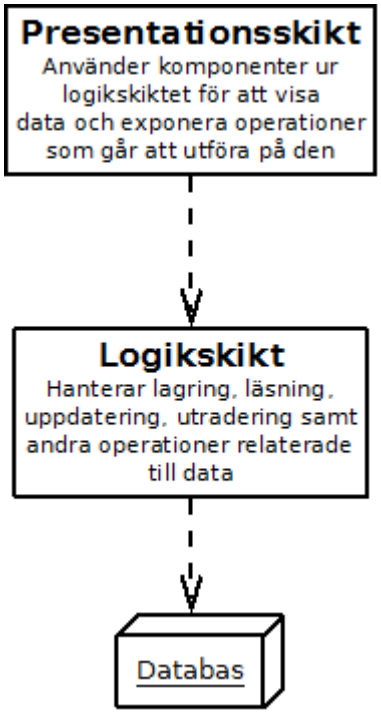
Från *eng. Dependency Injection*. Beroendeinjicering är ett sätt att förverkliga *Beroendeinvertering* genom att *injicera* gränssnittsimplementeringar i klasser utan att dessa är medvetna om processen som krävs för att skapa den implementerande klassen. [23]

3.7 Treskiktsmodellen

När applikationsdata behöver lagras i till exempel en databas eller som XML-filer kan det uppstå hög koppling mellan olika komponenter. Kopplingen skapas när logik som hanterar användargränssnittet utseende också hanterar logiken i anslutning till det, vidare visar användargränssnittet data från den underliggande databasen och har även en koppling till den. På detta sätt blir även kohesionen låg då endast en stor komponent står för flera olika arbetsuppgifter.

Figur 5: Struktur utan uppdelning av skikt

För att dela upp denna komponent till mindre komponenter med tydligare arbetsuppgifter kan en så kallad *treskiktsmodell* introduceras, där tre olika skikt utgör *Databas*, *Logik* och *Presentation*.



I databasen lagras den renodlade datan som kan komma från flera olika källor, detta kan vara i form av till exempel en relationsdatabas eller XML-filer.

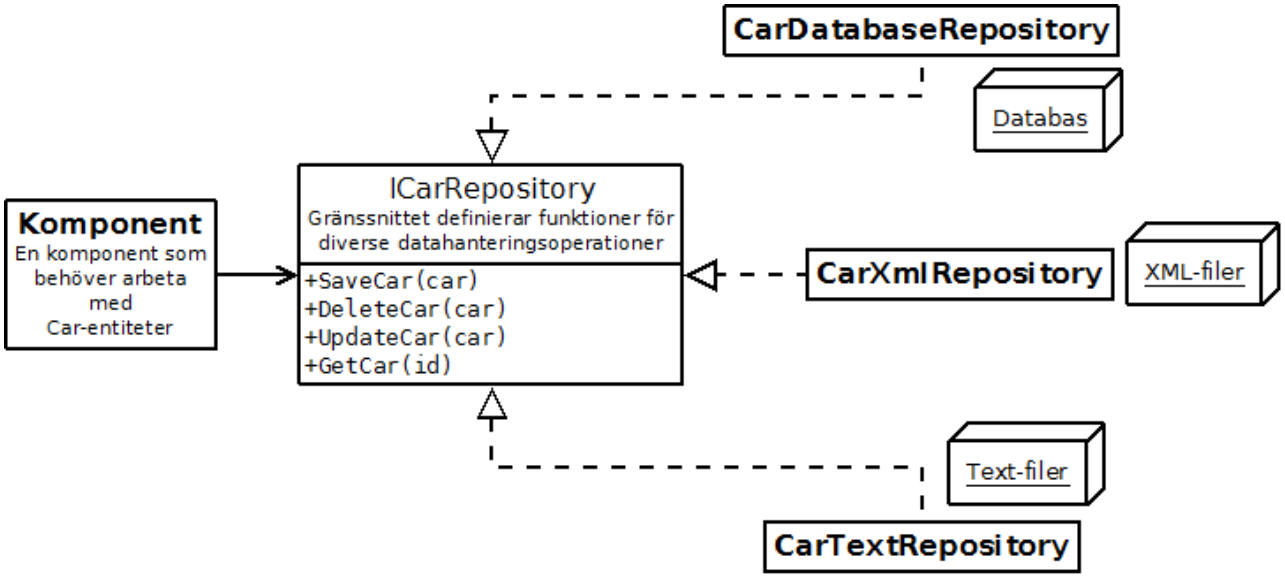
Närmast kontakt med databasen har logikskiktet, detta skikt har komponenter för dataåtkomst och olika dataoperationer. Logikskiktet fungerar som en mellanhand mellan presentationen och datalagret som frånkopplar presentationen från databasen. Datan representeras ofta i form av entiteter från domänmodellen [36]. Ett designmönster som används för att sköta datahanteringsoperationer är *datamagasin*, se nästa stycke för en genomgång av detta.

Presentationsskiktet känner inte till hur datan lagras eller hur de underliggande processerna ser ut. Vidare kan presentationsskiktet delas upp ytterligare genom mönster som *MVC* och *MVVM* för att separera utseende från funktion och datamodell.
[24]

Figur 6: Treskiktsmodellen

3.8 Datamagasin

Datamagasin utgör eller ingår ofta i *Logikskiktet* i *treskiktsmodellen* och brukar ha uppgiften att hantera datatransaktioner. Datamagasin brukar erbjuda operationer för lagring, hämtning, uppdatering och utradering av data, dessa operationer kan vara specifikt utformade för en bestämd entitet. Ett gränssnitt som definierar operationer för en specifik entitet kan till exempel heta *ICarRepository* och deklarerar då funktioner för hantering av *Car*-entiteter. Gränssnitt av denna typ skapar en abstraktion mot den underliggande datahanteringen och implementerande datamagasin kan arbeta mot olika sorters relationsdatabaser eller till exempel XML-filer:



Figur 7: Struktur vid användande av datamagasin

Komponent behöver inte känna till hur datan hanteras eftersom den enbart arbetar mot gränssnittet, den tunga datahanteringslogiken placeras då i de implementerande klasserna istället för i *Komponent*. [22]

3.9 Konverterare

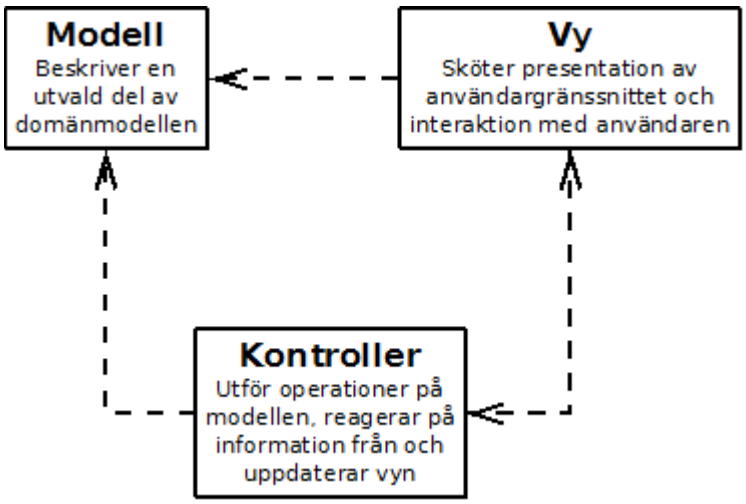
En konverterare är en klass med uppgift att konvertera objekt från en klasstyp till en annan. Ett exempel vore en klass som accepterar strängar och returnerar heltal: *StringToNumberConverter*. Denna klass skulle kunna konvertera strängar som *”one”* och *”thirteen”* till heltalen *1* och *13* respektive. [21]

3.10 Dataladdare

När en applikation är i behov av mer sofistikerade eller avancerade domän- och datamodeller kan så kallade *dataladdare* introduceras. Deras uppgift är att tolka data och konstruera objekt eller samlingar av objekt så att de lättare går att använda i andra delar av applikationen. En objektsamling skulle till exempel kunna beskrivas i en komplex XML-fil, risken blir i detta fall stor att det krävs tung logik för att läsa in och tolka filen. Denna logik kan i detta fall samlas i en separat *dataladdare* som sköter all tolkning och inläsning. [29]

3.11 MVC

Från *eng. Model View Controller*. MVC är ett designmönster som uppkom tillsammans med de första grafiska användargränssnitten med avsikt att skapa en tydlig separation mellan *Modell*, *Vy* och *Kontroller*.



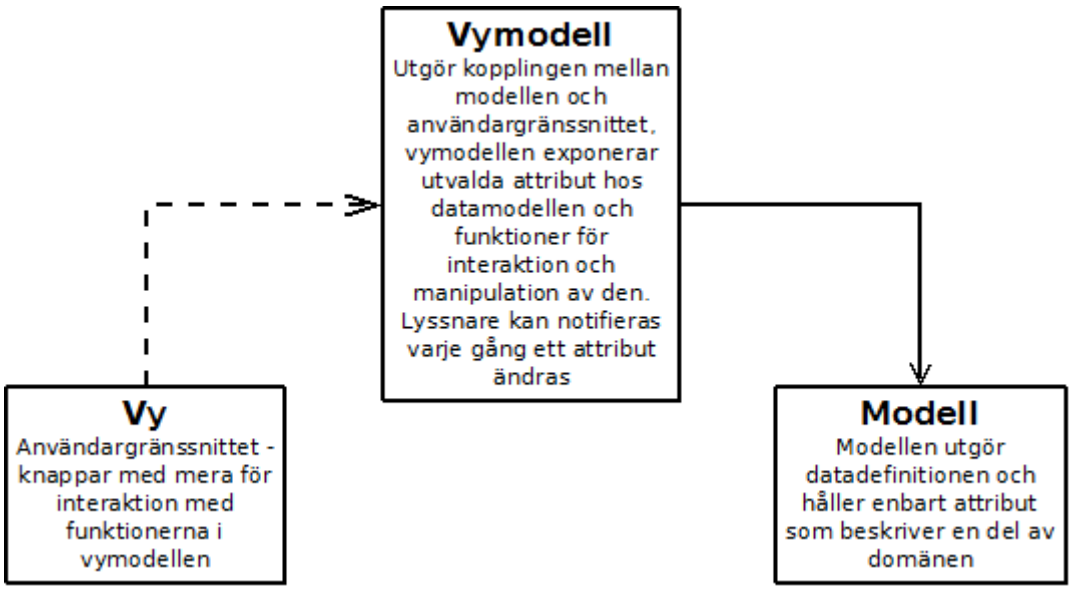
Figur 8: MVC-mönstret

Modell: En entitet, ofta från domänmodellen, som talar om vilka data som hanteras av den relaterade vyn och kontrollern.
Vy: Vyns uppgift är att visa data som levererats av kontrollern samt att förmedla användarinteraktioner till kontrollern.
Kontroller: Sammankopplar modellen och vyn. Kontrollerns uppgift är att tolka information som kommer från vyn, utföra operationer, uppdatera modellen och notifiera vyn.

Målet med denna uppdelning är att separera applikationens domänmodell, funktionalitet och presentation och ultimat erhålla lägre *koppling* samt högre *kohesion*.
[26]

3.12 MVVM

Från *eng. Model View ViewModel*. MVVM är en variant av MVC-mönstret som har blivit populär i samband med bland annat uppkomsten av WPF. Skillnaden gentemot MVC är att *Kontrollern* har eratts med en vymodell *(eng. ViewModel)* som representerar modellen, resultatet av detta är att kopplingarna mellan komponenterna är mer linjär där vyn känner till vymodellen som i sin tur har en modell.



Figur 9: MVVM-mönstret

En stor del i användandet av MVVM-mönstret är nyttjandet av den kraftfulla bindningsmotor som finns inbyggd i WPF. Detta brukar innebära att vyn är *bunden* till modellen via vymodellen och att när vymodellen på något sätt uppdaterar modellen så uppdateras också vyn automatiskt (uppdateringen är i praktiken inte automatisk utan hanteras av WPF:s bindningsmotor). [8][9]

3.13 Refaktorering

Refaktorering innebär i princip *omskrivning* av kod. Målet är i dessa fall inte att ändra kodens funktionalitet sett utåt, utan snarare dess underliggande struktur i syfte att förbättra kvalitén. Omskrivningen kan innebära en analys av *hur* applikationen är strukturerad samt eventuella brister och förbättringar i kodbasen. Resultat av refaktoreringen kan bland annat mätas genom metriker för *koppling* och *kohesion*. [7]

4. Metod

4.1 Daglogg
Dagloggar förs varje dag arbete har gjorts:

- Textfiler med paragrafer för varje dag med information om vad som har gjorts, datum och av vem.
- Filerna döps efter iterationerna (Iteration 1, Iteration 2, etc).

4.2 Backlog
Backlog förs inför varje iteration med information om vad som ska göras under den pågående iterationen.

4.3 Versionshantering
Dropbox används för delning av dokument, anteckningar och loggar. Här delas dokument och filer som inte är mål för versionshantering av olika anledningar, det kan till exempel vara stora filer eller tillfälliga anteckningar som inte är direkt relaterade till källkoden. Dropbox fungerar smidigt och är välintegrerat med Windows, det finns även en primitiv versionshantering inbyggd.

Git används som versionshanteringssystem för källkoden. Det kräver inte mycket lagringsutrymme och det är lätt att sätta upp lagringsmagasin.

4.4 Genomförande
Först görs en analys och genomgång av programkoden. Under analysen sammanfattas vad programmet gör och till viss del hur, här görs även en genomgång av vilka teknologier som används. Analysen ska gå igenom styrkor och svagheter hos den nuvarande kodbasen, vad som skulle behöva förbättras samt vilka, om några, förändringar som är nödvändiga inför införsel av nya funktioner. Förändringar och omstruktureringar görs därefter i syfte att ge programmet en hållbar struktur inför de planerade ändringarna, men även inför eventuella framtida uppdateringar. Vidareutvecklingen innefattar skrivning av programkod för att leverera av företaget önskade funktioner. Detta arbete kommer att ske i små iterationer om veckor, där varje iteration avslutas med presentation för företaget tillsammans med feedback och diskussion om nästa steg.

5. Analys av programmet och dess komponenter

5.1 Varför analys och refaktorering är relevant

- Det är ett sätt att lära känna den befintliga koden som utgör applikationen.
- Bedöma om det går att vidareutveckla och implementera nya funktioner med den nuvarande kodbasen.
- Åtgärda fel som står i vägen för vidareutvecklingen.
- Göra programmet mer framtidssäkert och skapa möjlighet för fortsatt arbete med det.

Utöver detta så kan en refaktorering och förbättring av kodbasen innebära att implementering av nya funktioner blir smidigare.

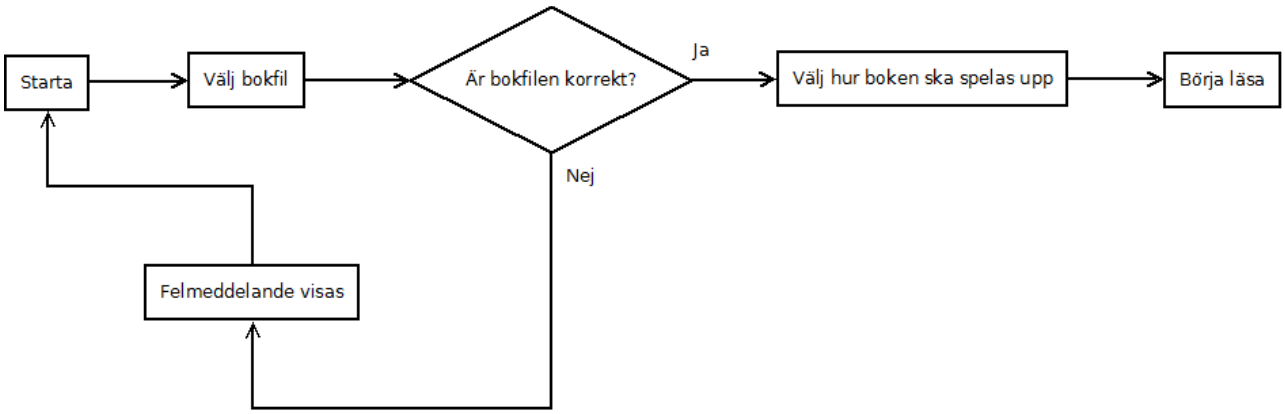
5.2 Hur analysen genomförs

Analysen genomförs genom att ett antal frågor besvaras:

- Vad gör programmet och hur agerar det utifrån användarens perspektiv?
- Vilken miljö är programmet byggt för?
- Går det att identifiera några komponenter och hur samarbetar dessa?
- Används några specifika filstandarder eller egna dataformat?
- Finns det några problem och begränsningar i kodstrukturen?
- Vilka ändringar kan göras för att förbättra kodbasen?

5.3 Vad gör programmet?

Programmet har två lägen. Innan en bok har valts av användaren och laddats visas en enkel skärm, när väl en giltig bok har laddats så går den att läsa. Här är ett enkelt flödesschema som beskriver beteendet innan en bok har öppnats.



Figur 10: Flödesschema över applikationens funktionalitet innan en bok har öppnats

När användaren har valt en bokfil säkerställer bokläsaren att den är korrekt formaterad och att certifikatet är giltigt. Uppfylls inte kraven visas ett felmeddelande och användaren uppmanas att ange en annan bokfil.

Väl inne i läsläge kan användaren göra en rad olika handlingar. Här går det att interagera med innehållet i boken genom att klicka på exempelvis bilder, hyperlänkar eller meningar för att få dem uppspelade. Referera till *Bilaga 1* för ett komplett flödesschema över applikationens beteende i läsläge och *Bilaga 2* för en överblick över applikationens utseende.

5.4 Vilken miljö är programmet byggt för?

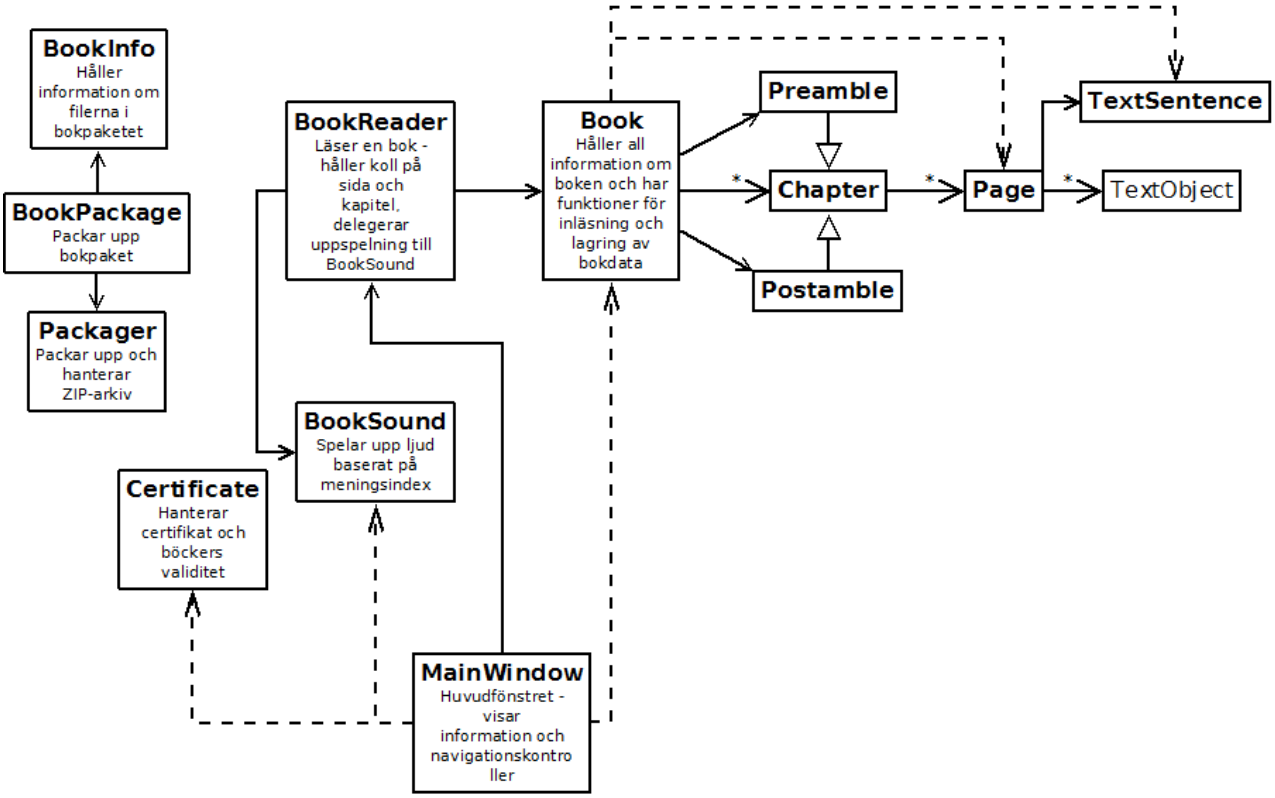
Applikationen är byggd för Windows med .NET version 3.5 Client Profile, vidare används ramverket WPF och programspråket C#.

5.5 Komponenter

Programmet utgörs av totalt 5723 rader kod fördelade på 31 filer. Dessa filer håller till största delen klassdefinitioner som tillsammans bildar tre huvudkomponenter.

5.6 Vilka komponenter finns?

- Användargränssnittet: Utgör fönster och en del logik relaterad till presentation av data och interaktion med användaren.
- Logik: Hanterar enbart logik för bland annat hantering av certifikat, bokläsning, ljuduppspelning samt pakethantering. Denna komponent handhaver ingen presentation eller interaktion.
- Bokdata: Hanterar den största delen av bokdatahantering. Denna komponent har flera arbetsuppgifter och sköter bland annat dataoperationer och konvertering mellan textobjekt och WPF-objekt. Domänmodellen är till största del definierad här.



Figur 11: Översikt över applikationens ursprungliga struktur

Book är uppbyggd av flera kapitel där varje kapitel har flera sidor som i sin tur har flera *TextObjekt*, ett textobjekt kan vara en paragraf, en bild eller annat objekt som kan visas löpande i sidor. Vidare känner Book till varje kapitels sidor och sidors meningar. Flertalet explicita kontroller av för- och slutord görs i Book, där en direkt distinktion mellan de olika kapiteltyperna görs.

5.7 Finns det några filstandarder eller egna filformat?

- Filer med ändelsen *.sbook* utgör ZIP-arkiv där alla filer relaterade till boken lagras och packas.
- Indexeringsinformation och bokinnehåll sparas som logiska XML-filer i ZIP-arkiven.
- Diverse krypteringsmetoder används för att kryptera och verifiera böcker.

6. Förslag på ändringar

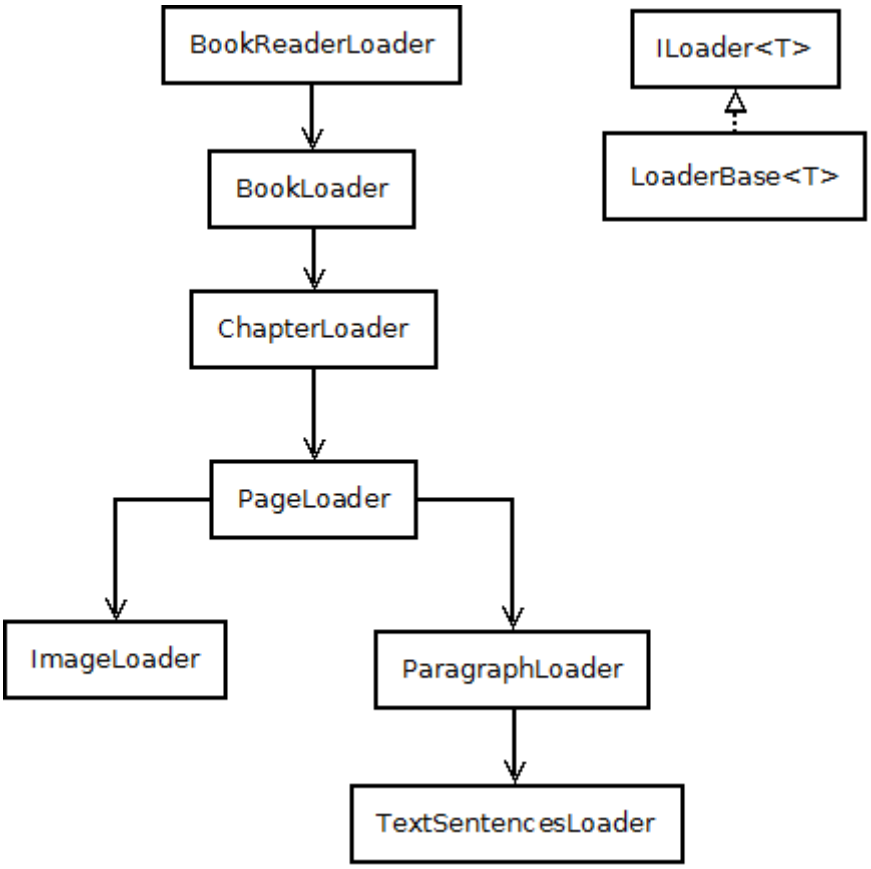
Här följer beskrivning av ändringsförslagen. Ändringarna ämnar att minska kopplingen mellan komponenter och öka kohesionen inom dem, samt att ge en mer intuitiv kodstruktur.

6.1 Prioritering

1. Dataladdare som läser in och konstruerar textrelaterad data.
2. Separation av textobjekt och dess presentation genom konverterare.
3. Borttagning av koncepteten Förord och Slutord samt omstrukturering av meningsindexering.
4. Uppdelning av användargränssnittet och dess logik.
5. Generella ändringar.

6.2 Skapande av dataladdare

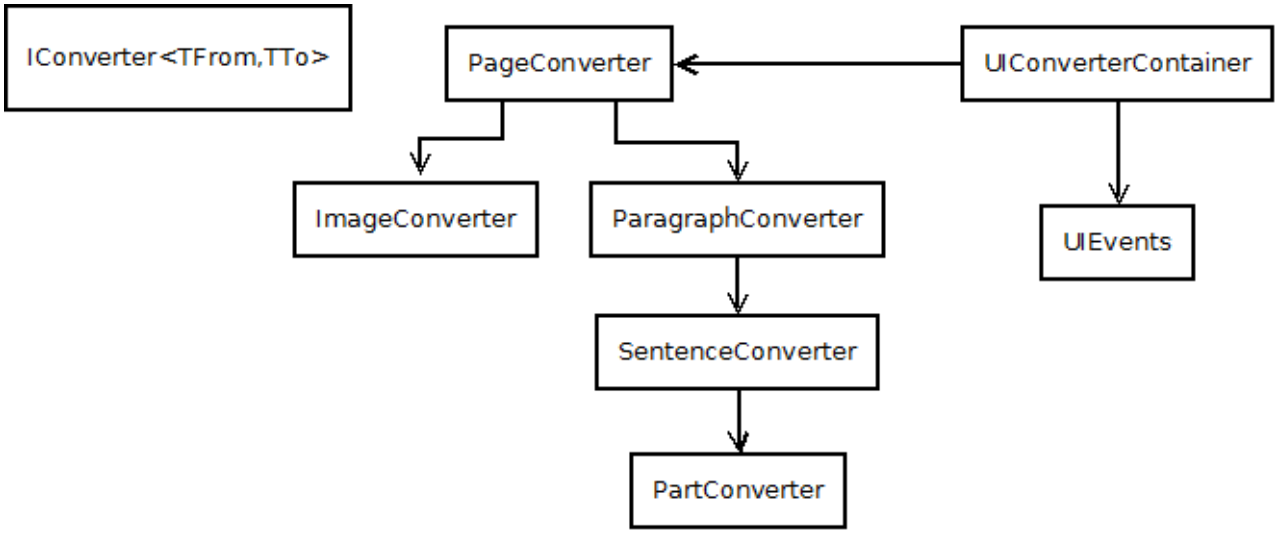
All inläsning av data ska ske via separata dataladdningskomponenter, detta innebär att all kontroll vid skapande av böcker och dess textobjekt lämnas över till olika dataladdare. Målet med detta är att göra bok-komponenter omedvetna om hur de sparas och läses in. Resultatet är att böcker kommer att kunna sparas i olika format och på olika sätt, formatet kan även ändras utan att det påverkar komponenter eller entiteter som inte har någon direkt koppling till det.



Figur 12: Målet med det planerade loader-mönstret

6.3 Skapande av konverterare

I den ursprungliga strukturen hanterar textobjekt sin egen presentation samt omvandling mellan data och presentationsobjekt. Resultatet av detta är hög koppling mellan data och presentation samt att ändringar behöver göras i båda lagren. För att lösa detta införs en konverterare mellan data och presentationsobjekt: När data behöver presenteras på ett visst sätt kontaktas konverteraren som returnerar en representation av datan som går att använda i användargränssnittet.



Figur 13: Målet med det planerade konverterar-mönstret

6.4 Refaktorering av kapitelstruktur

Ljudindexeringen sker på ett sätt som inte återspeglar bokens trädliknande struktur. En bok har en ljudfil associerad till sig och en XML-fil där meningars index finns lagrade. Strukturen är hierarkisk med förord, flera kapitel och sedan slutord. Varje kapitel (inklusive förord och slutord) lagrar sedan indexering för varje mening som hör till kapitlet. Enda skillnaden mellan förord, slutord och kapitel är deras namn och därmed är en tydlig distinktion mellan dem egentligen inte nödvändig. Utöver detta bör kapitel inte lagra varje meningsindex utan istället ha en lista med sidor som i sin tur håller meningars index då detta återspeglar en boks fysiska struktur.

Denna ändring utförs i syfte att ge en tydligare och rakare struktur. Det kommer också att krävas mindre arbete för att redigera böcker i efterhand; om indexering i en sida behöver ändras kommer detta inte att beröra andra delar av boken eller kapitlet.

6.5 Införande av MVVM-mönstret

I den ursprungliga strukturen sköter huvudfönstret inläsning av en bok och skapande av den boknavigationskomponent som sköter navigation i boken. Resultatet av detta är att komponenterna har hög koppling till varandra och otydliga arbetsuppgifter. Klasserna blir dessutom stora då de behöver innehålla mycket kod för att hantera de olika delarna. Istället ska en separat modul sköta bokrelaterad logik, så som navigation mellan sidor och kapitel samt ljuduppspelning. Det är vanligt att MVVM-mönstret införs för att lösa strukturella problem som detta. I praktiken innebär det att så mycket icke-presentationsrelaterad logik som möjligt flyttas till en vymodell så att det tyngre arbetet inte görs i vyn.

6.6 Upprensning och småändringar

Här genomförs en upprensning av koden i allmänhet. Det finns ett antal klasser, filer och pragmadirektiv som inte tycks användas.

7. Refaktorering

Nedan beskrivs genomförandet av de föreslagna ändringarna.

7.1 Skapande av dataladdare

Det första som kommer att utföras är en uppdelning av filerna till flera moduler: All logik placeras i modulen *Data/Loaders* och själva domänmodellen flyttas till modulen *Data/Entities*. Detta för att göra koden mer översiktlig och lättare att arbeta med. Dataladdare för de olika bokkomponenterna implementeras och har funktioner för uppbyggnad av dem. Ett generiskt gränssnitt *(ILoader<T>)* definierar vad för funktioner en dataladdare behöver erbjuda för att klassas som en dataladdare. En implementering av en laddare förväntas därefter konstruera en separat komponent och kan själv specifiera vilka inparametrar och vilken information som den behöver. En bok är uppbyggd av kapitel, kapitel är uppbyggda av sidor och sidor är i sin tur är uppbyggda. Detta gör att ett beroende mellan dataladdarna skapas, där till exempel laddare för kapitel behöver kunna ladda in sidor. För att strukturera upp detta görs en implementering av *beroendeinjicering* där varje dataladdare injiceras med de dataladdare den behöver, viktigt i detta fall är att inga cirkulära beroenden uppstår.

7.2 Skapande av konverterare

Alla konverterare placeras i en separat modul. Ett generiskt gränssnitt *(IConverter<TFrom, TTo>)* som definierar konverteringar av entiteter till komponenter som kan användas i användargränssnittet skapas. Händelser i användargränssnittet kommer att resultera i datamanipulation och i den ursprungliga strukturen registreras lyssnare för dessa händelser direkt i entiteterna. För att utradera denna koppling flyttas alla lyssnare till en separat container. En komponent som behöver en konverterare mellan data och presentation registrerar då lyssnare i containern. Bland konverterarna uppstår samma beroende som hos dataladdarna där till exempel kapitelkonverterare är beroende av konverterare för sidor. Även här används *beroendeinjicering* för att lösa beroendet mellan konverterare.

7.3 Refaktorering av kapitelstruktur

Hela konceptet med Förord och Slutord ska tas bort då det inte finns något principiellt behov av förord och slutord i en bok, den enda egentliga skillnaden mellan de olika kapiteltyperna är titelnamnen. Vidare görs många hårdkodade sökningar efter förord och slutord som inte behövs om det inte görs skillnad mellan dem och kapitel. Vidare tydliggörs strukturen så att böckerna byggs upp genom en trädstruktur där kapitel innehåller sidor och sidor i sin tur innehåller meningar. Förord, slutord och all indexering samt beräkningar relaterade till dessa tas bort för att ersättas av den mer generella lösningen.

7.4 Införande av MVVM-mönstret

Här kommer hanteringen av data separeras från presentationen i huvudfönstret. En vymodell som används av fönstret skapas och logik relaterad till modellen flyttas gradvis till vymodellen. Vidare kopplas så många attribut som möjligt ihop med WPF:s bindningsmotor.

7.5 Upprensning

Här görs en upprensning av kod som är utkommenterad, ändras i klasser som anses göra onödigt jobb eller som inte längre används på grund av omstruktureringen.

8. Implementering av nya funktioner

Här identifieras de olika funktionerna efter en prioritering där den viktigaste funktionen kommer först. Funktionerna prioriteras i syfte att säkerställa att de mest betydande funktionerna hinner implementeras.

8.1 Bokmärken

Användaren ska kunna spara bokmärken i själva bokfilen för att sedan snabbt kunna hitta tillbaka till en tidigare sida. Ett bokmärke kopplas till en unik sida i ett av bokens kapitel, användaren kan sedan navigera direkt till sidan genom en lista med alla bokmärken.

8.2 Sökfunktion

En sökfunktion där användaren ska kunna ange ett ord eller en fras som sedan hittas i boken. En mer avancerad funktion där även felstavade ord hittas är också önskad.

8.3 Egna anteckningar

Det ska gå att föra in korta annoteringar i boken likt PostIt-lappar. Anteckningarna kopplas då till specifika stycken eller bilder i boken. Användaren ska sedan kunna navigera till och läsa anteckningar som gjorts.

8.4 Video

Video-element ska kunna ingå i bokfiler och användaren ska kunna spela, pausa eller stoppa en video som finns inbakad i sidan.

8.5 Animationer

Det ska vara möjligt att definiera rörliga animationer som spelas kontinuerligt på bestämda platser i boken.

8.6 Modernare layout

En modernare layout och design är efterfrågad. En designer är tillgänglig för specifiering av hur det ska se ut. Utöver designerns specifikation behöver layouten omstruktureras för att de nya funktionerna ska få plats i användargränssnittet.

8.7 Bakgrundsfärg

Användaren ska kunna ändra själva applikationens bakgrundsfärg. Det är också tänkbart att applikationen automatiskt anpassar sin bakgrundsfärg efter bokfilens färgschema.

8.8 Genomförande
Här följer en beskrivning av hur de olika funktionerna implementerades.

8.8.1 Bokmärken
Bokmärken implementeras lämpligen som en del av själva bokfilen, på så vis kan de sparas i bokpaketet och lagras tillsammans med resten av bokdatan. Detta gör också att inga filer utöver själva bokfilen behöver sparas på disk för att stödja bokmärken. Det innebär också att en användare kan skapa bokmärken i en bok och sedan dela med sig av bokmärken tillsammans med själva boken.

Ett bokmärke utgörs av en referens till ett kapitel i boken och en sida inom det kapitlet, det bör också vara möjligt att namnge bokmärket och eventuellt ge en beskrivning av det.

Eftersom XML används mycket i projektet redan passar ett egendefinierat XML-baserat format bra även för bokmärken. För att applikationen ska kunna skilja bokmärkesfiler från andra filer i paketet ges de en egen filändelse.

8.8.2 Sökfunktion
Sökfunktionen kan utformas på två olika sätt, lösningen som implementeras först är en enklare sökalgoritm som gör en skiftlägesokänslig matchning mellan sökt ord och ord i boken. Till exempel kommer *foo* att resultera i matchingar både mot *foo* eller *FOO*. Resultatet som returneras till användaren är en lista med sökresultat, där varje sökresultat representerar en lista av de matchningar på varje sida som gjorts.

En mer avancerad sökfunktion som även hittar potentiellt felstavade går att implementera genom att räkna antalet gemensamma bokstäver och bokstavsindex mellan sökord och ord i boken.

8.8.3 Egna anteckningar
Här utnyttjas mycket av koden som användes för bokmärken. För att undvika redundant kod gjordes så att en anteckning principiellt sett är ett bokmärke med indexering till en specifik paragraf i texten. Detta gör så att enbart en kontroll om det är ett bokmärke eller en anteckning behövs innan en operation kan utföras. Dessa skapas också som en del av bokfilen så att inga filer utöver bokfilen behöver sparas på disk, anteckningar har också en egen filändelse.

8.8.4 Video
Här skapas först en separat videospelare med spel-, paus- och stoppknappar. Det kommer också att finnas möjlighet att spola i videon. Först görs en datarepresentation av video som innehåller information om höjd, bredd och källa. Därefter görs en komponent som kan användas i användargränssnittet för att spela upp själva videofilen. Uppbyggnad och inläsning av data görs genom det tidigare uppbyggda systemet med en dataladdare för videodata och en konverterare för att representera datan i användargränssnittet.

8.8.5 Animationer
Här skapas animationsmöjligheter i applikationen. Det diskuterades två olika sätt att implementera detta på där ett av alternativen var att representera geometriska förändringar över tid, där figurers (linjer, kvadrater, ellipser) attribut förändras över tid. Det andra alternativet är att definiera en följd av bilder och tala om hur länge varje bild ska visas, innan växling till nästa bild sker. Det andra alternativet valdes eftersom formatet blir enklare att definiera och inte inger några begränsningar. Nackdelen är att det inte går att definiera generella animationer, animationer baserade på matematiska funktioner eller vektorbaserad grafik. Det ena alternativet behöver inte utesluta det

andra, men det ansågs lämpligast att det snabbare aternativet implementeras först. Tre olika sorters animationer implementerades där uppspelningen kan vara kontinuerlig, aktiveras av användare eller upprepas ett bestämt antal gånger.

8.8.6 Layout

Det skapades flikar istället för funktionsknappar och fönster vilket den ursprungliga layouten hade. Detta ger en översikt över applikationens olika funktionsavdelningar. Programmet skickades sedan till företagets designer för vidare revidering.

8.8.7 Bakgrundsfärg

Här skapades det en flik med inställningar för applikationens färgschema där användaren kan ändra bakgrunds-, text-, markerings- och länkfärg. Ingen separat kontroll över färgerna görs varpå användaren kan ange samma färg för både förgrund och bakgrund (vilket skulle resultera i oläsliga sidor). Motiveringen bakom att inga kontroller utförs är att det är subjektivt vilka färger som skapar oläslighet och var gränsen går. Vidare är gränsen individuell då personer med nedsatt färgseende kan ha svårt för att läsa till exempel grön text på röd bakgrund. Färgschemat sparas som en separat logisk fil i bokpaketet.

9. Resultat

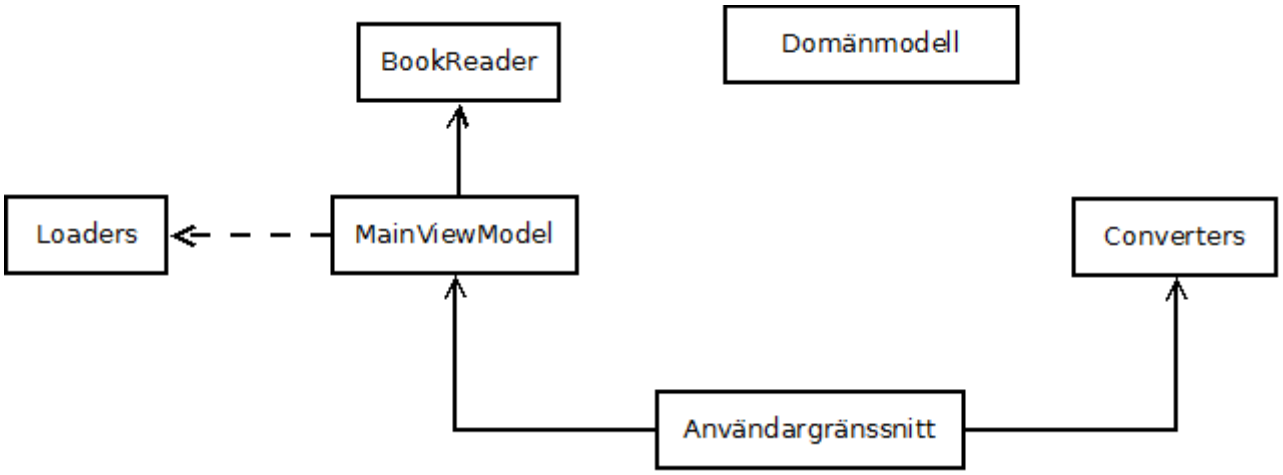
Analysen pekade på att begränsningar i programkodens arkitektur skulle kunna vara ett hinder inför implementeringen av de nya funktionerna. Under analysen uppdagades det också att projektet inte följt något specifikt designmönster, vilket kan göra det svårare för framtida programmerare att sätta sig in i koden och utveckla den vidare.

Det beslutades att det fanns tre högprioriterade förändringar som behövde genomföras för att vidareutvecklingen skulle bli så effektiv som möjligt. Ändringarna innefattade att separera modulerna ifrån varanda så mycket som möjligt i syfte att minska kopplingen och i sin tur kunna vidareutveckla dem i isolation. Utöver detta ändrades hur programmet hanterade kapitel, sidor och meningar då detta ansågs ha en bristande struktur. Vidare utfördes ytterligare två ändringar som ansågs kunna ge en förbättring av applikationens kvalitet, dessa innefattande införande av MVVM-mönstret samt en upprensning av koden i allmänhet i syfte att ge programkoden en renare struktur.

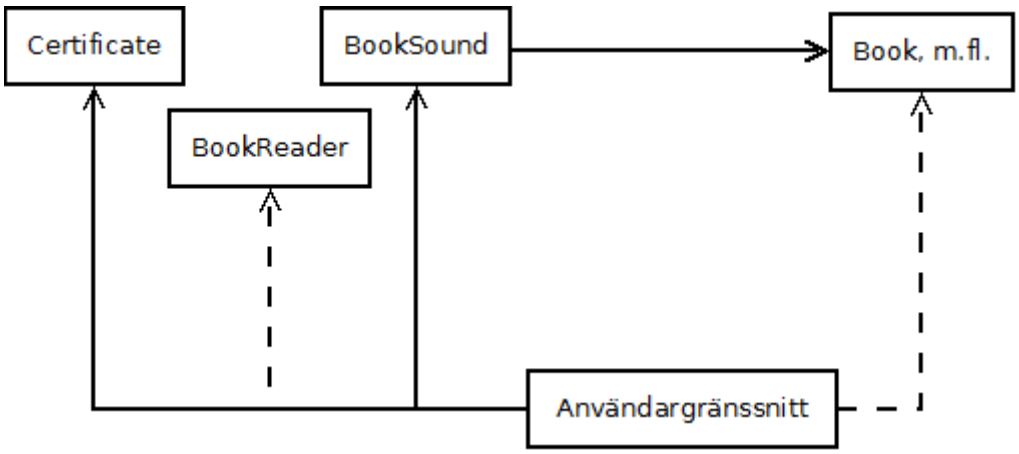
Refaktoreringen resulterade i att programkoden minskades från 5723 rader kod fördelade på 31 filer till 5257 och 68 filer. Antalet filer blev alltså fler medan antalet kodrader färre, en stor del av logiken i klasserna delades i mindre klasser, till stor del för att öka deras kohesion. En hel del kod togs även bort i samband med utraderingen av Förord och Slutord.

Resultatet av refaktoreringen blev att applikationen fick en striktare struktur och mer avgränsade komponenter, i teorin ökar detta applikationens skalbarhet. Det blir lättare att byta ut eller vidareutveckla separata delar av programmet. I koden gjordes en ordentlig upprensning av olika delar som inte användes eller som inte var nödvändiga för applikationen i sig.

Programmet följer nu flera olika designmönster som syftar att ge en tydligare uppdelning av arbetsuppgifter, data, logik och presentation. Ett dataladdningsmönster används för att bygga upp bokrelaterad data som sedan presenteras i användargränssnittet med hjälp av ett konverterarmönster. Nu arbetar komponenter till stor del mot gränssnitt istället för direkt mot andra komponenter. Detta gör att den underliggande implementeringen kan byggas ut eller ersättas helt så länge gränssnittets kontrakt fortfarande åtföljs. Nu används även MVVM-mönstret i användargränssnittet för att separera applikationslogiken från dess presentation. I praktiken innebär detta att en vymodell hanterar bokläsningslogik såsom bläddring och ljuduppspelning, medan användargränssnittet enbart hämtar resultatet av operationerna.



Figur 14: Översikt över applikationens struktur efter ändringarna



Figur 15: Översikt över applikationens struktur före ändringarna

10. Slutsats

10.1 Resumé

Företaget ville att deras bokläsarapplikations funktionalitet skulle uppdateras som ett steg i lanseringsprocessen. Målet med uppdraget var att implementera de funktioner som företaget önskade, efter en grundlig analys av programkoden uppdagades att strukturen skulle göra arbetet mer omständligt än nödvändigt och att en uppdatering skulle underlätta. Applikationen följde ursprungligen inte heller några konkreta designmönster.

Efter analysen genomfördes en refaktorering i fem steg. Programmets funktion regressionstestades hela tiden under förändringarna för att applikationen inte skulle förlora någon av sin ursprungliga funktionalitet. Resultatet av analysen och refaktoreringen blev att programkoden fick en struktur med minskad koppling mellan modulerna och högre kohesion inom dem, till största del genom införsel av olika designmönster. Efter refaktoreringen implementerades de nya funktionerna och de fördes in i den nya strukturen, detta tog kortare tid och fungerade smidigare än förväntat.

10.2 Diskussion

Här kommer vi att diskutera olika saker som uppkommit under arbetets gång samt styrkor och svagheter hos de olika valen vi har gjort.

Att angripa ett problem genom att först analysera och strukturera om delar av koden kan kännas ineffektivt, i vissa fall skulle det även kunna leda till att tid läggs ner på fel saker och att omstruktureringar görs i onödan. Är den grundläggande arkitekturen okänd finns ingen garanti för att den har några brister men en genomgripande analys är aldrig dålig och dessutom ett bra sätt för att ge en förståelse av kodbasen. Det ger en god överblick av hur programmet fungerar och hur det är uppbyggt, därigenom kan man snabbare identifiera olika delar som behöver ändras.

En god lärdom har varit att man aldrig ska fortsätta på fel väg. Så fort man upptäcker att man är på väg i fel riktning, så är det bättre att gå tillbaka och välja en bättre väg. Detta kan dock vara svårt i dagens samhälle eftersom mycket är färskvara och det mesta ska gå snabbt, risken med detta är att problem skjuts fram och istället belastar senare uppdragstagare. Våra egna erfarenheter är att för mycket planering kan bli kontraproduktivt av flera skäl, bland annat är det vanligt att både krav och förutsättningar förändras under projektets gång. Ett projekt tjänar ofta på att vara öppet för förändringar men att samtidigt ha uppsatta delmål har hjälpt oss att identifiera vad som skulle göras härnäst.

Det finns en mängd olika arbetssätt att tillgå vid projektföring (till exempel SCRUM) men vi valde att arbeta utifrån en egen, någorlunda agil approach. En tydlig plan för projektets upplägg har följts och är öppen för förändringar genom iterationsvis planering. God sikt över arbetsuppgifter samt delmål har gett ett bra flöde under projektets gång. En nära kontakt med företaget har hållits och när vi hade gjort klart implementeringen av de nya funktionerna skickade vi applikationen till företaget för testning inför färdigställandet.

Versionshanteringen har varit hjälpsam då Git känns som en mogen lösning för att dela programkod och att ha tillgång till de olika funktionerna det erbjuder har varit en fröjd. Bland annat har vi haft möjligheten att se historiska ändringar och skapa olika förgreningar vid början av varje större förändring.

Dropbox, som vi använde för att dela dokument och filer som inte varit direkt relaterade till själva koden, var också väldigt hjälpsamt. De främsta fördelarna med Dropbox är enkelheten och att det går att återhämta äldre versioner av filer om olyckan är framme.

Designmönster och lösningsmetoder som tas upp i rapporten är valda för att lösa problem som framkommit i samband med detta projekt, det finns många olika sätt att bemöta liknande problem och det är en hel vetenskap vilka designmönster och metoder man ska använda för ett specifikt problem. Vi vet inte om vår lösning på varje problem är den bästa men vi har försökt samla så mycket teori som möjligt innan vi tagit några beslut.

WPF är ett avancerat och komplext ramverk men även väldigt kraftfullt, det har också varit smidigt att arbeta med när inlärningskurvan väl passerats. När de nya funktionerna skulle implementeras visades detta då till exempel videospelaren, som vi trodde skulle ta mest tid att utveckla, visade sig vara en av de delar som tog kortast tid.

Referenslista

[1] Microsoft Developer Network. Introduction to the C# Language and the .NET Framework (Elektronisk) (2012-05-14)
Tillgänglig: http://msdn.microsoft.com/library/z1zx9t92.aspx

[2] World Wide Web Consortium. Extensible Markup Language (XML) (Elektronisk) (2012-05-14)
Tillgänglig: http://www.w3.org/XML/

[3] Microsoft Developer Network. .NET Framework Conceptual Overview (Elektronisk) (2012-05-14)
Tillgänglig: http://msdn.microsoft.com/library/zw4w595w.aspx

[4] Microsoft Developer Network. WPF Architecture (Elektronisk) (2012-05-14)
Tillgänglig: http://msdn.microsoft.com/en-us/library/ms750441.aspx

[5] Chacon, Scott. Pro Git (Elektronisk) (2012-05-14)
Tillgänglig: http://git-scm.com/about

[6] DropBox, DropBox (Elektronisk) (2012-05-14)
Tillgänglig: https://www.dropbox.com/about

[7] Fowler, Martin. Refactoring Improving The Design Of Existing Code (Elektronisk) (2012-05-14)
Tillgänglig: http://martinfowler.com/books.html#refactoring

[8] Moura, Samuel. Quick introduction to MVVM design pattern for WPF (Elektronisk) (2012-05-22)
Tillgänglig: http://blog.smoura.com/quick-introduction-to-mvvm-design-pattern-for-wpf/

[9] Garofolo, Raffaele. Building Enterprise Applications with Windows Presentation Foundation and the Model View ViewModel Pattern, 2011, ISBN: 9780735650923

[10] CTDP. Software Modules (Elektronisk) (2012-05-25)
Tillgänglig: http://www.comptechdoc.org/independent/programming/programming-standards/software-modules.html

[11] BetterExplained. A Visual Guide To Version Control (Elektronisk) (2012-05-25)
Tillgänglig: http://betterexplained.com/articles/a-visual-guide-to-version-control/

[12] Mifflin, Houghton. GUI (Elektronisk) (2012-05-25)
Tillgänglig: http://dictionary.reference.com/browse/GUI

[13] Stallings, William. Computer Security Principles and Practices, 2007**,** 978-0136004240

[14] Rouse, Margaret. Platform (Elektronisk) (2012-05-25)
Tillgänglig: http://searchservervirtualization.techtarget.com/definition/platform

[15] S. Pressman, Roger. Software Engineering: A Practitioner's Approach, 1996**,** 9780070521827

[16] McConell, Steve. Code Complete A Practical Handbook of Software Construction, Second Edition, 2004**,** 9780735619678

[17] Aivosto Oy. Cohesion Metrics (Elektronisk) (2012-05-25)
Tillgänglig: http://www.aivosto.com/project/help/pm-oo-cohesion.html

[18] Tempero, Ewan. Advanced Software Engineering Development Methods,
Lecture 5: Coupling & Cohesion (Elektronisk) (2012-05-25)
Tillgänglig: https://www.se.auckland.ac.nz/courses/SOFTENG701/lectures/lec05-c+c.pdf

[19] Sanderson, Steven, Freeman, Adam. Pro ASP.NET MVC 3 Framework, 2011, 9781430234043

[20] Hunt, Andy, Subramaniam, Venkat. Practices of an Agile Developer, 2006**,** 9780974514086
Tillgänglig: http://pragprog.com/book/pad/practices-of-an-agile-developer

[21] Cashman, Mark. The Converter Pattern (Elektronisk) (2012-05-25)
Tillgänglig: http://www.temporaldoorway.com/programming/cbuilder/techniquesandpatterns/converterpattern.htm

[22] Microsoft Developer Network. The Repository Pattern (Elektronisk) (2012-05-25)
Tillgänglig: http://msdn.microsoft.com/en-us/library/ff649690.aspx

[23] Fowler, Martin. Inversion of Control Containers and the Dependency Injection pattern (Elektronisk) (2012-05-25)
Tillgänglig: http://www.martinfowler.com/articles/injection.html

[24] Microsoft Developer Network. Three-Tiered Distribution (Elektronisk) (2012-05-25)
Tillgänglig: http://msdn.microsoft.com/en-us/library/ff647546.aspx

[25] Pkware. .ZIP File Format Specification (Elektronisk) (2012-05-26)
Tillgänglig: http://www.pkware.com/documents/casestudies/APPNOTE.TXT

[26] Freeman, Adam. Pro ASP.NET MVC 3 Framework, 2011, 9781430234043

[27] Virtual Machinery. Object-Oriented Software Metrics – Class Level Metrics (Elektronisk) (2012-05-26)
Tillgänglig: http://www.virtualmachinery.com/jhawkmetricsclass.htm

[28] Oracle. Interfaces (Elektronisk) (2012-05-28)
Tillgänglig: http://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html

[29] Margueire, Fabrice. The Data Loader Object design pattern: Lightweight Containers, Inversion Of Control, Abstract Factory (Elektronisk) (2012-05-29)
Tillgänglig: http://weblogs.asp.net/fmarguerie/archive/2004/03/09/86536.aspx

[30] Oracle. What Is a Class? (Elektronisk) (2012-05-31)
Tillgänglig: http://docs.oracle.com/javase/tutorial/java/concepts/class.html

[31] Clifton, Mark. What Is A Framework? (Elektronisk) (2012-05-31)
Tillgänglig: http://www.codeproject.com/Articles/5381/What-Is-A-Framework

[32] Saqib, Muhammad. What is a function? (Elektronisk) (2012-05-31)
Tillgänglig: http://www.mycplus.com/tutorials/c-programming-tutorials/functions/

[33] Microsoft Developer Network. entity type (Entity Data Model) (Elektronisk) (2012-05-31)
Tillgänglig: http://msdn.microsoft.com/en-us/library/ee382837.aspx

[34] Biggert, Johnny. Interfaceskolan del 3: vägen mot oförstörbar kod, bygg in testbarhet redan från början (Elektronisk) (2012-05-31)
Tillgänglig: http://www.johnnybigert.se/interface_testbarhet.html

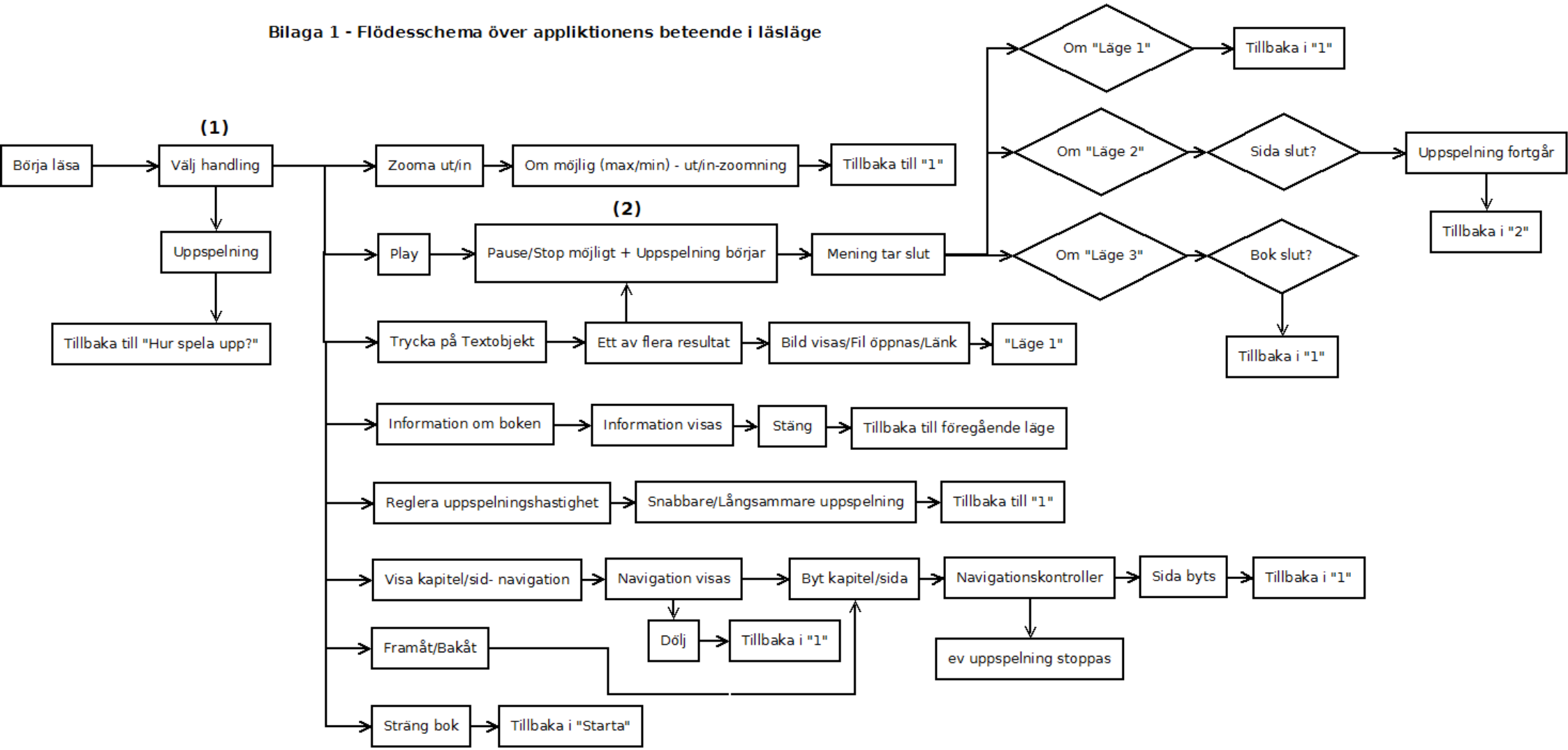
[35] A. Moore, Scott. Entity-relationship modelling (Elektronisk) (2012-05-31)
Tillgänglig: http://www.inf.unibz.it/~franconi/teaching/2000/ct481/er-modelling/

[36] Ambler, Scott. Mapping Objects to Relational Databases: O/R Mapping In Detail (Elektronisk) (2012-06-01)
Tillgänglig: http://www.agiledata.org/essays/mappingObjects.html

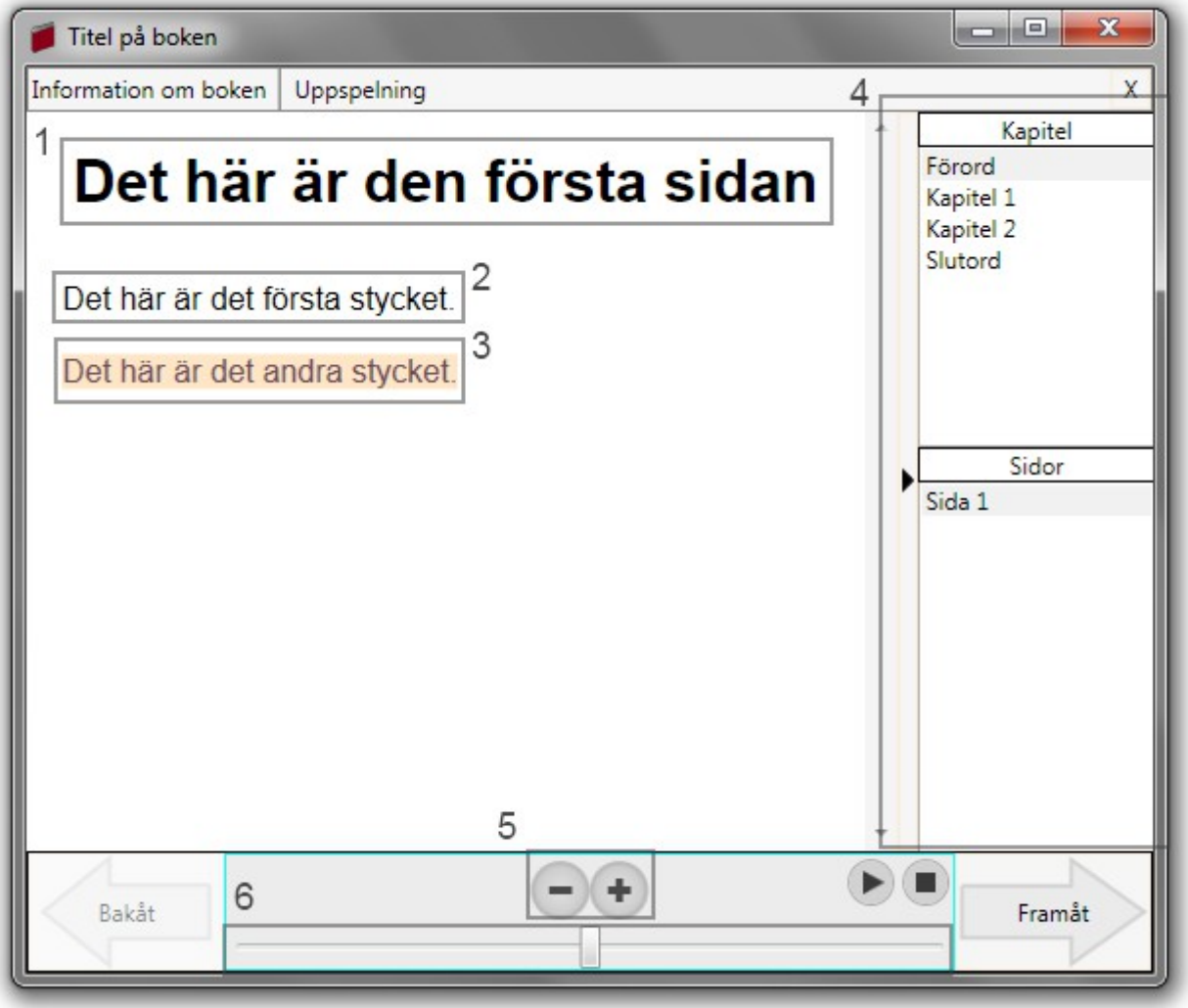
Bilagor

Bilaga 1 Komplett flödesschema över applikationens beteende i läsläge
Bilaga 2 Översikt över applikationens utseende

Bilaga 1 - Flödesschema över appliktionens beteende i läsläge



Bilaga 2



1. Rubrik i boken
2. Icke-markerat stycke
3. Markerat stycke (spelas upp eller har musen över sig)
4. Navigations-menu, här kan användaren snabbt navigera till specifika kapitel och sidor
5. Zoom-knappar
6. Reglering av uppspelningshastighet