Avdelning för datavetenskap

Sebastian Skoglund och Per-Erik Svensson

Telefonkataloghantering för mobila enheter

Telephone directory management for mobile units

Examensarbete (20p)
Civilingenjör Informationsteknologi

Datum:	07-01-18
Handledare:	Katarina Asplund
Examinator:	Donald Ross
Löpnummer:	D2007:02

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

Telefonkataloghantering för mobila enheter

Sebastian Skoglund
Per-Erik Svensson

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

Sebastian Skoglund

Per-Erik Svensson

Godkänd, Date of defense 07-01-18

Handledare: Katarina Asplund

Examinator: Donald Ross

Sammanfattning

The PhonePages är ett företag som utvecklar mjukvara för mobila enheter, främst mobiltelefoner. Denna uppsats behandlar utvecklingen av, och möjligheterna för, en mobil telefonkatalogtjänst, resurssnål nog för att passa i en mobil enhet. Arbetet utfördes åt The PhonePages med avsikten att en färdig produkt ska kunna säljas till tredje part.

Genom att studera olika tänkbara lösningar, deras fördelar och nackdelar, så byggs en abstrakt bild av produkten upp. Här behandlas vissa kompatibilitetsproblem med de olika plattformar som dagens mobila system innebär. Uppsatsen tar också upp hur data kan sparas, organiseras och presenteras i dessa system. Huvudmålet är att skapa en telefonkatalogtjänst vilken inte behöver göra externa uppslag eller förfrågningar. Tjänsten ska innehålla både företag och privatpersoner, med namn och telefonnummer till respektive. Dessutom ska fullständig adressinformation finnas. Vad gäller företag ska applikationen kunna hantera olika prioritet och logotypbilder.

Den applikation som blev resultatet av arbetet på The PhonePages fungerar självständigt, utan uppslag mot Internet och är helt implementerad i J2ME, helt i enlighet med kravspecifikationen. Analysen av de olika tänkbara lösningarna ledde med andra ord fram till en fungerande applikation.

Abstract

The PhonePages of Sweden is a company that develops software for mobile units, especially cell phones. This thesis treats the development of, and contingencies for, a mobile phone directory, using the limited resources found in a mobile unit. The project was implemented and executed at The PhonePages with the intention of creating a product to sell to a third party.

By studying different solutions, their benefits and drawbacks, an abstract picture of the product was constructed. Problems covered include compatibility problems caused by todays platform diversity as well as problems with saving, organizing and presenting data. The main goal was to create a phone directory which does not make external information retrievals. The service should contain both company and personal information, with name and phonenumber. Complete address information should also be available. The application should also manage different priorities and logotypes for the company information.

The application, that emerged as a result of our work at The PhonePages, works independently, without making connections to the Internet and is completely implemented in J2ME, all according to the requirement specification. In other words, the analysis of the different solutions led to a working application.

Innehåll

1 Introduktion **1**

2 Bakgrund **3**

2.1 Existerande system 3

2.1.1 Helhetslösningar 3

2.1.2 Pointbase 4

2.2 Tunna klienter kontra tunga klienter 4

2.2.1 Fördelar/nackdelar med tunna klienter (tunga servrar) 5

2.2.2 Fördelar/nackdelar med tunga klienter 7

2.2.3 Slutsats 8

3 Analys av möjliga lösningar **10**

3.1 Teknisk kravspecifikation 11

3.2 Spara data 13

3.2.1 RecordStore 14

3.2.2 File Connection Optional Package 16

3.2.3 Data som del av programmet 17

3.2.4 Data i resursfiler (.jar) 18

3.2.5 PointBase Databashanterare 19

3.2.6 Resultat 19

3.3 Organisera data 20

3.3.1 Data i en fil 21

3.3.2 Token-tabell 22

3.3.3 Token-tabell med buckets 24

3.3.4 Konstgjord Random Access 27

3.3.5 Resultat 28

3.4 Indexera data 29

3.4.1 Trie hashing . . . 29
3.4.2 Simple prefix B-tree . . . 30
3.4.3 Binära sökträd . . . 31
3.4.4 Resultat . . . 33
3.5 Presentera data . . . 34
3.5.1 J2ME enligt javax.microedition.lcdui . . . 34
3.5.2 J2ME enligt de.enough.polish.ui . . . 35
3.5.3 Resultat . . . 36
3.6 Komprimera data . . . 37
3.6.1 Entropi . . . 37
3.6.2 Strömkoder gentemot blockkoder . . . 39
3.6.3 Huffman . . . 40
3.6.4 Byte Pair Encoding BPE . . . 41
3.6.5 Resultat . . . 41
3.7 Stöd för kartor . . . 42
3.7.1 Rastergrafik . . . 42
3.7.2 Vektorgrafik . . . 44
3.7.3 Slutsats . . . 45

4 Design **48**
4.1 Användargränssnitt . . . 48
4.2 Använda designmönster . . . 51
4.2.1 Model-View-Controller . . . 52
4.2.2 Consumer/Producer . . . 54
4.2.3 Observer . . . 55
4.2.4 Singleton . . . 57
4.3 View . . . 58
4.4 Controller . . . 61

4.5 Model 64
4.5.1 Entitetsrelationer 65
4.5.2 Databas 66

5 Implementation **70**
5.1 Samtidig sökning och inmatning 70
5.2 Inmatning utan J2ME 73
5.3 Fonetisk strängmatchning 78
5.4 Spara och ladda en trädstruktur 83
5.4.1 Traversera träd 84
5.4.2 Uttryckskträd 86
5.4.3 Spara trädstruktur 88
5.4.4 Ladda trädstruktur 89
5.5 Mängdlära och Relationsalgebra 92
5.5.1 Snitt och union på sorterad lista 95

6 Slutsats **99**
6.1 Resultat och utvärdering 99
6.2 Problem 101
6.3 Framtida arbete 102

Referenser **104**

A Projektbeskrivning **107**

B Kravspecifikation **109**

Figurer

2.1 Olika typer av ändsystem 5
3.1 Data i fil 21
3.2 Token-tabell 23
3.3 Token-tabell med buckets 25
3.4 Exempel på ett Trie Hashing 31
3.5 Exempel på ett simple prefix B-tree 32
3.6 Exempel på ett binärt sökträd 33
3.7 Ett träd där lövnoderna är händelser som beror på sina föräldrar. 38
3.8 Exempel på karta 43
3.9 Centrala Göteborg utan skalning 45
3.10 Centrala Göteborg, skalat 46
3.11 Exempel på karta i vektorformat. 47
4.1 De fyra logiska vyerna över applikationen. 49
4.2 En Sony-Ericsson W800i med sina två “softkeys” inringade. 50
4.3 Hur Model-View-Controller kopplas till varandra. 53
4.4 Model-View-Controller och dess implementation i applikationen. 54
4.5 Designmönstret Observer. 55
4.6 Designmönstret Singleton. 57
4.7 Programflödet styrt av Main. 59
4.8 Applikationen när en SearchForm är aktiv. 60
4.9 UML-klassdiagram över View. 61
4.10 UML-klassdiagram över Controller. 63
4.11 UML-klassdiagram över Event. 64
4.12 ER-diagram över Vita Sidorna. 65
4.13 ER-diagram över Gula Sidorna. 66
4.14 Konceptuell vy över databas. 67

4.15 UML-klassdiagram för ADT:n Hash Trie. 68
4.16 Design av Hash Trie. 69
5.1 Källkoden för setChar(). 71
5.2 Källkoden för getChar(). 72
5.3 Källkoden för run(). 73
5.4 En tråd vars uppgift är att ta tid. 75
5.5 En tråd vars uppgift är att ta tid. 76
5.6 Interfacet för klassen *Soundex* - med privata metoder synliga. 80
5.7 Den publika metoden soundex(char). 81
5.8 Den privata metoden soundex(String). 82
5.9 Pseudo-kod för metoden *preorder()* för noder i ett träd. 85
5.10 Pseudo-kod för metoden *inorder()* för noder i ett träd. 85
5.11 Pseudo-kod för metoden *postorder()* för noder i ett träd. 86
5.12 Trädet som representerar uttrycket $(a + b)/((c - d) \cdot e)$. 87
5.13 Kod för utskrift av inre noder till fil i en trädstruktur. 88
5.14 Kod för utskrift av lövnoder till fil i en trädstruktur. 88
5.15 Exempel på en trädsktruktur som ska sparas. 89
5.16 Kod för att ladda ett träd från fil. 90
5.17 Process där träd byggs från datat i (5.4). 91
5.18 Venn-diagram. 93
5.19 Pseudo-kod för snitt på två mängder 96
5.20 Två sorterade listor. 96
5.21 Snitt. 98

Tabeller

3.1 Huvudargument för val av lösning 20
3.2 Ordning för hur efternamnen sattes in 30
5.1 Olika mängder och deras relation till venn-diagrammet i figur 5.18 93

1 Introduktion

Syftet med den här uppsatsen och det projekt som utfördes åt The PhonePages of Sweden AB är att utveckla en mobil applikation som kan användas för att söka i en telefonkatalog på vita och gula sidorna. Vita sidorna innehåller information om namn, telefonnummer, adress m.m. för privatpersoner. Gula sidorna innehåller motsvarande information för företag. Bakgrunden till projektet är dagens lösningars ofta långa väntetider och låga grad av interaktivitet. Applikationen ska utvecklas i Java 2 Platform, Micro Edition (J2ME). J2ME är en variant av Java 2 Platform, Standard Edition (J2SE) för mindre, ofta mobila, enheter, t.ex. mobiltelefoner eller PDA:er (Personal Digital Assistance). Ett av de mest centrala kraven som projektbeställarna, The PhonePages of Sweden AB, har ställt är att produkten inte får göra uppslag mot en server på Internet. Informationen för vita och gula sidorna ska därför ligga lokalt på den mobila enheten, något som kommer genomsyra resten av den här rapporten då det innebär många utmaningar eftersom de mobila enheterna har begränsade resurser i form av minne och beräkningskapacitet. Dessa begränsningar kommer påverka på vilket sätt vi kan spara och organisera data; det ska å ena sidan ta så liten plats som möjligt men samtidigt gå snabbt att söka i det. Dessa två koncept motsäger ofta varandra vilket gör det nödvändigt att hitta nya lösningar. Fullständig information angående kraven för applikationen finns i kravspecifikationen i bilaga B. Uppdragsbeskrivningen finns i bilaga A.

Den produkt som slutligen överlämnades fungerar självständigt, utan uppslag mot Internet, och är helt implementerad i J2ME. Vi har själva skrivit kravspecifikation, analyserat, designat och slutligen implementerat applikationen. Detta har inneburit bland annat att egna indexeringsalgoritmer, fonetisk strängmatchning och konstgjord slumpmässig filaccess implementerats. Vi har också undersökt vilka komprimeringsalgoritmer som lämpar sig bäst för den här typen av applikation, hur strängar indexeras och hur mängdoperationer implementeras mest effektivt.

Nästa kapitel, kapitel 2, tar mer ingående upp bakgrunden till projektet och redogör

framförallt för existerande system och vilka nackdelar respektive fördelar dessa system har. Därpå följer, i kapitel 3, en inledande analys baserad på kravspecifikationen. Här beskrivs kraven mer kortfattat och därefter tas olika lösningar upp, ställs mot varandra och slutligen väljs den mest lämpade lösningen. Här kommer också många av de problem som stötts på under projektets gång redovisas, i vissa fall implicit. I kapitel 4 beskrivs applikationens design, såväl användargränssnitt som mjukvarudesign. En kort introduktion till vad designmönster är och vilka som använts finns också här. Därefter följer implementation logiskt i utvecklingskedjan varför kapitel 5 tar upp just detta. Hela applikationens implementationslösningar kommer inte att beskrivas, men däremot för applikationen speciellt intressanta delar. Avslutningsvis följer, i kapitel 6, slutsatsen vilken beskriver våra resultat, problem och möjliga framtida arbeten.

2 Bakgrund

Detta kapitel tar, mycket kortfattat, upp bakgrunden till projektet. Först redogörs för redan existerande lösningar och deras begränsningar. Därefter kommer en introduktion till tunna kontra tjocka klienter, ett avsnitt motiverat dels av den något kontroversiella kravspecifikationen[1] och dels av att läsaren snabbt får en inblick i de problem som uppstår då databaser ska distribueras till värdsystemen/klienten.

2.1 Existerande system

Vi tar här upp de system som tillhandahåller antingen helhetslösningar eller delar av de lösningar som vi tänker utforma. De system som delvis uppfyller våra behov är i synnerhet de som hanterar lagring och sökning av data i mobila applikationer.

2.1.1 Helhetslösningar

Det finns i dagsläget ett motsvarande system i Sverige från Eniro där man via en Java-klient kan göra sökningar på privatpersoner och företag. Det som skiljer deras applikation från vår är att sökningar sker mot en server på Internet via General Packet Radio Services (GPRS), 3G eller hur användaren än är uppkopplad med sin mobiltelefon. Modellen som de använder är alltså den klassiska *client/server*. Problemet är att detta ställer vissa krav på bandbredden mellan klienten och servern, krav som sällan uppfylls av GSM. Vidare kan en applikation med data lagrat lokalt ofta göras mer interaktiv. Här läggs steglös zoomning in som ett argument av uppdragsgivaren. Vår allmänna uppfattning av Eniros tjänst är att den är något långsam. Detta kan bero på att vi är begränsade av Groupe Spécial Mobile/General Packet Radio Services (GSM/GPRS) vilket inom en snar framtid ersätts av 3G. När så skett är det mycket möjligt att hastigheten inte upplevs som ett hinder. Det finns dock andra, mer övergripande, problem med tunna klienter. Det bör dock nämnas att

[1] Kontroversiell i den mening att denna typ av applikation traditionellt använder en ”klient-server-modell med en mycket tunn klient.

det finns tydliga begränsningar även med fristående klienter. För en något mer ingående analys av detta, se avsnitt 2.2.

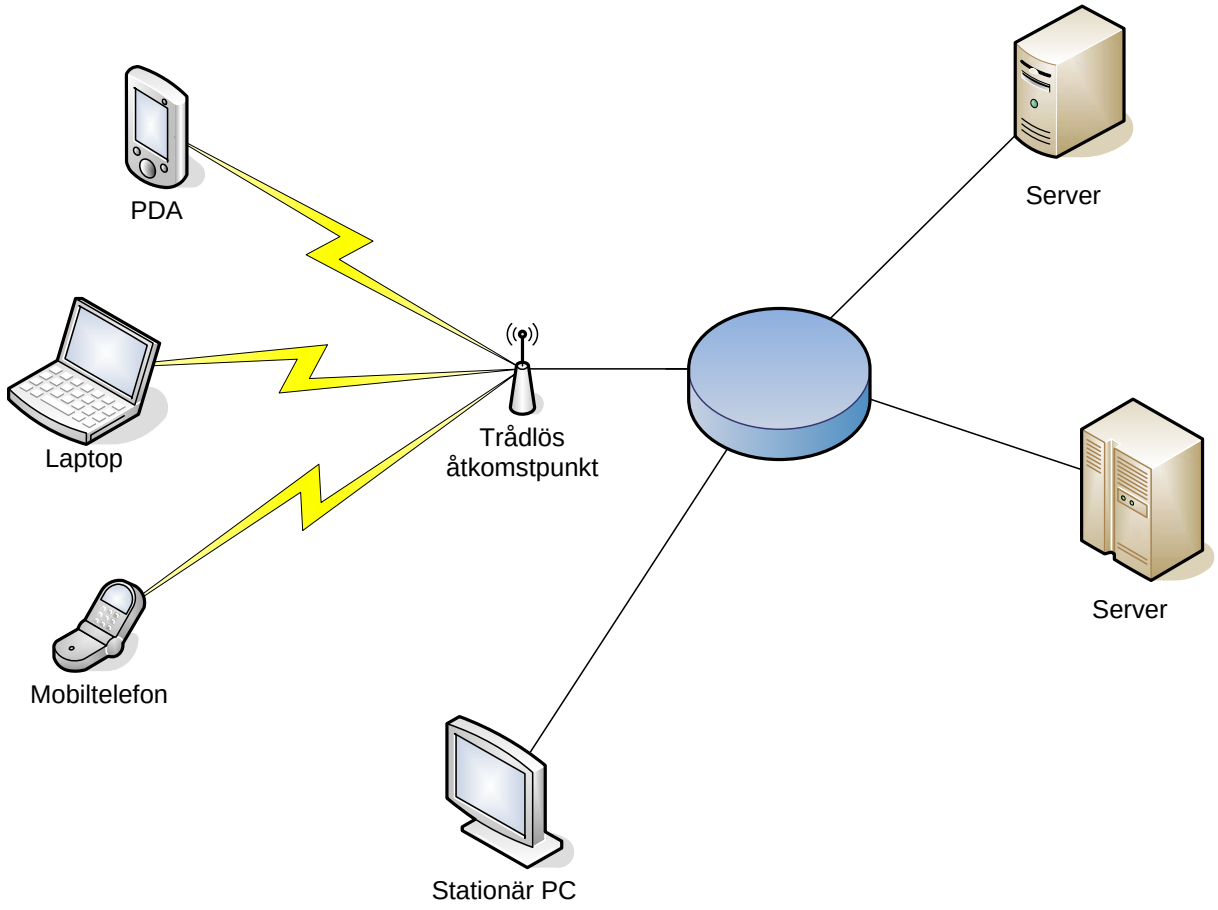
2.1.2 Pointbase

Pointbase är ett företag som utvecklar relationsdatabaser i och för olika Java-plattformar, på så vis blir databasen plattformsoberoende. Det finns bl.a. en produkt som heter Pointbase Micro som är speciellt framtagen för att köras på mindre enheter under J2ME. Pointbase Micro använder SQL-syntax (Structured Query Language) och implementerar en delmängd av det API som finns för JDBC (Java Database Connectivity) [19]. Detta sammantaget gör det idealt att använda som databas för vår applikation förutsatt att det går tillräckligt fort att söka och att data inte tar för mycket utrymme att lagra. Mer om detta finns i avsnitt 3.2.

2.2 Tunna klienter kontra tunga klienter

I många sammanhang inom nätverk så talar man om tunna (eng. thin) eller tunga (eng. thick, fat) klienter (alt. tunga klienter eller tunga servrar) för att beskriva ansvaret för de olika komponenterna i ett system i sin helhet. En server eller klient är ofta vad som beskrivs som ett *end system*, vilket implicerar att det konceptuellt sett befinner sig i utkanten av ett nätverk. De båda benämns även som *värddatorer* (eng. hosts), eftersom de är värdar för programvara som körs på dem. Klienter är ofta stationära hemdatorer, laptops, PDA:er, mobiltelefoner etc. Servrar är kraftfullare maskiner som kör server-mjukvara, t.ex. web- eller mailtjänster [14]. Figur 2.1 illustrerar de olika typer av ändsystem som återfinns på ett nätverk.

Med “tjocklek” på en klient eller server så avser man hur mycket data eller logik som finns på den. Det som är intressant är alltså hur man distribuerar arbetsbördan mellan de olika komponenterna för att uppnå bästa prestanda, säkerhet och effektivitet för en given applikation eller service. En tjock klient/server innebär således att man lägger mycket av



Figur 2.1: Olika typer av ändsystem

logiken och/eller informationen på just den aktuella enheten. Ett klassiskt exempel på en tjock server och en tunn klient är *html* över *http*, där i princip all logik och data finns lagrad i servern och klientens enda uppgift är att presentera informationen för användaren. Nedan följer en kortare beskrivning av fördelarna samt nackdelarna med tjocka servrar och klienter för en applikation som vi utvecklar, tillsammans med en slutsats.

2.2.1 Fördelar/nackdelar med tunna klienter (tunga servrar)

Att lägga mycket av logiken och data på en server har följande fördelar:

- **Lätt att administrera.**

Eftersom data ligger centraliserat så är det lätt att administrera och uppdatera vid behov. Eventuella ändringar behöver då bara genomföras på ett ställe.

- **Anpassat för klienter med begränsade resurser**
  Kraven på hårdvaran hos klienterna behöver inte vara lika höga eftersom dess uppgift är att skicka och ta emot information och presentera denna. Det innebär att klienten inte har samma behov av minne, beräkningskapacitet etc. Dessutom innebär det att målgruppen för applikationen blir större eftersom att antalet mobiltelefoner som uppfyller minimikraven blir fler.

- **Låg energiförbrukning**
  Energiförbrukningen hos klienterna blir mindre då de inte behöver utföra tunga beräkningar.

- **Mer information**
  En kraftfull server har, i princip, obegränsat med minne jämfört med en mobiltelefon. Det innebär att mer information kan lagras och därmed kan en attraktivare tjänst erbjudas. Exempelvis så kan fler bilder, logotyper, kartor och annan information lagras.

- **Mindre programvara**
  En tjock server implicerar en mindre programvara vilket gör det enklare för användaren att ladda ner den till sin mobila enhet.

Nackdelar med tunga servrar:

- **Single point of failure**
  Sårbarheten för systemet ligger i en punkt: Servern. Om servern skulle gå ner så blir hela systemet oanvändbart under den tiden. Detta gör servern utsatt och kräver åtgärder för att förhindra, t.ex. backup-servrar.

- **Nätverksanslutning**

  Ett möjligt scenario är att en mobiltelefon inte kommer åt servern p.g.a. att den inte har kontakt med Internet (via GSM- eller 3G-nätet). Användaren har då ingen möjlighet att utnyttja systemet.[2]

  Internet via det mobila GSM-nätet medför ofta en lång fördröjning (jämfört med en traditionell bredbandsuppkoppling) vilket ger sämre responstid i sökningar osv.

2.2.2 Fördelar/nackdelar med tunga klienter

Fördelarna med att använda en lösning med tunga klienter är i synnerhet:

- **Oberoende av server**

  Klienten blir mer oberoende av servern och behöver därför inte vara i kontakt med en basstation (och därmed Internet) för att fungera.

  Det innebär samtidigt att man slipper problemet med *single point of failure* som det innebär att använda en tung server.

- **Bättre responstid**

  Eftersom lite eller ingen kommunikation behöver ske mot en server så kommer responstiden för sökningar bli mindre.

- **Interaktiv tjänst**

  En tung klient har fördelen att den kan erbjuda mer interaktiva multimedia-tjänster. T.ex. så skulle en tung klient kunna ha inbyggda kartor där man steglöst kan zooma.

Nackdelar med tunga klienter:

- **Krav på hårdvara**

  Eftersom data och logik ska ligga lokalt på en mobil enhet så krävs det mer i pre-

[2]Detta är för vår applikation inget stort problem eftersom man ändå inte kan ringa det nummer man söker om man inte har kontakt med en basstation

standa av enheten. Den ska rymma en databas för c:a 90.000 poster[3] och kunna göra sökningar inom rimliga tidsramar[4].

- **Stor programvara**

  Eftersom mycket eller all logik och data ska ligga i klienten så blir därför programvaran större, vilket förutom mer minne hos enheten, medför att det blir mer problematiskt att ladda ner applikationen till mobiltelefonen.

- **Svår att administrera**

  En telefonkatalog är dynamisk, i den bemärkelsen att människor flyttar och byter telefonnummer, adress, ort osv. Ändringar i katalogen, som användaren vill ta del av, kommer medföra att användaren måste ladda ner en ny version av mjukvaran till sin mobila enhet. Det samma gäller naturligtvis för andra sorters uppdateringar i programvaran (t.ex. bugfixar).

2.2.3 Slutsats

Att prata om tunga klienter/servrar är egentligen bara aktuellt i kontexten av *client/server*-modellen och då vår applikation inte använder sig av någon server överhuvudtaget så utgår vi från att det är en så tung klient som man kan åstadkomma.
Samtliga nackdelar med en tung klient (avsnitt 2.2.2) gör den lösning olämplig för ändamålet. Uppdragsgivaren vill att applikationen ska fungera på de tio vanligaste mobiltelefonerna (se bilaga A), vilket vi anser svårt då det kommer kräva för mycket minne och beräkningskapacitet än vad de flesta mobiltelefoner idag erbjuder. Svårigheterna med att administrera ett system som inte är centraliserat gör att ändringar i databasen, t.ex. när en person byter adress, tvingar fram en helt ny "version" av applikationen och att informationen hos klienter efter en tid blir föråldrad. Nackdelarna med att använda en tung server (avsnitt 2.2.1)

[3] Uppsalakatalogen, som enligt Lokaldelen själva ska vara den största, består av lite mer än 90.000 poster

[4] "Rimliga" tidsramar är att det ska gå fortare än att göra en sökning över Internet. Exakt information finns i bilaga B

kan motverkas genom att använda back-up servrar som träder in ifall den ordinarie servern av någon anledning skulle gå ner. När det gäller nackdelen med dålig eller ingen kontakt med basstationen för den mobila enheten så tror vi inte att man har så stor användning av en sådan tjänst i alla fall eftersom att man ändå inte kan ringa eller kontakta den person man söker.

Vi anser att den bästa lösningen för denna sorts tjänst är att göra en tunn klient uppkopplad mot en tung server. En mobiltelefon är inte anpassad för att användas som tung klient och det rekommenderas att lägga så lite logik och data på den som möjligt[12]. Beslutet om en tung klient ligger alltså utanför vårt inflytande vilket också är en del av motivationen bakom detta avsnitt.

3 Analys av möjliga lösningar

Huvudmålet med analysen är att fastställa huruvida det är praktiskt möjligt att ha en mobil, lokal katalogtjänst. Då mobila enheter har mycket begränsade resurser, både vad gäller processorkapacitet och minne, så ställs stora krav på applikationen. För att kunna fastställa vilka begränsningar som finns i dagens mobila enheter, då främst mobiltelefoner, gjordes först en lista över de tio mest sålda mobiltelefonerna under augusti månad 2006, enligt Telia-Sonera [26]. Utifrån denna lista sattes sedan en kravspecifikation upp, hur mycket resurser kan applikationen kräva?

Avsnitt 3.1, *Teknisk kravspecifikation*, beskriver kortfattat applikationens krav och ska endast ses som en grov uppskattning då det vid tidpunkten för skapandet rådde stor osäkerhet kring vad som var tekniskt möjligt ens med de mest kraftfulla mobiltelefonerna.[5]

Avsnitt 3.2, *Spara data*, beskriver analysen av de olika möjligheterna att spara information på en mobil enhet.

Avsnitt 3.3, *Organisera data*, handlar om hur data ska struktureras för att kunna uppnå önskade resultat gällande söktider och minnesanvändning i enlighet med kravspecifikationen.

Avsnitt 3.4, *Indexera data*, tar upp ett antal möjligheter för att indexera data, vilket är nödvändigt för att ytterligare snabba upp sökningen.

Avsnitt 3.5, *Presentera data*, beskriver möjligheterna för att grafiskt presentera data i J2ME på de mobila enheterna.

Avsnitt 3.6, *Komprimera data*, beskriver hur man kan komprimera data som samtidigt ska vara sökbar utan att påverka söktiderna i någon större skala.

Avsnitt 3.7, *Stöd för kartor*, tar upp metoder för att hantera kartor i mobiltelefonen.

[5]Fullständig kravspecifikation finns i bilaga B

3.1 Teknisk kravspecifikation

Här listas i korthet de viktigaste tekniska kraven.

- Applikationen ska kunna hantera både vita och gula sidorna.
- En post i de vita sidorna ska innehålla följande:
  - Förnamn
  - Efternamn
  - Gatuadress
  - Gatunummer
  - Postnummer
  - Stad
  - Telefonnummer
  - Geografisk position enligt WGS84 datum [17]
- En post i de gula sidorna ska innehålla följande:
  - Namn
  - Gatuadress
  - Gatunummer
  - Postnummer
  - Stad
  - Två telefonnummer (ett huvudnummer och ett valfritt nummer)
  - Logotyp för företaget (om så önskas)
  - Geografisk position enligt WGS84 datum [17]

  - Prioritet

- För att kunna söka i branscher krävs en branschtabell. Branschtabellen ska innehålla följande:
  - Branschnamn
  - Lista över alla företag som hör till den branschen
- Följande fält i databasen ska vara sökbara
  - Förnamn (Vita sidorna)
  - Efternamn (Gula sidorna)
  - Telefonnummer (Vita/Gula sidorna)
  - Bransch (Gula sidorna)
  - Företagsnamn (Vita sidorna)
- En hög prioritering för ett företag innebär att företagets namn placeras högre upp vid sökning.
- Det ska endast finnas ett sökfält.
- När användaren har påbörjat en sökning presenteras först antalet träffar som matchar söksträngen i de gula respektive vita sidorna.
- Användaren ska kunna ringa upp ett framsökt nummer.
- Om och endast om användarens telefon stödjer Personal Information Management(PIM, i enlighet med JSR 75) ska användaren kunna spara framsökta nummer i sin telefonbok.
- Ingen säkerhet krävs. Dock måste personuppgiftslagen följas.

- Applikationen måste kunna räkna antalet sökträffar på mindre än 500 ms.
- Applikationen måste kunna visa de 10 första sökträffarna på mindre än 500 ms.
- Applikationen måste, då användaren så efterfrågar, visa nästa/föregående 10 sökträffar på mindre än 500 ms.
- Databasen ska kunna hantera 100 000 poster.
- Applikationen ska inte ta upp mer än 5 MB av varaktigt minne.
- Applikationen ska inte uppta mer än 250 Kb av internminne.
- Applikationen ska inte någon gång under en sökning ta hjälp av extern källa, så som en Web-server.
- Systemet ska klara av en 'black box test'-svit i JUnit, designad för att täcka alla funktionella krav uttryckta i kravspecifikationen. (Se bilaga B.)

3.2 Spara data

Då applikationen ska kunna hantera en stor mängd data lokalt måste det finnas möjlighet att spara data varaktigt (eng. persistent). I J2ME ges flera möjligheter till detta och i det här avsnittet ställs dessa mot varandra för att slutligen avgöra vilken lösning som passar den tänkta applikationen bäst. För att det ska vara enkelt att utvärdera resultaten jämförs varje lösning med avsikt på följande:

- Hastighet: hur lång tid tar en sökning bland data
- Enkelhet användning: hur enkelt blir det för en användare att utnyttja programmet
- Minne, allokering: hur mycket minne måste programmet allokera för att lösning ska vara genomförbar

- Minne, varaktigt: hur mycket plats kommer applikationen att ta upp i det varaktiga minnet.

Utöver detta finns för var och en lösningarna också en kort beskrivning av hur lösningen fungerar.

3.2.1 RecordStore

Översikt En RecordStore är ett objekt i J2ME vilken kan liknas vid en tabell i en databas. Varje RecordStore innehåller noll eller flera records, vilket motsvarar en tuple i en databas. En record kan dock inte, till skillnad från en tuple, innehålla flera olika celler utan är begränsad till en byte-array. På så sätt kan utvecklaren placera godtycklig data i en record men måste själv hålla reda på vilken information varje byte representerar. Utöver själva datat så innehåller en record också ett unikt ID vilken används för access till en specifik record. Detta kan jämföras med vektorindexering [7].

Hastighet En sekventiell sökning på 100 objekt tar cirka 1600 millisekunder på vår testenhet (Sony-Ericsson W800i). Detta är att betrakta som ytterst långsamt med tanke på att en katalog kan innehålla 100 000 poster eller mer. Testet utfördes genom att först skapa en RecordStore med 100 records innehållandes slumpmässiga bytes. Då detta var gjort hämtades informationen i varje record sekventiellt och jämfördes med ett användarinmatat tal. Var innehållet och talet lika så ökades en räknare och processen fortsatte. Då samtliga records undersökts skrevs antalet träffar samt tiden det tog ut. Anledningen till att sökningen inte avslutades vid första träffen var för att simulera det faktum att en katalog kan innehålla sökbara poster som inte är unika. En sådan sökning kommer med andra ord hitta alla förekomster av ett givet element vilket är ett måste för denna typ av applikation. Då en användare måste kunna tillförskansa sig data på något sätt undersöktes också hur snabbt den mobila enheten kunde skapa nya records. Försöket visade att skapandet tar 28,354 millisekunder per record. Detta skulle innebära att 1000 poster tar ungefär 28

sekunder att skapa.

Enkelhet - användare Då det tar tid att skapa records så kommer det också ta tid att överföra en katalog till den mobila enheten. Detta ställer visserligen inga tekniska krav på användaren men måste ändå beaktas då alla väntetider bör minimeras. Det får anses orimligt att anta att användaren accepterar en flera timmar lång installationsprocedur.

Minne - allokering (jmf. RAM) En RecordStore kräver lite overhead vad gäller minnesallokering. Det som tar plats är objektet RecordStore vilket är tillräckligt litet för att kunna allokeras i samtliga J2ME-kompatibla enheter. (Detta påstående stöds av att RecordStore är en del av J2ME som måste finnas implementerat.)

Minne - varaktigt (jmf. HDD) Förutom de data som sparas i varje record så håller systemet också internt en identifikator för varje record samt en header för varje RecordStore. Dessutom kan det finnas ytterligare implementationsberoende information som systemet sparar för att exempelvis kunna synkronisera data [7]. Ett enkelt test genomfördes på Sony-Ericsson W800i för att fastställa exakt hur mycket overhead denna lösning innebär (observera att detta är implementationsberoende och därför bara ger en fingervisning). Metoden getSize() returnerar, i bytes, hur stort utrymme en given RecordStore har tagit i anspråk, inklusive samtliga overheads [7]. För att ta reda på hur mycket overhead varje RecordStore kräver skapades en sådan utan records. getSize() returnerade då 48 vilket alltså innebär en overhead på 48 byte för varje RecordStore. Därefter lades en record till i RecordStore:n vilket gav värdet 80 byte. I denna record placerades en byte data vilket alltså innebär en overhead på 31 byte ($80 - 48 - 1 = 31$). För att vara säkra på att den första record:en i en RecordStore inte kräver extra overhead skapades ytterligare en, med samma resultat - 31 byte overhead.

3.2.2 File Connection Optional Package

Översikt File Connection Optional Package (FCOP) är ett tilläggspaket för J2ME vilket inte behöver implementeras av alla mobila enheter som stödjer J2ME. För de enheter som har stödet så fungerar FCOP analogt med strömmar i J2SE med avvikelsen att filaccess bara kan göras sekventiellt. Det finns inget sätt att få 'random access' i filer [12].

Hastighet Huruvida implementationen av FCOP är snabb eller inte har ej undersökts då varje försök att öppna en ström till filsystemet måste bekräftas av användaren (Se Enkelhet användare). Detta skulle innebära ett antal extra knapptryckningar för användaren och som resultat skulle applikationen upplevas som långsam. Genom att få applikationen certifierad av en tredje part kan detta problem på vissa enheter undvikas. Detta är dock något som uppdragsgivaren vill undvika i ett första skede.

Enkelhet - användare För att öppna en filström i J2ME anropas objektet Connector. Varje sådant anrop kommer att generera en fråga till användaren om huruvida applikationen ska få göra denna typ av anslutning. Detta skulle försvåra användandet av applikationen. Dessutom finns ingen form av 'random access' för filer i J2ME vilket innebär att fler filer måste skapas, innehållandes olika typer av information, för att snabba upp sökningar. Detta skulle generera ytterligare förfrågningar och agera irritationsmoment för en användare. Då FCOP inte måste finnas implementerat på alla enheter som stödjer J2ME så kan det finnas användare vars enheter i övrigt uppfyller kraven (på arbetsminne, J2ME-kompatibilitet, varaktigt minne osv.) men som inte stödjer FCOP. Dessa användare skulle då uteslutas från kundkretsen.

Minne - allokering FCOP kräver ingen extra minnesallokering förutom de objekt som skapas för att hantera filerna.

Minne - varaktigt I likhet med andra filsystem så innebär varje fil som skapas på en mobil enhet också en overhead. Denna kommer sig av att varje fil tilldelas ett visst minnesblock där blockstorleken är avgörande för hur stor overheaden blir. Detta är helt beroende av filsystemet på den mobila enheten och kan därför variera.

3.2.3 Data som del av programmet

Översikt All data som applikationen behöver finns tillgänglig i koden i form av exempelvis vektorer. Lösningen bygger på att katalogen är fix, det vill säga, den ändrar sig inte. Detta innebär i sin tur att då en användare laddat hem programvaran och data så ska inte användaren kunna ändra på data. Således kan all data ligga inkodat i programmet.

Hastighet Denna lösning är mycket snabb. En sökning bland 100 000 heltal tar på testenheten mellan 4 och 5 millisekunder. Testet utfördes genom att skapa en statisk vektor innehållandes 100 000 slumpmässiga (pseudoslump) heltal mellan 0 och 19999. Applikationen tog sedan ett användarinmatat tal och sökte upp alla förekomster av talet genom en enkel sekventiell sökning.

Enkelhet - användare Inga extra krav ställs på användaren med denna lösning.

Minne - allokering En post i den tänkta applikationen ska innehålla information om namn, efternamn, gatuadress, postadress, telefonnummer och geografiska koordinater. Preliminära beräkningar visar att en sådan post, som lägst, tar upp 103 bitar. För 100 000 element skulle detta innebära en total minnesanvändning på 1,3 MB. Sun, företaget bakom J2ME, menar att det tillgängliga arbetsminnet för en Java Virtual Machine implementerad på en mobil enhet kommer att ligga mestadels under 1 MB, för nästa generations mobiltelefoner [11]. (Artikeln är skriven 2001 vilket innebär att det som omtalas som nästa generations mycket väl kan vara dagens.) Vår applikation skulle kräva mer i allokerat minne än vad som kan förväntas finnas tillgängligt.

Minne - varaktigt Inga extra varaktiga minnesresurser tas i anspråk av denna lösning.

3.2.4 Data i resursfiler (.jar)

Översikt JAR står för Java ARchive och är ett sätt att distribuera applikationer. Alla klassfiler som applikationen kräver placeras i en JAR-fil vilket underlättar distribueringen. Förutom klassfiler kan JAR-filen också innehålla olika typer av resursfiler. Det finns inga krav på vad resursfilerna kan innehålla vilket leder till att det är ett sätt att lagra data. I J2ME finns ingen möjlighet att skriva till JAR-filen vilket gör att all data som placeras i filen är 'read-only', vilket också är det enda som krävs för vår tänkta applikation [12].

Hastighet En sekventiell sökning på 100 000 heltal tar mellan 2600 millisekunder och 2700 millisekunder. Testet utfördes genom att skapa en binär fil, test.txt, innehållandes 100 000 slumpmässiga heltal mellan 0 och 19999, på en persondator. Filen lades till i JAR-paketet och fördes över till den mobila testenheten. Här itererade enheten igenom 100 000 gånger och läste för varje iteration in 4 byte och omtolkade dessa till heltal.

Enkelhet - användare Lösningen är helt transparent för användaren.

Minne - allokering Ingen extra allokering krävs för denna lösning, utöver minnet för att skapa objekt vilka kan hantera strömmen från JAR-filen.

Minne - varaktigt Lösningen är helt analog med den ovan angivna via FCOP vad gäller overhead. Det som markant skiljer de båda åt är att JAR-filer kan komprimeras med formatet ZIP, vilket innebär en minskning av det utrymme som krävs i det varaktiga minnet [21].

3.2.5 PointBase Databashanterare

PointBase är en tredjepartstillverkad databashanterare för Java 2 Micro Edition, skriven för att fungera på Connected Limited Device Configuration (CLDC). Implementationen ligger utanför allmän kännedom då det inte handlar om mjukvara skriven med öppen källkod. Det som kan sägas är att PointBase uppvisar prestanda i likhet med den för RecordStores (enligt ovan). Detta är en indikation på att PointBase har löst problemet via just RecordStores, vilket också bekräftats från företaget. De har även en version där endast läsning från databasen är möjlig. Allt företaget säger om denna lösning är att databasen då ligger inpackad med applikationen (i samma JAR-fil). Detta är med andra ord analogt med den lösning som föreslagits ovan (Data i resursfiler).
Med anledning av likheterna mellan PointBase och de lösningar vi redogjort för under tidigare avsnitt så har inte applikationen undersökts vidare. Applikationen är också förknippad med ett relativt högt pris vilket uppdragsgivaren vill undvika.

3.2.6 Resultat

Av de undersökta lösningarna så är data som del av programmet den absolut snabbaste lösningen. Den omöjliggörs dock på grund av begränsningarna hos dagens mobila enheter vilka inte tilldelar nog med arbetsminne till den virtuella maskinen. Att spara data i RecordStores går långsamt men den verkliga nackdelen är att det tar för lång tid att skapa dem. Då applikationen startas för första gången kommer det krävas att runt 100 000 poster skapas vilket skulle ta 47 minuter på testenheten. (Märk att vi inte testat att lägga till mer än 1000 poster och att operationen inte är linjärt beroende av tiden. Det tar längre tid per post desto fler poster som läggs till!) De två alternativen vilka handlar om filaccess lider båda av att J2ME inte har stöd för 'random access'. Detta leder till långa åtkomsttider till filsystemet. FCOP har dessutom problemet att en användare måste godkänna att applikationen öppnar filströmmar. Databashanterare så som PointBase kan inte heller gå runt dessa problem utan är beroende av det API som J2ME erbjuder. Då kostnaden för Point-

Tabell 3.1: Huvudargument för val av lösning

Lösning	Argument för/emot
RecordStore	Tar för lång tid att infoga data
FCOP	Ställer för stora krav på användaren
Data som del av programmet	Inte möjlig, tar för mycket plats i arbetsminnet
PointBase	Kostnaden för applikationen för hög
Data i resursfil	Har inget av problemen ovan. Lång åtkomsttid kan förbättras.

Base av uppdragsgivaren anses för stor är det således inte ett alternativ. Sammantaget leder detta till slutsatsen att lagring av data bäst sker via resursfiler i JAR-filen. För att öka åtkomsttiden kan en virtuell 'random access' skapas (Se avsnitt 3.3). I tabell 3.1 syns huvudargumentet för detta.

3.3 Organisera data

I analysen *Att spara data* så är slutsatsen att det mest lämpliga sättet att spara data är via resursfiler i JAR-paketet (se avsnitt 3.2). Denna lösning kräver dock vissa optimeringar då det tog mellan 2600 - 2700 ms att söka bland 100.000 element och vi har enligt kravspecifikationen endast totalt 500 ms (Bilaga B). Problemet bottnar i att det i J2ME inte finns något stöd för *Random Access Files*, dvs. filer där vi kan flytta filpekaren till önskad offset hur vi vill. Detta beror på att man i J2ME endast hanterar filer som strömmar (eng. streams) via *File Connections* [12]. Det här avsnittet ger en analys av tänkbara lösningar för att organisera data. Varje lösning kommer att analyseras utifrån följande koncept:

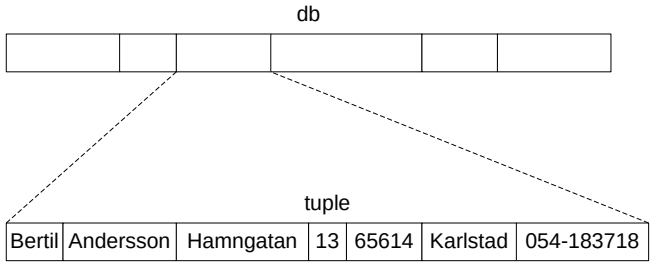
- Hastighet: Tiden det tar att utföra en sökning.
- Minne: Mängden minne som filen/filerna tar i anspråk.

Utgångspunkten för analysen i detta avsnitt baseras till stora delar på Lokaldelens katalog över Uppsala som vi har erhållit som xml-dokument.[6]
En fördel som vi vill dra nytta av är att informationen i katalogen är konstant. Vi har av den anledningen inget behov att ta hänsyn till att:

- ny information kan läggas till
- existerande information kan modifieras
- information kan raderas

3.3.1 Data i en fil

Översikt En enkel lösning är att lägga all data i sekvens i en och samma fil. Varje tuple ligger då efter varandra och innehållet för varje tuple separeras genom ett specialtecken, t.ex. semikolon (Se Figur 3.1).



Figur 3.1: Data i fil

Hastighet Denna metod tar inte hänsyns till hastigheten för att söka överhuvudtaget. En sökning måste ske sekventiellt (även om det finns tillgång till *random access filer*) och söksträngen måste jämföras med samtliga element i hela databasen. I det test vi har utfört med att söka 100.000 heltal på detta sätt gav en söktid på 2600 - 2700 ms. Att söka strängar på motsvarande sätt skulle ta mer tid än så eftersom strängar inte är en primitiv

[6]Lokaldelen för Uppsala innehåller c:a 88000 privatpersoner och 5000 företag. Detta är enligt uppgift den största katalogen med avseende på antalet poster.

typ, kan vara av variabel längd och innehåller mer information än ett heltal. I den fortsatta utredningen antas dock att en jämförelse kräver en instruktion, där det med instruktion menas att någon typ av operation ska utföras. Ett sådant mått tar alltså inte hänsyn till om det är heltal eller strängar som jämförs.

Minne Denna metod är, i relation till dess prestanda i sökhastighet, inte minnessnål. Det existerar mycket redundant information som ligger utspritt i databasen då t.ex. många personer har samma namn, bor på samma gata eller har samma postnummer. Den här lösningen är inte optimerad för denna typ av redundant information.
För att beräkna den totala storleken för en godtycklig post i databasen, i syfte att kunna jämföra med de andra lösningarna, så definierar vi parametrarna n och g:

$$\begin{aligned} n &= \text{totala antalet poster} \\ g &= \text{antal tecken i genomsnitt per post} \end{aligned} \tag{3.1}$$

Totala antalet bitar s för en godtycklig post blir då[7]:

$$s = 8 \cdot g \cdot n \,\text{bitar} \tag{3.2}$$

I Uppsalakatalogen finns det $n = 87891$ personer. I genomsnitt består efternamnen av c:a $g \approx 7,46$ tecken per post. Applicerar vi formeln från ekvation 3.2 för efternamnen i Uppsala så blir den totala storleken s:

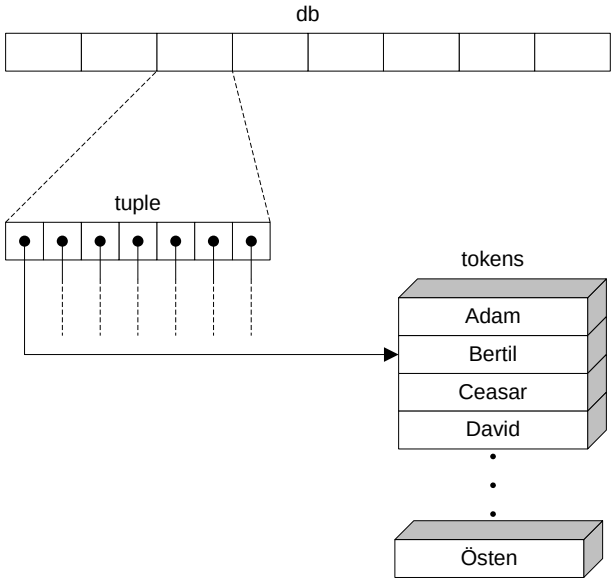
$$s = 8 \cdot g \cdot n = 8 \cdot 87891 \cdot 7,46 \approx 640\,\text{Kb} \tag{3.3}$$

3.3.2 Token-tabell

Översikt För att optimera minnesutrymmet när det gäller redundant information, som beskrevs tidigare, så kan man lägga den informationen i en egen tabell. I en sådan tabell

[7] 1 tecken = 8 bitar

finns t.ex. alla **unika** efternamn samlade på ett organiserat sätt, exempelvis i bokstavsordning. För varje relevant fält i databasen, d.v.s. där redundant information förekommer, konstrueras motsvarande tabeller. För varje tuple i databasen ligger nu istället *pekare*. Pekarna hänvisar till var i de olika tabellerna som informationen för just den tupeln finns. Detta illustreras i Figur 3.2.



Figur 3.2: Token-tabell

Hastighet Den här metoden skiljer sig inte nämnvärt från den föregående gällande sökhastighet. Lite längre söktid kan förväntas då man först måste göra ett uppslag i tabellen för att veta vilket *pekarID* man ska leta efter i databasen.

Minne Hur mycket minne den här lösningen kräver beror till stor del på egenskaperna och karaktären i databasen; mycket redundant information medför bättre minnesutnyttjande. Det innebär att man måste analysera vilka av fälten som är lämpliga att ha tabeller för. Vilka parametrar kan påverka ett sådant beslut? Förutom de parametrar vilka intro-

ducerades i ekvation 3.1 så lägger vi till:

$$p = \text{antal unika poster} \tag{3.4}$$

Från parametern p kan vi definiera b som antalet bitar som krävs för att representera en *pekare*:

$$b = \log_2(p) \tag{3.5}$$

Det finns totalt n poster i databasen och vi kan beräkna totala antalet bitar för *pekarna* i databasen som $b \cdot n$. Storleken för tabellen beräknas genom att multiplicera det genomsnittliga antalet tecken per post med antalet unika poster: $g \cdot p$. Sammantaget blir den totala storleken s alltså:

$$s = b \cdot n + 8 \cdot g \cdot p \,\text{bitar} \tag{3.6}$$

Storleken för en post beräknat på formeln från ekvationen 3.6 ovan bör vara mindre än storleken för samma post beräknat på formeln från ekvation 3.2, vilket ger:

$$b \cdot n + 8 \cdot g \cdot p < 8 \cdot g \cdot n \tag{3.7}$$

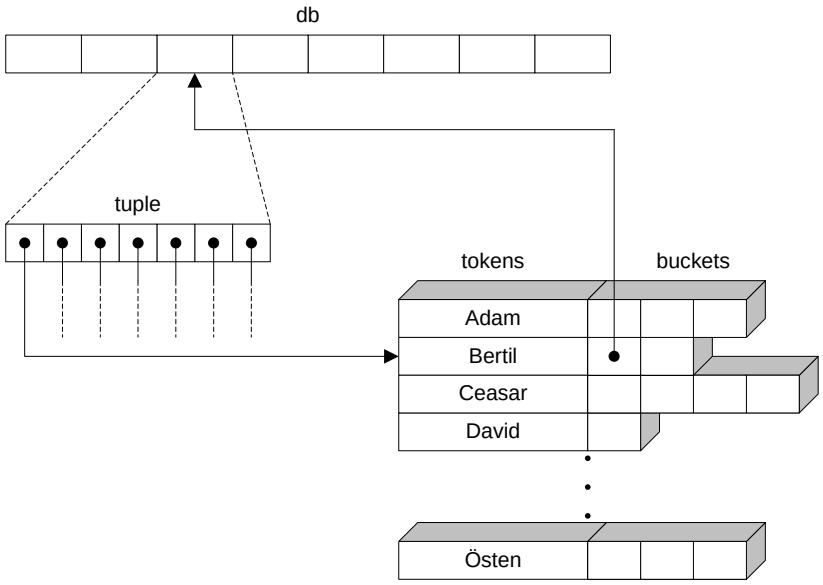
I Uppsalakatalogen finns det $n = 87891$ personer och $p = 20027$ unika efternamn. I genomsnitt består efternamnen av c:a $g = 7,46$ tecken per post. Applicerar vi formeln från ekvation 3.6 så blir storleken s för efternamnen i Uppsala totalt sett:

$$s = b \cdot n + 8 \cdot g \cdot p = \log_2(20027) \cdot 87891 + 8 \cdot 7,46 \cdot 20027 \approx 172\,\text{Kb} \tag{3.8}$$

3.3.3 Token-tabell med buckets

Översikt För att öka sökhastigheten kan tokentabellen utökas med pekare till relevanta poster i databasen. Som tidigare finns varje unik förekomst av exempelvis ett efternamn i tokentabellen. Associerad med varje post i tabellen finns också en bucket innehållandes

pekare till de databas-tuplar vilka har den egenskap som beskrivs av tabellposten. (Se Figur 3.3.)



Figur 3.3: Token-tabell med buckets

Hastighet Sökhastigheten förbättras eftersom sökalgoritmen inte behöver gå igenom hela databasen utan endast tokentabellen (förutsatt random access) vilken, enligt 3.3.2, innehåller färre poster. Dessutom, då en förekomst väl har hittats i tokentabellen så fås alla *pekarID* för förekomster i databasen genom undersökning av bucket:en. Förutsatt att det går att hämta data i databasfilen med random access så krävs det nu alltså först x instruktioner för att hitta posten (och tillhörande *pekarID*) i tokentabellen. Därefter krävs det en instruktion för varje *pekarID* i bucket:en. I genomsnitt så innehåller en bucket $\frac{n}{p}$ *pekarID*, vilket alltså resulterar i lika många instruktioner. Den tidigare beskrivna lösningen behöver, precis som denna, x instruktioner för att hitta rätt *pekarID* i tokentabellen. Därefter krävs n genomsökningar i databasen. Det genomsnittliga antalet instruktioner för att hitta rätt

token i tabellen är $x = \frac{p}{2}$. Formlerna för den totala accesstiden blir då:

$$buckets = \frac{p}{2} + \frac{n}{p} \tag{3.9}$$

$$sekventiell = \frac{p}{2} + n \tag{3.10}$$

Vi har med andra ord minskat söktiden till $\frac{buckets}{sekventiell} \cdot 100$ procent av tidigare söktid. Sätts värdena för Uppsalakatalogens efternamn in fås:

$$\frac{\frac{20027}{2} + \frac{87891}{20027}}{\frac{20027}{2} + 87891} \approx 0,10 \tag{3.11}$$

Detta innebär en förbättring med 90 % av tidigare algoritm. Den nya algoritmen tar med andra ord 10 % av den tidigare.

Minne Denna lösning har bättre hastighet men då det för varje post i databasen nu krävs en pekare från tokentabellen kommer minnesanvändningen påverkas negativt. För varje tuple (n st) krävs $\log_2(n)$ bitar vilket innebär att strukturen nu tar upp ytterligare $\frac{n \cdot \log_2(n)}{8}$ bytes. En avvägning får då göras mellan sökhastighet och minnesutnyttjande. Vi ställer upp formler för tokentabellens storlek i de båda fallen:

$$buckets = g \cdot c \cdot p + n \log_2(n) \tag{3.12}$$

$$sekventiell = g \cdot c \cdot p \tag{3.13}$$

Där

g = genomsnittligt antal tecken per token

c = antalet bitar för att representera ett tecken

n = antalet tupler i databasen

p = antalet unika token i tokentabellen

Precis som ovan blir förhållandet mellan denna algoritm och den angivna i 3.3.2 $\frac{buckets}{sekventiell}$. Sätts värdena för Uppsalakatalogens efternamn in i denna ekvation fås:

$$\frac{7,46 \cdot 8 \cdot 20027 + 87891 \log_2(n)}{7,46 \cdot 8 \cdot 20027} \approx 2.21 \tag{3.14}$$

Detta innebär en dryg fördubbling av minnesanvändningen. Sökhastigheten har dock förbättrats med nästan 90 % vilket innebär att lösningen som helhet är en förbättring.

3.3.4 Konstgjord Random Access

Översikt Då det inte finns något stöd för *Random Access Files* i J2ME så begränsas möjligheterna att söka och traversera i filer. De abstrakta modeller vi har diskuterat hittills i kapitlet bygger på att mycket information ligger i ett fåtal filer och att man kan peka till en given offset i en fil momentant. Dessvärre tillåts vi inte att söka i filer på detta sätt men är i behov av den funktionaliteten för att erbjuda önskad hastighet i sökningar.
Genom empiriska studier har vi upptäckt att hastigheten för att öppna en given fil inuti ett JAR-paket är mycket snabb. Med denna kunskap så konstruerades ett JAR-paket bestående av c:a 6000 filer indelade i olika kataloger och mätningar på hastigheten för att öppna filer uppmättes. Experimentet visade ingen signifikant skillnad i hastighet jämfört med att öppna filer i ett JAR-paket bestående av färre antal filer.
Man kan distribuera informationen som var tänkt att ligga i en stor fil i ett stort antal mindre filer och på så sätt snabba upp sökhastigheten markant. Vi har valt att kalla detta för *Konstgjord Random Access* då vi åstadkommer ungefär samma funktionalitet. Begreppet *filpekare* i denna kontext innebär alltså **vilken** fil man ska söka i.

Hastighet Hastighetsförbättringen är av mycket stora proportioner jämfört med att lägga mycket information i en enda fil där man tvingas söka sekventiellt. Hur mycket sökhastigheten förbättras är naturligtvis beroende av mängden information men i synnerhet på hur många filer man delar upp den i. Vi kan konstatera att sökhastigheten bland n

element i en sekventiell fil tar $\frac{n}{2}$ instruktioner[13]. Om man bortser från den korta tid det tar att öppna önskad fil (uppskattningsvis < 1 millisekund) så kan vi beräkna sökhastigheten för att distribuera informationen i b antal filer[8] genom följande ekvation ($b \leqslant n$):

$$\frac{\frac{n}{b}}{2} = \frac{n}{2 \cdot b} \tag{3.15}$$

Om vi sedan jämför detta med hastigheten för sekventiell sökning i en fil så erhåller vi följande förhållande:

$$\frac{\frac{n}{2}}{\frac{n}{2 \cdot b}} = \frac{1}{b} \tag{3.16}$$

Det tar alltså $1/b$ av antalet instruktioner för att genomföra en sökning jämfört med en sekventiell sökning, eller;

Teorem 1. Det går b gånger så fort att söka där informationen är distribuerad i b filer än om den är i en fil.

Minne Det innebär en overhead för varje extra fil, som är minst lika med storleken för filnamnet. delar man dessutom upp det i en katalogstruktur så blir det en extra overhead för varje katalog, som är minst lika med storleken för katalognamnet. Det krävs, utöver det redan nämnda, också information för att beskriva hela fil- och katalog-strukturen i JAR-paketet.

3.3.5 Resultat

Våra intentioner är att kunna hämta information snabbt utan att tvingas lägga för mycket overhead för att uppnå detta. Den bästa metoden för att erbjuda snabb access till informationen anser vi är genom att distribuera data i ett "stort" antal filer. Hur många filer som anses vara "stort" beror egentligen på hur mycket information som ska sparas. Våra

[8] I en sekventiell sökning är $b = 1$.

beräkningar visar att storleken på de distribuerade filerna inte bör överstiga 30 Kb om den sekventiella sökningen, som måste ske, inte ska ta för lång tid (lång tid = mer än 500 ms). Detta leder till att antalet filer, b, beror på storleken s (beräknat i Kb) enligt sambandet:

$$b = \frac{s}{30}\text{filer} \tag{3.17}$$

Vi har en databasfil på c:a $s = 3\text{MB} = 3072\text{Kb}$ vilket, enligt ekvation 3.17, alltså leder till att antalet filer $b = 3072/30 \approx 103$.

3.4 Indexera data

För att kunna hitta den fil som innehåller det token man söker så krävs att filerna är indexerade på något sätt. Det finns flera tekniker att indexera data för sökning; Binära sökträd [1, 20, 13], multilevel träd som B-träd och B+träd [6, 22, 13], hash funktioner [13, 1, 6, 22, 20], trie [13, 15, 18] och PATRICIA-träd [13] bara för att nämna några. Vi är i synnerhet intresserade av att söka på strängar t.ex. förnamn eller efternamn. Detta ställer speciella krav på indexeringen jämfört med indexering av heltal. De tekniker vi har tittat närmare på är *trie hashing* [15, 18] och *simple prefix B-tree* [2, 22] vilka är de av litteraturen föredragna indexeringsmetoderna för strängar.
För att kunna söka på telefonnummer så måste vi, förutom att söka på strängar, också kunna söka på heltal. Vi har valt att göra detta genom att använda ett vanligt binärt sökträd (BST), av den anledningen att det är ett enkelt och mycket snabbt sätt att indexera på. Alternativen hade bl.a. varit att använda en hashfunktion eller t.ex. ett B+träd.

3.4.1 Trie hashing

Översikt För att söka på stora mängder av långa alfanumeriska strängar föreslog Litwin en metod kallad *trie hashing* [15, 18]. Det är en hashstruktur som, i motsats till traditionella hash-algoritmer där data lagras osorterad baserat på algoritmen, lagrar data sorterad och

Tabell 3.2: Ordning för hur efternamnen sattes in

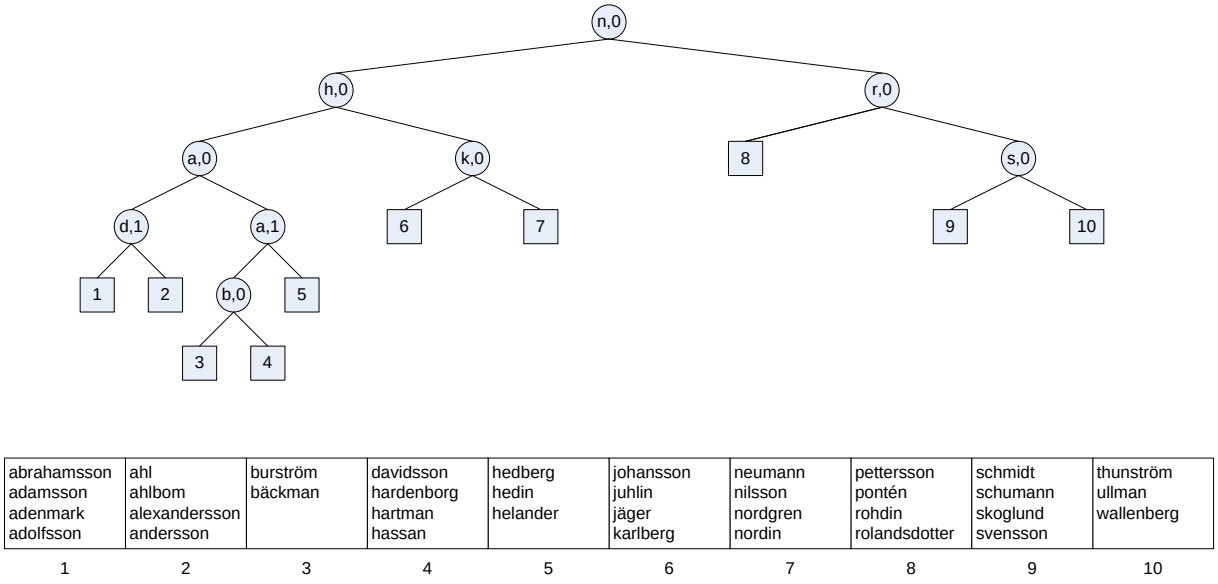
schumann, thunström, adolfsson, helander, *neumann*, rolandsdotter, pontén, hartman, *hardenborg*, *hedin*, *hedberg*, *svensson*, adenmark, nilsson, burström, karlberg, hassan, nordgren, wallenberg, abrahamsson, *nordin*, jäger, andersson, *ahlbom*, *bäckman*, *skoglund*, pettersson, davidsson, alexandersson, schmidt, rohdin, juhlin, johansson, ullman, ahl, adamsson

beskrivs som ett modifierat binärt sökträd (tillsammans med en algoritm). Detta gör den dynamisk och samtidigt snabb att söka i.
Lövnoderna i trädet består av s.k. hinkar (eng. *buckets*) och innehåller textsträngarna som är sökbara. De inre noderna i trädet har till uppgift att "guida" sökningen till rätt hink. När en sökning har nått en hink så sker en sekventiell sökning för att matcha söksträngen. Figur 3.4 beskriver en liten *trie hashing*-struktur för några efternamn. Det som syns nedanför trädstrukturen är innehållet i de olika hinkarna som refereras m.h.a. siffror i lövnoderna i trädet. Ordningen för hur efternamnen lades in i trädet illustreras i tabell 3.2. De efternamn som är kursiva har orsakat en *overflow* i en hink och medfört att den delats upp i nya hinkar. I exemplet så används en hinkstorlek $b = 4$.

Hela trädstrukturen finns lagrad i primärminnet och hinkarna ligger på disken. Ett träd av denna struktur innebär att endast **en** diskläsning är nödvändig när en sökning sker.

3.4.2 Simple prefix B-tree

Översikt Simple prefix B-tree är en specialvariant av ett vanligt B-träd utformat för strängar. Ett B-träd är en multilevel trädstruktur vilket innebär att noderna i trädet kan ha m barn (till skillnad från binära träd där varje nod endast kan ha 2 barn). Egenskapen för hur många barn en nod kan ha benämns som *graden för trädet*. Minsta antalet barn för en given nod är hälften av *graden för trädet*. Trädet har även den egenskapen att det alltid är balanserat vilket innebär att en sökning alltid sker med samma hastighet.



Figur 3.4: Exempel på ett Trie Hashing

Sökning i trädet sker på liknande sätt som i ett binärt träd fast i en nod med m barn så ska sökuttrycket x jämföras mot nycklarna $k_1, k_2, \ldots, k_{m-1}$. Barnen $\alpha_1, \alpha_2, \ldots, \alpha_m$ för den aktuella noden har följande relation till nycklarna:

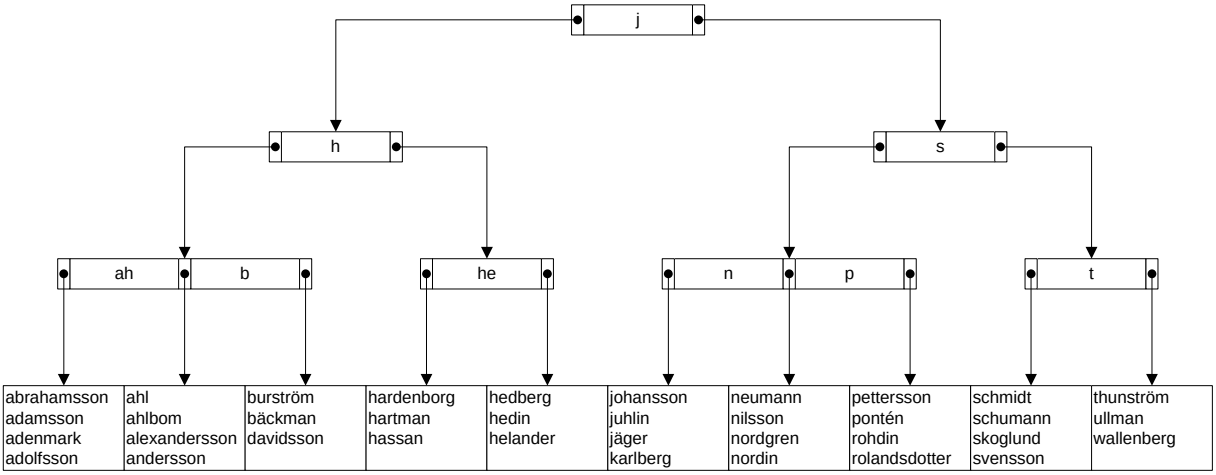
$$\alpha_1 < k_1 \leqslant \alpha_2 < k_2 \leqslant \cdots < \alpha_{m-1} < k_{m-1} \leqslant \alpha_m \tag{3.18}$$

Det implicerar att vägen för den fortsatta sökning för x blir till nod: α_1 omm $x < k_1$, α_2 omm $k_1 \leqslant x < k_2$, $\cdots$, α_{m-1} omm $k_{m-2} \leqslant x < k_{m-1}$, α_m omm $k_{m-1} \leqslant x$

Ett simple prefix b-tree indexerar strängarna genom att enbart spara det kortaste prefixet nödvändigt som nyckel för en nod. Jämförelsen mellan strängarna sker lexografiskt.

3.4.3 Binära sökträd

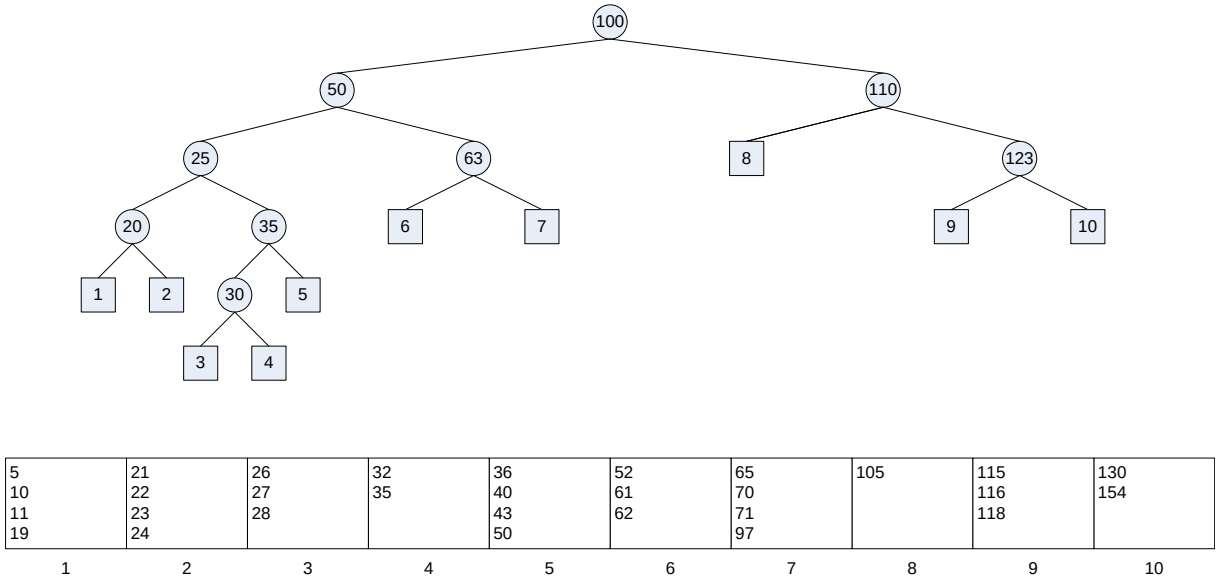
Översikt I avsnitten 3.4.1 och 3.4.2 beskrevs metoder för att indexera strängar. Vissa typer av data lämpar sig dock inte för att sparas ner som en sträng, däribland tele-



Figur 3.5: Exempel på ett simple prefix B-tree

fonnummer. Ett telefonnummer består uteslutande av siffror [9] och därav lämpar det sig bättre att spara telefonnummer som heltal. Dels för att spara minne, men också för att hantering av primitiva typer är snabbare och i många avseenden enklare. Detta beslut får alltså konsekvenser även för indexeringen då strängjämförelser lämpar sig dåligt för heltal. (Olämpligt eller inte, det är möjligt att konvertera emellan de båda datatyperna och sedan jämföra.) Ett enkelt och effektivt sätt att söka bland heltal (nycklar) är att använda sig av ett binärt sökträd. Strukturen liknar den som används vid trie hashing (se avsnitt 3.4.1) vilket syns i figur 3.6. (Figuren har av pedagogiska skäl inte verkliga telefonnummer.) Det som sker vid en sökning är samma sak som sker implicit vid binärsökning, förutsatt att sekvensen som genomsöks är sorterad [13]. Binärsökning fungerar genom att titta i mitten på sekvensen S_1. Detta delar upp sekvensen i två lika stora delsekvenser, S_{11} och S_{12}. Här är alltså (på grund av sorteringen) alla element i S_{11} mindre än de i S_{12}. Är det sökta elementet nu mindre än det mittersta i S_1 jämförs det istället med det mittersta i S_{11} och analogt med det mittersta i S_{12} om det sökta elementet är större än det mittersta i S_1. Denna procedur upprepas rekursivt till rätt element påträffas eller sökningen termineras

[9] Här bortses från all typsättning som kan göras på ett telefonnummer. Denna typsättning, exempelvis bindestreck mellan rikt- och huvudnummer, saknar informationsvärde eftersom den är helt typografisk.



Figur 3.6: Exempel på ett binärt sökträd

genom att delning av delsekvenser inte längre är möjlig. Det som sker här är alltså en implicit uppbyggnad av ett binärt sökträd. Dessa två tekniker är ur söksynpunkt ekvivalenta, men för vissa ändamål passar ett explicit träd bättre. Detta gäller främst då insättning och borttagning av element ur sekvensen/trädet sker med en hög frekvens [13]. I vårt fall blir valet därför ett explicit träd. Kravspecifikationen (se bilaga B) klargör att insättning från den egna telefonboken till katalogen ska vara möjlig. Det innebär att insättning kan komma att ske vilket som sagt gör att ett explicit träd är att föredra. Dessutom är den konceptuella modellen av ett explicit träd enklare att förmedla till uppdragsgivaren och därför lättare att underhålla.

3.4.4 Resultat

Detta avsnitt har beskrivit tre olika metoder för att indexera data. Två av dessa, trie hashing och binära sökträd, har använts. Binära sökträd har använts för att alternativen inte hanterar insättning och borttagning av element på ett tillfredställande sätt. Detta är inget problem med den nuvarande releasen men för att vara förberedd på de krav som

ställs på kommande releaser så är det här en faktor.
Anledningen till att använda trie hashing istället för ett simple prefix b-tree är för att trie hashing är en enklare och mer "lättviktig" (binär, mindre information i varje nod) struktur. Detta är viktiga egenskaper då vi har begränsade resurser i form av beräkningskapacitet och minne. Eftersom trie hashing är binär kan vi ladda in strukturen på liknande sätt som det binära sökträdet. Trie hashing kan beskrivas med en finit automaton vilket innebär att strukturen istället finns lagrad som kod i programmet och gör den mycket snabb att söka i. Metoder för detta finns beskrivet i [13].

3.5 Presentera data

Detta avsnitt är ägnat åt att reda ut vilka möjligheter J2ME ger för att presentera data. Först undersöks vilka komponenter som finns i paketet *javax.microedition.lcdui* och om dessa kan skräddarsys för de krav som ställs på applikationen enligt bilaga B. Om så inte är fallet undersöks huruvida det finns tredjepartsbibliotek vilka löser de problem som stötts på.

3.5.1 J2ME enligt javax.microedition.lcdui

J2ME innehåller en rad klasser för att rita upp ett grafiskt användargränssnitt. Dessa finns alla samlade i java-paketet javax.microedtion.lcdui. Grundtanken med detta bibliotek är att applikationer som använder sig av det ska fungera på alla enheter som stödjer CLDC1.1/MIDP2.0 vilket gör att klassernas funktionalitet begränsas av enheter med mycket liten skärm, få eller inga färger och komplicerade gränssnitt för användarinmatning. Det vida spektrat av enheter vilka stödjer MIDP2.0 gör att API:et för grafiska gränssnitt blir mycket begränsat. Det finns dock två sätt att lösa problemet:

- Abstraktion: Låt MIDP-implementationen styra så mycket som möjligt. Istället för att låta programmeraren avgöra exakt vad som ska ritas upp, hur det ska ritas och

var, låt programmeraren säga vad och MIDP säga hur och var. På så sätt är det upp till enheten att avgöra hur gränssnittet fungerar och programmeraren kan endast avgöra vilka komponenter som ska visas samt deras innehåll (data).

- Förfrågning: Programmeraren frågar enheten vilka typer av komponenter som stöds och i vilken utsträckning. Beroende på svaret så ritas gränssnittet upp, helt på programmerarens premisser.

Båda dessa lösningar stöds i MIDP, där den förra är att föredra då det kräver mindre arbete från programmerarens sida[12]. Enligt kravspecifikationen ska användaren behöva göra maximalt en knapptryckning (utöver inmatningen av text) för att påbörja en sökning. Detta innebär att direkt då programmet startas ska användaren kunna påbörja inmatningen av söksträngen. Då söksträngen är färdigskriven trycker användaren på exakt en knapp för att påbörja sökning. Här räcker dock inte abstraktionslösningen till, då de komponenter som finns för textinmatning är konstruerade så att användaren först måste genomföra en knapptryckning för att påbörja själva inmatningen. Enda sättet att komma undan problemet är att använda förfrågningslösningen. Olyckligtvis leder även detta till problem. Applikationen ska enligt uppdragsgivaren fungera på de tio vanligaste mobiltelefonmodellerna i Sverige. Med förfrågningsmodellen innebär detta att applikationen måste anpassas till tio skilda modeller vilket i sin tur innebär att mycket utvecklingstid kommer att läggas på design av gränssnittet. Då tiden som avsätts för projektet redan är knapp måste en annan lösning hittas.

3.5.2 J2ME enligt de.enough.polish.ui

J2ME Polish är ett Java-bibliotek skrivet i „open-source„-anda för att underlätta utvecklandet av skräddarsydda program i J2ME. Det använder sig av ovan beskrivna förfrågningsmodell för att själv avgöra hur komponenterna ska ritas upp. Förfarandet ställer större krav på utvecklaren men ger också fler möjligheter till densamma. Det är inte längre upp till enheten

att avgöra hur komponenterna ritas upp eller vilka egenskaper de ska ha. Problemet med att programmera till tio olika plattformar tar Polish hand om genom att också inkludera en rad verktyg för preprocessing. Allt utvecklaren behöver göra är att tala om för preprocessorn vilken modell programmet ska köras på och Polish tar själv, via en enhetsdatabas, reda på vilka förutsättningar den modellen har [27]. Detta innebär en rad fördelar, bland annat:

- Utvecklaren behöver inte själv programmera för en specifik enhet.
- Utvecklare som vill skriva för flera enheter själv slipper programmera förfrågningen.
- Gränssnittet kan designas helt efter utvecklarens tycke.
- Eftersom enheten skräddarsys vid kompileringstillfället finns inga förfrågningar med i den färdiga applikationen vilket innebär att programmet är mindre (viktigt för mobiltelefoner med begränsade minnesresurser) och effektivare (inga anrop till enheten med förfrågningar om vad enheten klarar av).

Naturligtvis finns också nackdelar varav några är:

- Enhetsdatabasen måste uppdateras för varje ny modell som släpps och utvecklare som vill använda de senaste teknikerna som endast stöds av nya modeller kan få problem då modellen inte finns i enhetsdatabasen.
- På grund av att applikationen skräddarsys till en viss modell via preprocessing så fås en applikation för varje modell vilket kan innebära distributionsproblem. Användare kan inte längre förvänta sig att använda samma applikation på alla plattformar. Detta är i direkt opposition mot en av Javas grundtankar.

3.5.3 Resultat

Då fördelarna vägs mot nackdelarna står det klart att den bästa lösningen för det generella fallet är att använda J2MEs abstraktionslösning. Här behöver programmeraren inte bry sig

om vilken typ av enhet som applikationen skrivs för och resultatet blir precis en applikation, inte en för varje modell. De krav som ställs på just vår applikation medför dock att valet faller på J2ME Polish vilket innebär att det kommer finnas en applikation för varje modell vi vill stödja.

3.6 Komprimera data

I detta avsnitt beskrivs en rad olika kompressionstekniker samt deras för- och nackdelar. Det har redan i tidigare avsnitt funnits inslag av komprimering genom användandet av tokentabeller och distribution av applikationen i ett JAR-paket. Här lyfts dock konceptet ut och en analys av vilken komprimeringsalgoritm som lämpar sig bäst för applikationen besvaras. För att kunna diskutera kring komprimering krävs dock först en utredning av begreppet entropi.

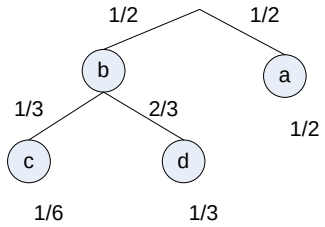
3.6.1 Entropi

Entropi var ursprungligen ett mått på oordning inom termodynamiken och det var först i och med Claude Elwood Shannon som begreppet fick en innebörd inom kommunikation. Han upptäckte att entropi är ett naturligt mått på informationsinnehåll. Entropi har ett tätt samband med den mängd valmöjligheter som finns då ett meddelande konstrueras. Om en situation har en hög grad av ordning så karakteriseras det inte av en hög grad av slumpmässighet eller valmöjlighet - det vill säga, informationsinnehållet är lågt om oordningen (entropin) är låg [25]. Entropi är med andra ord ett mått på osäkerheten, oordningen, i ett meddelande. Mer formellt, anta att det finns en mängd möjliga händelser vars sannolikheter för att inträffa är p_1, p_2, ..., p_n. Sannolikheterna är kända men det finns ingen ytterligare kännedom om vilken av händelserna som kommer att inträffa. Finns det nu något mått på hur osäkra vi är på utfallet, vilken händelse som kommer inträffa? Om det finns ett sådant mått, beteckna det $H(p_1, p_2, ..., p_n)$ så bör det uppfylla följande krav:

- H ska vara kontinuerlig över all p_i.

- Om alla p_i är lika, det vill säga, $p_i = \frac{1}{n}$, så ska H vara en strikt ökande funktion av n. Med lika sannolika händelser är osäkerheten större desto fler möjliga händelser det finns.

- Om ett val bryts ned i två successiva val så ska det ursprungliga H vara den viktade summan av de individuella värdena för H. Detta illustreras i figur 3.7



Figur 3.7: Ett träd där lövnoderna är händelser som beror på sina föräldrar.

där händelserna c och d beror på händelsen b medan a är obetingad [24]. H skall nu vara sådan att $H\left(\frac{1}{2}, \frac{1}{3}, \frac{1}{6}\right) = H\left(\frac{1}{2}, \frac{1}{2}\right) + \frac{1}{2}H\left(\frac{2}{3}, \frac{1}{3}\right)$. Uttryckt i ord så ska osäkerheten kring a, c och d vara lika stor som osäkerheten kring a och b plus osäkerheten kring c och d om b inträffar (vilket vi ju inte vet i förväg). Det kan bevisas att det enda H som uppfyller dessa krav är:

$$H = -K \sum_{i=1}^{n} p_i \log_x (p_i)$$

K och x är positiva konstanter av betydelse endast för vilken enhet resultatet önskas vara i[24]. Entropin för ett givet meddelande är alltså ett mått på den genomsnittliga osäkerheten i meddelandet. Om x i ekvationen ovan väljs att vara 2 och $K = 1$ så fås måttet i bitar. Det kan då tolkas som det genomsnittliga antalet bitar varje symbol i meddelandet har. För ett helt slumpmässigt (oordnat) meddelande är det genomsnittliga antalet bitar som krävs per symbol lika med det faktiska antalet bitar som symbolerna har. Annorlunda uttryckt, entropin är i sådana fall lika med det faktiska genomsnittliga antalet bitar [16][24].

Entropin för ett helt språk (med tillhörande alfabet) där symbolerna är beroende av tidigare

symboler i något givet meddelande kan vara svår att bestämma exakt. Bortser vi från beroendet blir problemet överskådligare då sådana beräkningar är mindre krävande att genomföra. Resultatet för Uppsalakatalogen blir:

$$H(U) = -\sum_{i=1}^{n} P(t_i) \log_2 (P(t_i)) \approx 5,26$$

där $P(t_i)$ är sannolikheten för tecken i. Om katalogen kodas med en standardimplementation av exempelvis zip ses dock en genomsnittslängd på 2,27 bitar per tecken. Detta är en indikation på att sannolikheten för ett givet tecken är betingat av tidigare tecken i meddelandet vilket sänker entropin. En osäker (men mer precis än ovan angivna) uppskattning av entropin för katalogen är alltså 2,27 bitar.

3.6.2 Strömkoder gentemot blockkoder

Det finns många sätt att klassificera koder på. Ett av dem är att dela in koderna utefter huruvida de arbetar på en ström av bitar/bytes eller på enskilda block, ett i taget. Strömkoder är sådana vilka kodar varje ny enhet baserat på tidigare enheter, exempelvis LZ-familjen och Aritmetisk kodning. Blockkoder arbetar med varje ny enhet som helt fristående från tidigare enheter, exempel är Huffmankoder och Fanokoder. [16] Den stora skillnaden mellan de båda grupperna är således beroendet mellan den enhet som ska kodas och tidigare, redan kodade, enheter. Vad gäller strömkoder är de mycket effektiva - Aritmetisk kodning kan med rätt modell skapa kompressionsgrader vilka tangerar entropigränsen. [16] Nackdelen med dessa är dock att en dekomprimeringsalgoritm måste börja avkodningen i meddelandets början, det vill säga, att börja avkoda på slumpmässig byte är inte möjligt. Detta beror på att strömkoden kommer att koda nya enheter olika, beroende på vilka tidigare enheter som kodats. En byte x kan alltså komma att kodas som bitsträngen y vid ett tillfälle i kodningen och som bitsträngen z vid ett senare tillfälle [3]. Denna brist finns inte hos blockkoder eftersom dessa kommer att koda alla likadana enheter på precis samma sätt,

oavsett var i meddelandet de förekommer. Denna egenskap är viktig då det är önskvärt att påbörja avkodningen av ett meddelande på slumpmässig plats i meddelandet, så som fallet är med exempelvis en komprimerad databasfil. Här finns sällan tid att avkoda hela filen och därefter genomföra en sökning. Istället är det att föredra att avkoda precis den bit information som är intressant. Då en användare till exempel söker efter en viss person i en telefonkatalogliknande databas, är det fördelaktigt om inte hela den fil i vilken den sökta posten ligger behöver avkodas, utan istället avkoda endast den sökta posten. Detta är alltså inte möjligt med strömkoder eftersom dessa är beroende av att se tidigare poster i databasen för att kunna avkoda den sökta. Av denna anledning koncentreras utredningen kring blockkoder.

3.6.3 Huffman

Det går att visa att Huffman är en optimal komprimeringsalgoritm om symbolerna i källan är oberoende av varandra [16]. I fallet med uppsalakatalogen finns dock ett starkt beroende mellan de olika symbolerna. En egen implementation av Huffman ger en genomsnittlig kodlängd per bokstav på ungefär 5,3 bitar. Detta är mycket nära entropin förutsatt att ett givet tecken inte har något beroende av tidigare tecken. Som visades i 3.6.1 så är den verkliga entropin lägre än 5,25 bitar. För att ta hänsyn till beroendet mellan de olika tecknen kan algoritmen använda två eller fler tecken som symbol vid kodningen istället för att låta varje tecken vara en egen symbol. [16] Detta skulle innebära att kodningen tar hänsyn till beroendet mellan två (eller fler) på varandra följande tecken. Kodtabellen växer dock exponentiellt mot antalet tecken som används som symbol vilket gör sådana lösningar opraktiska. [16] Vidare så är Huffman en variabellängdskomprimering vilket innebär att varje symbol i alfabetet som ska kodas får en längd proportionerligt mot sannolikheten för symbolen. Det innebär att det är svårt att veta var i en bitström som ett nytt tecken börjar såvida strömmen inte avläses från början. Då det är önskvärt med 'random access' i den komprimerade strömmen/filen så är det också önskvärt att varje symbol kodas med

ett fixt antal bytes. Detta gör Huffman till en olämplig kandidat då de flesta symboler kommer att kodas med en bitlängd ej jämnt delbar med åtta.

3.6.4 Byte Pair Encoding BPE

Byte Pair Encoding (BPE) är en enkel komprimeringsalgoritm som fungerar bra för mindre textfiler eller textblock [3]. Metoden innebär att undersöka ett meddelande och hitta bytes som inte används, det vill säga, tal mellan 0-255 vilka inte finns representerade i meddelandet. I ett vanligt textmeddelande används få av dessa tal eftersom våra vanliga bokstäver, interpunktionstecken och siffror tar upp en mindre del av den möjliga bytemängden. Därefter tas de vanligaste byteparen (två direkt på varandra följande bytes) och kodas om till en av de oanvända byten. På så sätt skapas ett meddelande vilket är kortare än det ursprungliga meddelandet. Viss plats behövs även (i likhet med huffman) för kodtabellen. Vid komprimering av uppsalakatalogen fås en genomsnittlig bitlängd per tecken på 5,55, vilket ligger i närheten av Huffman och entropin (utan hänsyn tagen till beroenden.) Undersökningar visar att Byte Pair Encoding är en mycket bra algoritm för databaser med få uppdateringar. [3] Det tar relativt sett lång tid att komprimera men fort att avkomprimera och kompressionsgraden är god. [3] Därmed är den en bra lösning för vår applikation.

3.6.5 Resultat

För att kunna läsa poster i databasen i godtycklig ordning måste någon typ av blockkod användas. Därmed försvinner många av dagens kraftfullaste komprimeringsalgoritmer så som LZSS och aritmetisk kodning. En undersökning av Huffmanalgoritmen visar att dess variabellängdsbeteende leder till svårigheter med att läsa godtycklig post i databasen. Kvar finns några enklare blockkodningar varav BPE lämpar sig bäst för vår applikation. En telefonkatalog i Uppsalakatalogens storleksordning tar dock inte upp plats som överstiger det av kravspecifikationen fastställda högstavärdet. Därav ligger en implementation av

BPE inte med i detta projekt.

3.7 Stöd för kartor

Ett av de krav som framgår ur kravspecifikationen (Bilaga B) är att kartor över de lokala regionerna ska finnas tillgängliga i applikationen. Syftet med den här analysen är att ta reda på hur mycket utrymme (räknat i byte) en karta över en stad kan tänkas ta, men också vilken upplösning vi kan få på kartan om vi bestämmer ett maximalt utrymme (t.ex. 100 Kb). Som utgångspunkt i analysen har vi valt arean över Uppsala.
Vi beskriver i den här analysen två metoder för att spara grafik; rastergrafik och vektorgrafik.

3.7.1 Rastergrafik

Rastergrafik, även kallat bitmap, är en metod för att spara digital grafik genom ett tvådimensionellt rutnät bestående av s.k. pixlar. En pixel representerar en punkt på bilden och innehåller information om färgen för just den punkten. Beroende på vilken sorts format man använder för rastergrafiken så tar de olika mycket minne. De vanligaste formaten är *Windows Bitmap* (bmp), *Graphics Interchange Format* (gif), *Joint Photographic Experts Group* (jpeg, jpg), *Portable Network Graphics* (png). De lämpligaste kandidaterna för vår applikation är GIF och PNG då de erbjuder förlustfri komprimering för bilder med 256 färger eller färre[10] och hanterar bilder med stora färgade områden bra. För den här analysen nöjde vi oss med att bara använda GIF då skillnaderna mellan dem är ganska små.
Nedan följer två analyser för att använda rastergrafik till kartor. Den första analysen leder till slutsatsen att det tar för mycket minne och att man måste skala ned grafiken för att göra det möjligt.

[10]De kartor vi analyserar har mindre än 256 färger så GIF samt PNG erbjuder då förlustfri kompression

Metod 1 Vi har använt ett urval på 10 st kartor över Göteborg. Kartorna är 256x256 pixlar stora och i genomsnitt 5702 byte stora. Varje karta representerar en yta på c:a 210x210 meter (44100 m^2). Detta ger oss en koefficient R:

$$R = \frac{5702}{44100} \approx 0,129297\,\text{byte}/\text{m}^2 \tag{3.19}$$

Det krävs alltså c:a 0,129 byte för att representera en m^2. Figur 3.8 illustrerar ett exempel på en av de kartorna som vi har använt. Uppsala har en area på 4771 hektar, vilket

Figur 3.8: Exempel på karta

motsvarar 47710000 m^2. Vi vet att varje m^2 tar c:a 0,129 byte att representera och kan då beräkna den totala datamängd d för att representera hela Uppsala:

$$d = 0,129297 \cdot 47710000 \approx 6168762\,\text{byte} = 6024\,\text{Kb} = 5,9\,\text{MB} \tag{3.20}$$

Resultat 1 För att representera en stad, i storleksordning med Uppsala (4771 hektar), så krävs det c:a 5,9 MB om man vill ha den upplösning som illustreras i figur 3.8.

Metod 2 Om man skulle välja att begränsa storleken av utrymmet som kartorna får ta till exempelvis 100 Kb (102400 byte) så kan vi beräkna vilken upplösning det kommer att

medföra, dvs. Hur koefficienten R (byte/m^2) påverkas:

$$R = \frac{102400}{47710000} \approx 0,0021463\,\text{byte/m}^2 \tag{3.21}$$

För en karta med samma dimensioner som i Figur 3.8 (210x210 meter = 441000 m^2) så blir storleken s:

$$s = 44100 \cdot 0,0021463 \approx 94,65\,\text{byte} \tag{3.22}$$

Detta gör att vi måste skala om kartan med en faktor f:

$$f = \frac{94,65}{5702} \approx 0,017 \tag{3.23}$$

Storleken kommer alltså att vara c:a 1,7% av originalet. Det medför att storleken på bilden i pixlar kommer att bli:

$$0,017 \cdot 256 \approx 4,3\,\text{pixlar} \tag{3.24}$$

Resultat 2 Figur 3.9 visar centrala Göteborg i normal skalning. Figur 3.10 visar hur samma karta skulle se ut om man applicerade skalningen från beräkningarna ovan.

3.7.2 Vektorgrafik

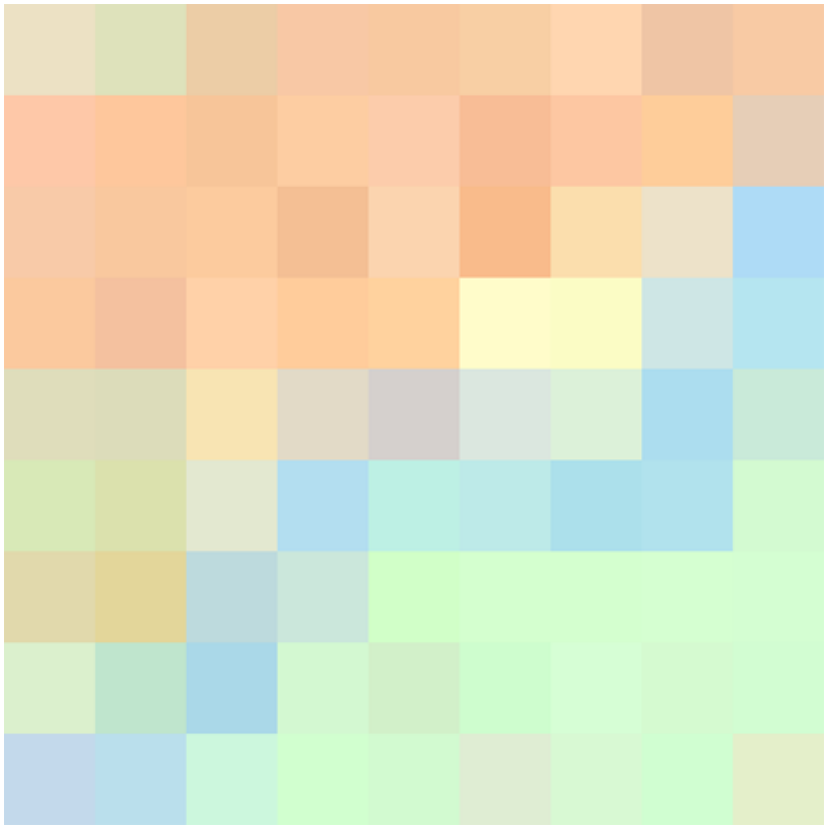
Istället för att spara informationen om grafiken som en array av pixlar, som i rastergrafik, så finns vektorgrafik. I vektorgrafiken sparar man information om vilka sorters former; cirklar, rektanglar, linjer m.m. samt färgerna på dessa för bilden. När bilden sedan ska ritas upp på skärmen så beräknas utseendet för komponenterna ut enlig matematiska formler. Detta gör att man kan zooma steglöst utan att förlora skärpa eller kvalitet.
Kartan i figur 3.11 är ett exempel på hur en vektorkarta kan se ut. Här finns det en byggnad (rektangel i figur 3.11) som är positionerad uppe till vänster i figuren. Nere i höger hörn ligger en liten damm (cirkel i figur 3.11) och mellan dessa slingrar sig en väg (linje i figur 3.11). Informationen för detta kan enkelt beskrivas genom att, i text, uttrycka vilken form

Figur 3.9: Centrala Göteborg utan skalning

figuren eller linjen har och färgen, t.ex. i xml-format. Ett format som är uppbyggt på precis detta sätt är *Scalable Vector Graphics* (svg). Kartan ovan tar endast 2.5 Kb i SVG-format och c:a 900b om man använder den komprimerade varianten SVGZ. Eftersom SVG är i XML-format så är det ganska mycket överflödig information och det går därför komprimera ytterligare.

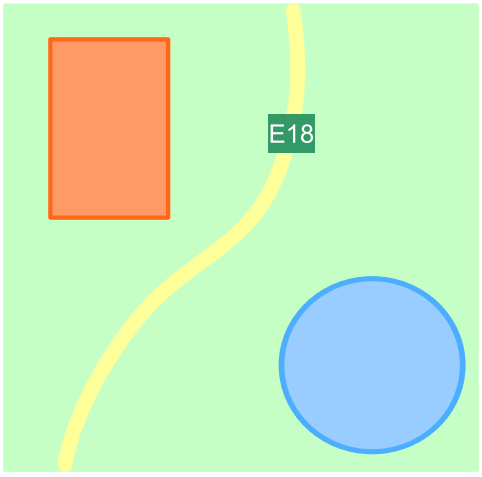
3.7.3 Slutsats

Resultaten från analysen ger en tydlig bild av att någon annan teknik för att representera kartor måste tillämpas för att göra det genomförbart inom ramarna för kravspecifikationen. En implementation med rastergrafik ger tydliga indikationer på att kravet om maximalt 5 Mb kommer misslyckas då beräkningen förutspår en storlek på c:a 5,9 MB. I de 5 MB



Figur 3.10: Centrala Göteborg, skalat

ska dessutom en databas för vita och gula sidorna få plats. Resultat 2 är en orealistisk implementation eftersom informationen ur kartan är obrukbar. En tänkbar lösning är att använda någon typ av vektorbaserad karta vilken alltså har en (teoretiskt) oändlig upplösning utan att ta upp allt för mycket plats. Att rendera en sådan bild kräver dock mer av enheten och rutiner för renderingen måste implementeras. Då stöd för kartor är ett sekundärt krav avsätts inte mer tid för en lösning m.h.a. vektorbaserade bildformat.



Figur 3.11: Exempel på karta i vektorformat.

4 Design

Detta kapitel kommer beskriva designen av applikationen. En mer detaljerad beskrivning av viktigare algoritmer presenteras i kapitel 5.
Applikationen består av tre delar, eller paket med Java-terminologi (eng. packages). Dessa tre är model, view och controller. Paketen innehåller klasser för att kontrollera respektive del av applikationen enligt Model-View-Controller (MVC) [4, 10, 23]. Dessa tre delar beskrivs nedan under respektive avsnitt. Kapitlet börjar dock med en beskrivning av de designmönster som finns implementerade i systemet. Här ses MVC som ett designmönster trots att det enligt vissa, exempelvis [4], snarare bör benämnas som ett arkitekturiellt mönster.

Terminologi och typsnitt För att skilja på vad som avses med ett användargränssnitt och ett gränssnitt mellan klasser (jmfr. eng. interface) kommer följande terminologi användas:

- Användargränssnitt: gränssnitt, användargränssnitt eller grafiskt gränssnitt
- Gränssnitt mellan klasser: interface

Angående typsnitt så skrivs klassnamn och metoder alltid med kursiv text, exempelvis *TextInput*.

4.1 Användargränssnitt

Applikationen består av fyra logiska vyer/gränssnitt där användaren presenteras för data. Dessa fyra syns i figur 4.1. Figur 4.1(a) visar den första vy som presenteras för användaren. Här sker inmatning av söksträng och sökning. Längst ned till höger, kopplad till en så kallad "softkey" (se figur 4.2), finns en meny bestående av följande val:

- "Sök" - Används för att påbörja en sökning.
- "Hjälp" - Används för att visa hjälpinformation.

Figur 4.1: De fyra logiska vyerna över applikationen.

- "Om" - Används för att visa information om upphovsrätt och programversion.
- "Rensa" - Används för att rensa sökformuläret och återställa programmet till dess startpunkt.
- "Avsluta" - Avslutar applikationen.

Menyn öppnas då användaren trycker ned den "softkey" som är kopplad till menyn. Allmänt gäller att för att göra val i programmet så trycks en "softkey" ned. Då användaren väljer att söka presenteras vyn som syns i figur 4.1(b). Här kan de två fälten "Privat" och "Företag" markeras och aktiveras (via "softkey") vilket visar de träffar som finns inom vita respektive gula sidorna. Den vyn som presenterar detta syns i figur 4.1(c). Här kan användaren bläddra mellan de olika sökträffarna. Genom att vandra upp eller ner markeras

Figur 4.2: En Sony-Ericsson W800i med sina två “softkeys” inringade.

en av tio poster. Genom att vandra vänster eller höger väljs tio nya poster att presenteras. Posterna ligger i bokstavsordning för vita sidorna och i en prioritetsordning vad gäller gula sidorna. (Prioriteten är tänkt att bero på hur mycket pengar företaget är berett att betala för annonsen. Den som betalar mycket hamnar med andra ord högt upp i listan.) Varje post kan aktiveras vilket tar användaren vidare till nästa logiska vy, (d) i figur 4.1. Här visas detaljerad information om personen/företaget och en provisorisk, statisk karta presenteras. Det är i denna vy som användaren kan lägga till kontakten i sin egen telefonbok eller ringa upp kontakten.

I de två logiska vyerna (c) och (d) finns möjligheten att gå bakåt, det vill säga, visa den vy vilken föregår den aktuella. I (a) och (b) kan programmet avslutas och i (b) kan programmet återställas (genom menyvalet "Rensa") till vy (a). Denna sistnämna vy är främst till för att annonsörerna ska få visa upp sina annonser. Som exempel syns i figuren en logotyp för DHL. Detta är en rullande annons som visar upp flera olika annonsörers varumärke med fem sekunders mellanrum. Logotypen kan markeras och aktiveras vilket tar användaren till den detaljerade informationen om företaget, det vill säga till vy (d). Detta annonssystem liknar med andra ord mycket det som finns på webben.

4.2 Använda designmönster

Designmönster Ett designmönster namnger, abstraherar och identifierar huvudaspekterna av en gemensam designstruktur. Detta gör mönstret användbart för att skapa återanvändbar objektorienterad design [9]. Ett designmönster inom mjukvaruutveckling kan liknas vid de designmönster vilka används av byggnadsarkitekter. De beskriver kärnan i en lösning på ett sätt så att liknande problem kan lösas om och om igen utan att lösningen behöver upptäckas på nytt varje gång. Mönstren gör det också möjligt att katalogisera och gruppera olika lösningar. Typiska egenskaper hos designmönster är:

- Namn. Ett namn, vilket helst ska beskriva både vilket problem mönstret löser men också hur lösningen ser ut. Genom att namnge mönstret blir det både enklare att förmedla våra erfarenheter och tala om problemen och dess lösningar.

- Problemet. Beskriver när mönstret ska användas. Vissa mönster (antimönster) beskriver när en viss design är olämplig, det vill säga vilka problem som inte ska lösas med en viss design.

- Lösningen. Beskriver vilka delar designen består av, deras förhållanden och ansvarsområden. Lösningen beskriver aldrig en konkret design eller implementation annat än för att exemplifiera. Designmönster är abstrakta beskrivningar av återkommande

problem och dess lösningar, inte lösningen på ett konkret problem. Lösningen beskriver med andra ord det allmänna problemet och hur en generell uppsättning av element kan lösa problemet.

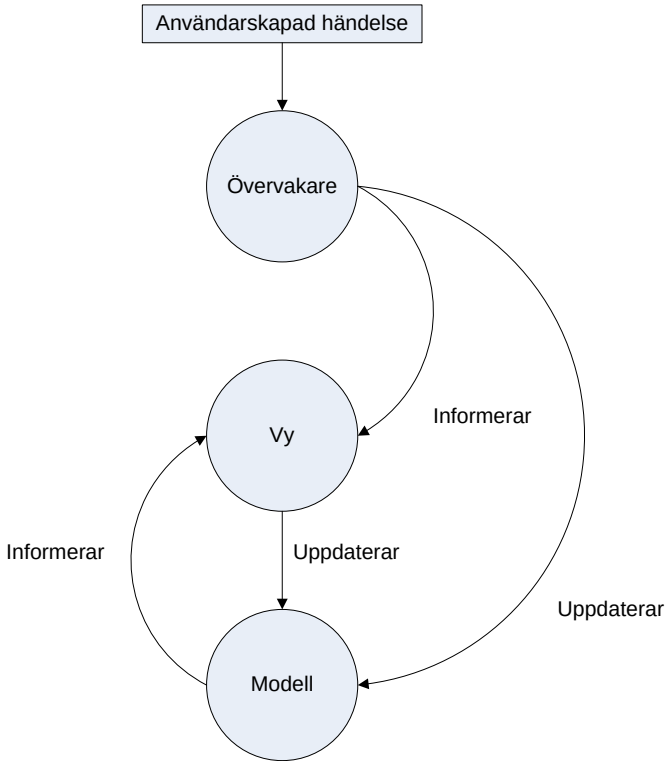
- Konsekvenser. Vilka resultat kan förväntas av mönstret och vilka kompromisser innefattas i mönstret. Konsekvensen av att använda ett visst mönster kan exempelvis vara att kopplingen mellan två objekt blir lägre men att systemet blir långsammare på grund av en mer omfattande arvshierarki.

Vad som är och inte är ett mönster kan sägas ligga i betraktarens ögon. En persons mönster kan vara en annans relativt primitiva byggblock [9].

4.2.1 Model-View-Controller

Syftet med Model-View-Controller (MVC) är att separera användargränssnittet från systemlogiken. Genom en sådan separering är det möjligt att återanvända logiken i ett system där ett annat gränssnitt är önskvärt [23]. Denna separation åstadkoms genom att dela in systemet i tre delar, en modell en vy och en övervakare (eng. controller). Modellen representerar den del av systemet som har hand om logiken, modellering samt modifiering av data. Vyn representerar användargränssnittet och dess grafiska komponenter, därmed är vyn ansvarig för att presentera data. Övervakarens uppgift är att ta emot inmatningar från användaren och sedan underrätta antingen modellen, vyn eller båda om inmatningen för att dessa ska uppdateras i enlighet med inmatningen. En illustration över detta finns i figur 4.3 [10].

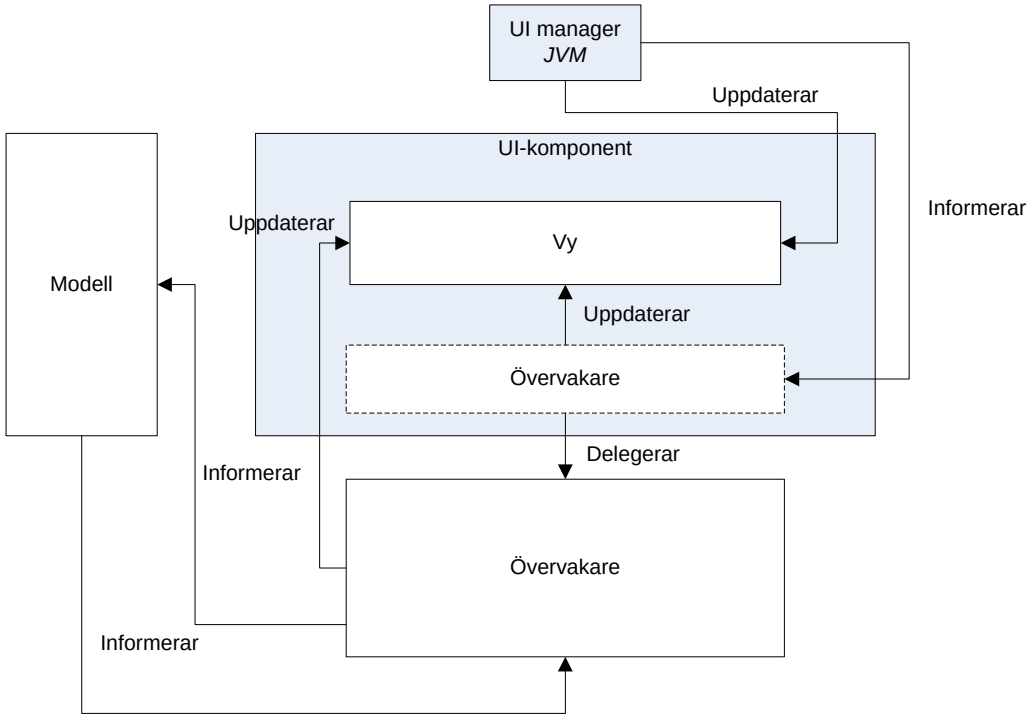
I en ideal objektorienterad implementation av detta skulle var och en av de tre komponenterna hamna i en egen klass, som egna typer. I praktiken visar det sig att denna implementation är svår att konstruera på grund av den starka kopplingen mellan vyn och övervakaren. Eftersom användaren interagerar men den fysiska representationen (verkligheten) av en vy-komponent så blir övervakaren beroende av implementationen av vyn.



Figur 4.3: Hur Model-View-Controller kopplas till varandra.

Av denna anledning ses ofta vyn och övervakaren i en och samma klass. Ett exempel är Java Swing där Sun lagt övervakaren och vyn i samma komponent. Sun kallar detta för "Separable Model architecture" vilken alltså endast gör en tydlig skillnad mellan modellen och vyn. [10] Även J2ME har denna typ av komponentuppbyggnad vilket tvingat designen av vår applikation i samma riktning. Den modell av MVC som används i applikationen kan ses i figur 4.4.

Figur 4.4 visar hur delar av övervakaren ligger direkt i de grafiska komponenterna, styrda av Java-maskinen, medan de delar vi själva har kontroll över lagts i en separat övervakare. Då en användare matar in något kommando (exempelvis trycker på en knapp) så skickar UI-manager en signal om detta till övervakningsdelen av en UI-komponent. Denna uppdaterar vyn, det vill säga, skriver ut ett tecken på skärmen eller visar ny data. Dessutom delegeras uppgiften i tillämpliga fall (om knapptrycket motsvarade att en sökning



Figur 4.4: Model-View-Controller och dess implementation i applikationen.

ska påbörjas t.ex.) vidare till den utomstående övervakaren vilken informerar modellen. Modellen utför sökningen och informerar övervakaren när sökningen är klar. Övervakaren uppdaterar sedan vyn vilken presenterar ny data.

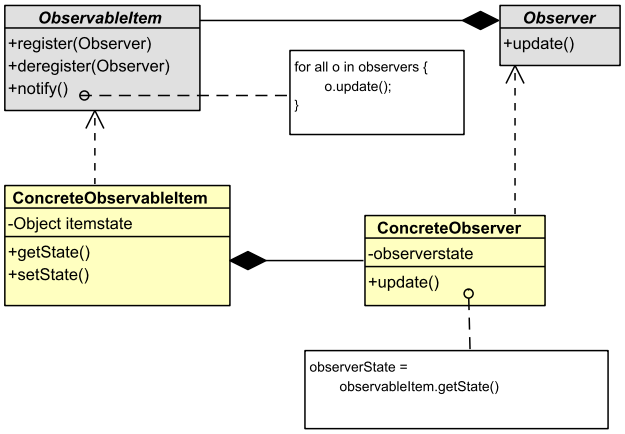
Då det finns en rad olika grafiska komponenter finns också en rad olika övervakare i applikationen. Uppdelningen mellan övervakare, vy och modell ligger därför snarast i paketindelning, det vill säga, de olika delarna ligger i olika Java-paket.

4.2.2 Consumer/Producer

Ofta behöver två processer eller trådar kommunicera enligt ett "producent/konsument"-schema; producenten tillverkar data vilket används av konsumenten. Som exempel kan nämnas en producent vilken tillverkar bokstäver för att skrivas ut på skärmen, och en konsument som läser rader av bokstäver. Detta kan naturligtvis modelleras procedurieIlt. När producenten tillverkar ett radbrott informeras konsumenten vilken skriver ut data på

skärmen. Problemet med en sådan lösning är att producenten står inaktiv under utskriften och därför inte kan skapa mer data.
För att ge sken av att de båda processerna exekverar samtidigt kan en buffert användas. Producenten skriver data till bufferten och konsumenten läser bufferten. Då bufferten är tom står konsumenten inaktiv och väntar på att ny data ska anlända. Denna lösning förutsätter att bufferten har en obegränsad storlek vilket är att förutsätta mycket, speciellt för mobila enheter med mycket begränsade resurser. [8] En annan lösning använder ett lås enligt "Guarded suspension"-mönstret. Detta innebär att producenten låser bufferten då skrivning pågår medan konsumenten låser bufferten då läsning pågår. Buffertstorleken är då exempelvis ett tecken, eller ett heltal, stort. [5] Detta innebär att producenten inte kan producera nya tecken i den takt den behagar. Lyckligtvis ger inte detta några problem i de fall där det används i vår applikation. Mer om hur detta mönster används finns i avsnitt 5.1.

4.2.3 Observer



Figur 4.5: Designmönstret Observer.

Tanken med observer är att skapa ett beroende mellan klasser, sådant att när en klass ändrar tillstånd så informeras alla klasser som är beroende och uppdateras i enlighet med

detta. Mönstret har många användningsområden och ett tydligt exempel på ett sådant är separationen av data och användargränssnitt i ett system. Genom att separera klasser vilka representerar data och klasser för användargränssnittet så kan de återanvändas oberoende av varandra. Genom att separera dem via observer kan de också samarbeta väl ihop. Som exempel; ett cirkeldiagram och ett stapeldiagram kan användas för att presentera samma information. De båda känner inte till varandra och är helt oberoende. Däremot beter de sig som om de kände till varandra - när informationen i cirkeldiagrammet ändras av en användare, ändras informationen i stapeldiagrammet i enlighet med förändringen. Detta innebär att de båda diagrammen (eller snarare de klasser vilka representerar diagrammen) är beroende av något dataobjekt. Då detta objekt förändras ska de båda diagramklasserna bli uppdaterade om händelsen. Detta är precis vad observermönstret åstadkommer. [9]
Klassdiagram i figur 4.5 visar mönstrets ingående klasser och beroenden mellan de olika klasserna. De viktigaste klasserna är *ObservableItem* och *Observer*, båda abstrakta. (*Observer* kan vara ett interface i Java.)

- *ObservableItem*

  Har en eller flera referenser till *Observer*s. För att registrera sig som lyssnare på ett *ObservableItem* så används *register(Observer)*.

- *Observer*

  Ett interface som tillhandahåller en uppsättning metoder vilka alla reella *Observer*s ska implementera. Detta för att *ObservableItem* ska veta vilka metoder som finns tillgängliga hos sina lyssnare. I de flesta fall räcker det med metoden *update()*.

- *ConcreteObservableItem*

  Ett konkret objekt hos vilket *Observer*s kan registrera sig. Håller information relevant för objektets (klassens) uppgift. Denna information kan sedan hämtas via metoden *getState()*, eller uppdateras via metoden *setState()*. Då något tillstånd hos ett *ConcreteObservableItem* ändrats anropas *ObservableItem*s (basklassens) *notify()* vilken

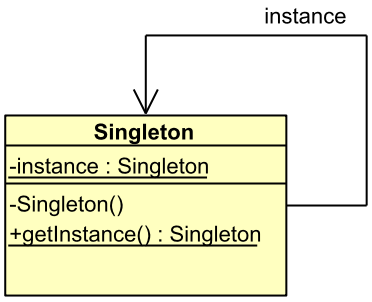
informerar samtliga registrerade *Observer*s om förändringen.

- *ConcreteObserver*

  Har en referens till ett *ConcreteObservableItem*. Implementerar *update()* hos *Observer*.

Detta mönster används, något modifierat, av vår applikation för att låta det grafiska gränssnittet underrätta controllern om att tillståndet i gränssnittet har förändrats. Klasser som ingår i detta mönster kan ses i figur 4.10. De modifikationer som gjorts är först och främst att vår *Observer* inte håller en referens till objektet den lyssnar på. Istället underrättas *Observer*n om vilken typ av förändring som skett genom att *update* tar emot dels ett *Event* men också en referens till objektet som anropade *update*. På detta sätt uppnås större separation mellan objekten då *Observer* inte behöver känna till vilken klass den observerar, bara vilken typ av *Event* den lyssnar på. Dessutom leder modifikationen till att en *Observer* kan lyssna på godtyckligt många objekt.

4.2.4 Singleton



Figur 4.6: Designmönstret Singleton.

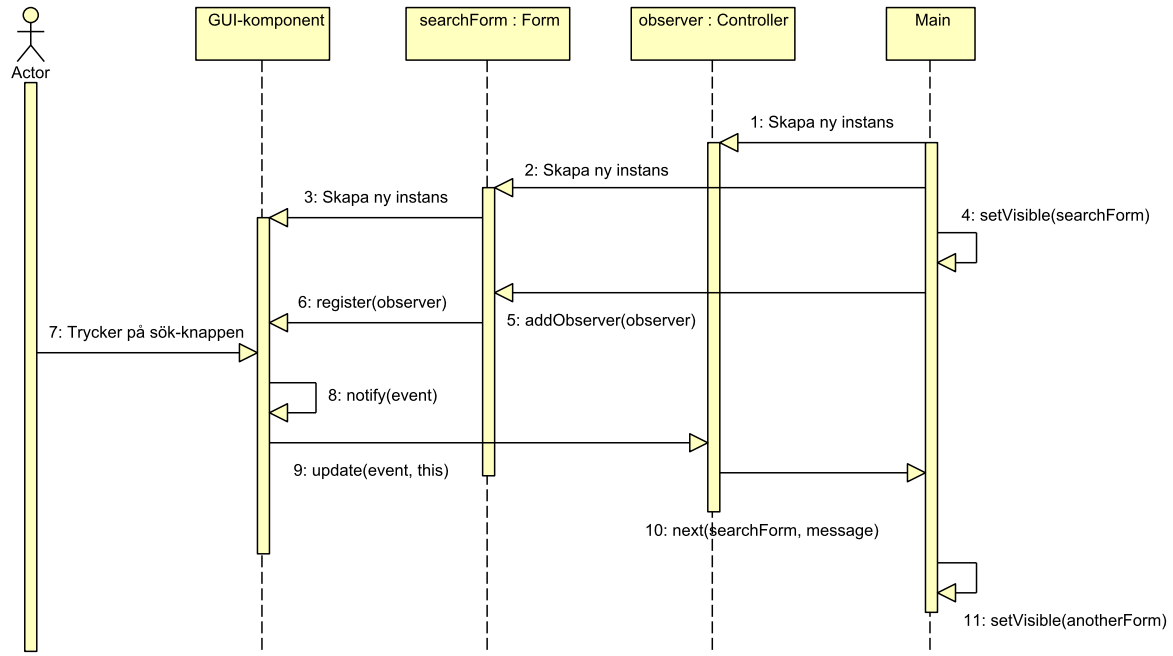
Singleton är ett designmönster som ligger under, vad man brukar kalla, *creational patterns* d.v.s skapande designmönster. Det innebär att mönstret har som uppgift att skapa objekt på ett eller annat sätt. Just singleton är till för att försäkra sig om att endast en instans av en klass har skapats. Designmönstret är konstruerat så att klassen, som det

endast ska få finnas en instans av, erbjuder en global accesspunkt till den enda instansen av sig själv. Detta sker via en statisk metod som kallas *getInstance()* som då returnerar denna instans. Figur 4.6 visar ett UML-diagram över designmönstret singleton [9].
I våran applikation så använder vi singleton-mönstret för att försäkra oss om att endast en instans av klassen för inläsning av information (kallad *DBManager*) från databasen existerar. Anledningen till att vi endast vill ha en instans av *DBManager* är för att kunna kontrollera hur information läses in från filerna. Hade det varit möjligt för flera instanser att existera så hade samma kontroll varit möjlig.

4.3 View

Paketets uppgift är att presentera information för användaren samt att ta emot kommandon från denne. Mycket av detta styrs av Javas virtuella maskin via Polish (se avsnitt 3.5) medan annat måste tillhandahållas av utvecklaren. I likhet med applikationer i J2SE så startar programmet alltid via ett anrop till funktionen *main()* vilken i en J2ME-applikation ligger i en subklass till MIDlet. Denna klass syns i figur 4.9 som *Main*. Klassens huvudsakliga uppgift är att starta upp programmet, koppla samman view och controller samt att kontrollera programflödet. Gränssnittet består av tre stycken Forms (jmfr. Swings JForm), *SearchForm*, *HitsForm* och *OneHitForm* vilka ärver från Form i Polish. Kontrollen av programflödet innebär att *Main*, beroende på användarens val, skiftar mellan dessa tre Forms. (Se figur 4.7.) Då det är dags att byta Form underrättas en controller vilken i sin tur skickar förfrågan till *Main.next(Form, Object)* eller *Main.previous(Form, Object)* beroende på vad för sorts händelse som har inträffat. (Mer om programflödet finnas i avsnitt 4.4).

SearchForm har som uppdrag att tillhandahålla ett gränssnitt för inmatning av en söksträng. Det är också via *SearchForm* som användaren talar om för applikationen att en sökning ska påbörjas. Då en sökning är genomförd presenteras antalet sökträffar i de vita respektive gula sidorna i *SearchForm* och användaren kan sedan välja att klicka på



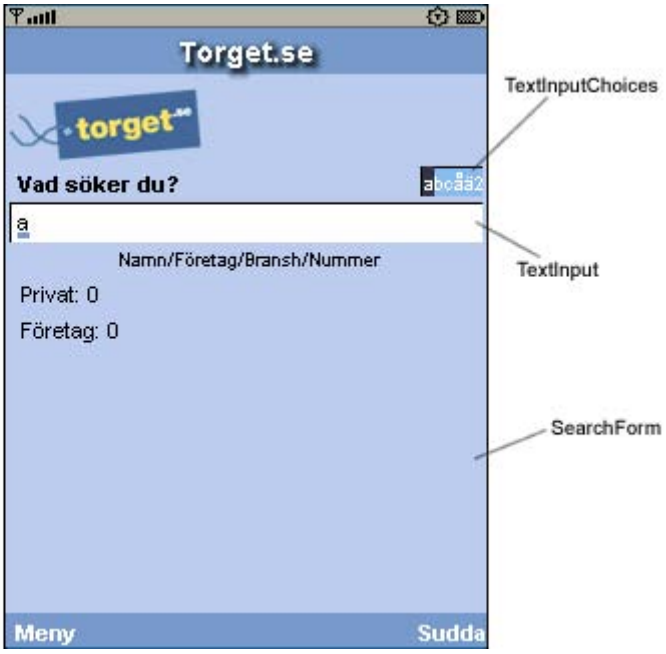
Figur 4.7: Programflödet styrt av Main.

antingen de vita sidorna eller de gula. Då så sker tas användaren (via *Main*) till *HitsForm*, där samtliga poster vilka matchar söksträngen visas med namn och adress. Ett klick på en av dessa poster tar användaren till en *OneHitForm* vilken visar all tillgänglig information om den angivna posten. För inmatning av söksträngen i *SearchForm* används *TextInput* vilken lyssnar på tangenttryckningar från användaren och översätter dessa till bokstäver enligt enhetens vanliga textinmatning - då T9 inte är aktivt. Som hjälp har *TextInput* en klass, *TextInputChoices*, vilken visar de tecken som är möjliga att skriva för en given knapp samt vilket tecken som kommer att skrivas om användaren trycker på en annan knapp alternativt väntar i en sekund. De olika komponenterna/klasserna kan ses uppritade i figur 4.8.

För att rita upp *TextInput* och *TextInputChoices* används metoden *paint(int, int, Grapics)*. Denna kallas på av systemet då komponenterna behöver målas ut i Formen. *TextInputChoices* styrs av *TextInput* och dessa är därför mycket tätt bundna (eng. coupled) till varandra. För att kunna styra även ritandet av *TextInputChoices* så använder *TextInput*

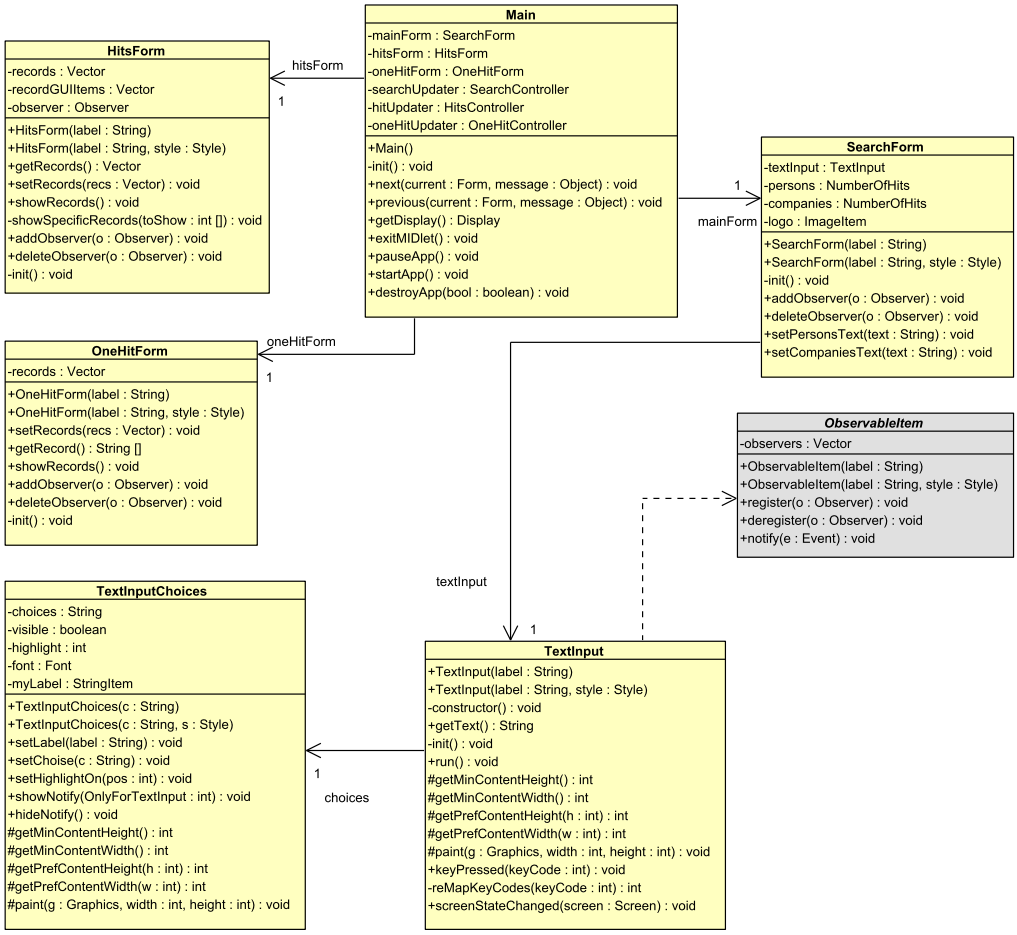
metoden *TextInputChoices.showNotify(int)* där *int*:en är en dummy-variabel endast till för att skilja metoden från basklassens *showNotify()*. Denna design gör med andra ord att det är svårt att använda *TextInputChoices* utan *TextInput* och vice versa. Å andra sidan har de olika klasserna skilda uppgifter vilket, i viss mån, rättfärdigar att de ligger i olika klasser.

TextInput har två metoder för att hantera inmatning från användaren - *keyPressed(int)*



Figur 4.8: Applikationen när en SearchForm är aktiv.

och *keyReleased(int)*. Den förra anropas då användare trycker på en knapp och den senare då knappen åter släpps upp. Anropen sker från systemet - via Polish. *TextInput* ärver från *ObservableItem* vilket är en del av gränssnittet för att hantera händelser (eng. events). *ObservableItem* är en del av designmönstret Observer [9]. (Se 4.4) Mönstret tillåter andra klasser (i paketet controller) att lyssna på ändringar i *TextInput* så att modellen kan underrättas om när det är dags att söka.



Figur 4.9: UML-klassdiagram över View.

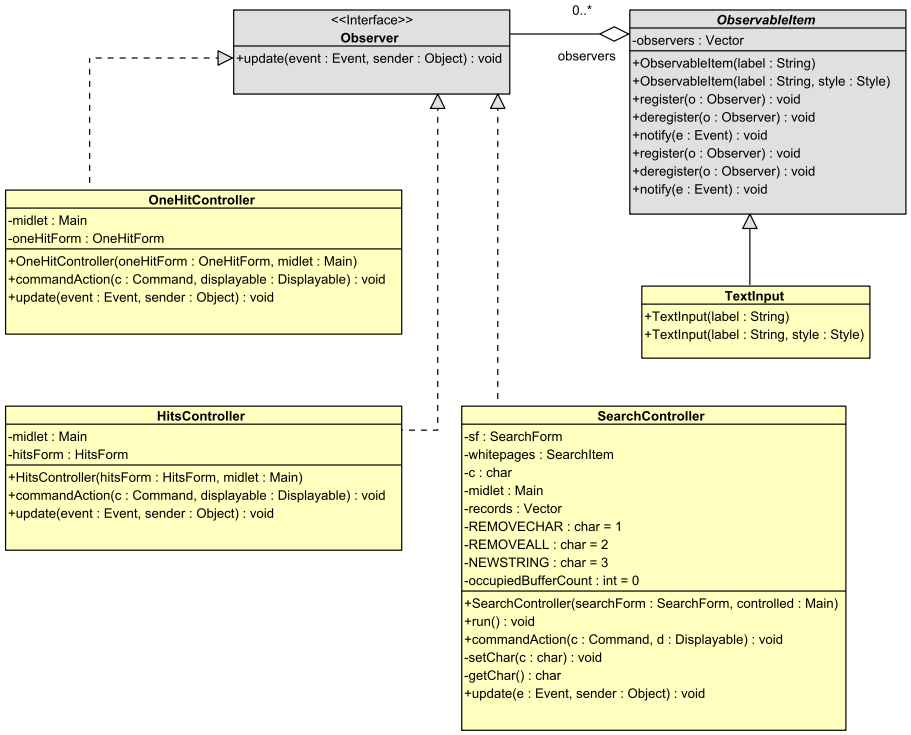
4.4 Controller

Paketets uppgift är att agera mellanhand mellan view och model så att dessa båda delar hålls separerade. För att åstadkomma denna separation innehåller paketet klasser vilka realiserar designmönstret Observer. (Mönstret är något modifierat mot vad som finns beskrivet i exempelvis [9].) De tre klasserna *SearchController*, *HitsController* och *OneHitController* (se figur 4.10) implementerar interfacet *Observer*. Detta innebär att de lyssnar på händelser som sker i användargränssnittet genom att nödvändiga komponenter i användargränssnittet implementerar den abstrakta klassen *ObservableItem*. *ObservableItem* har en lista innehållandes *Observers*. För att lyssna på ett *ObservableItem* måste den

som lyssnar först implementera *Observer*. Därefter anropas *register(Observer)* för den komponent som *Observer*:n vill lyssna på, vilket lägger till *Observer*:n i listan. När en *ObservableItem* utfört en viss åtgärd anropas *notify(Event)* vilken i sin tur anropar *update(Event, Object)* för varje *Observer* i listan. På så sätt underrättas de tre Observer-klasserna om att något har förändrats i det grafiska gränssnittet. I figur 4.10 syns att *TextInput* är en *ObservableItem*. Figuren visar dock av praktiska skäl inte alla gränssnittsklasser som implementerar *ObservableItem*. *TextInput* är alltså endast med i figuren som exempel på en sådan klass.

Vissa händelser styrs av systemet och kan därför inte lyda under Observer-mönstret. Istället hålls applikationen underrättad via så kallade *Command*:s. *Command*:s styr den meny som syns längst ned i bild i figur 4.8. Varje val som kan göras i en sådan meny är i botten ett *Command*, vilka läggs till för grafiska komponenter. För varje grafisk komponent innehållandes något *Command* anropas *setCommandListener(CommandListener)* för att registrera *CommandListener*:n. Detta innebär att samtliga tre *Observer*-klasserna också är *CommandListener*:s och får därmed alltså reda på om ett visst val har gjorts från menyn. (Detta syns inte i figur 4.8.)

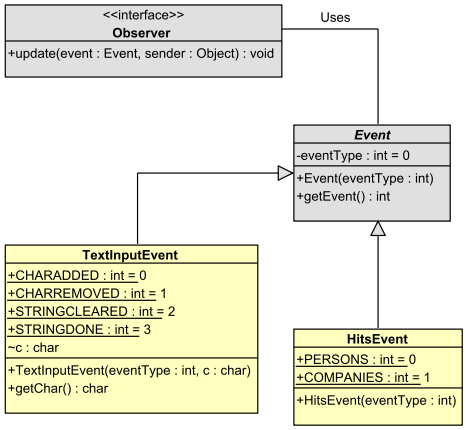
För att tala om vilken händelse som har inträffat, och var, tar *update(Event, Object)* emot ett objekt av typen *Event*. Detta är en abstrakt klass och måste således implementeras av subklasser för att kunna instansieras. Dessa subklasser är *TextInputEvent* och *HitsEvent* (se figur 4.11). *Event* har en medlem *eventType* vilken talar om vilken typ av händelse som har inträffat. Det är upp till subklasserna att ge mening till *eventType* vilket görs genom publika statiska konstanta medlemmar. När en händelse inträffar i ett gränssnittsobjekt anropas alltså *notify(Event)* med ett nytt *Event*, exempelvis *HitsEvent*. *eventType* sätts till det värde vilket motsvarar händelsen och *notify(Event)* anropar sedan *update(Event, Object)* med detta *Event* vilket informerar lyssnarna om vilken typ av *Event* som inträffat (ex. *HitsEvent*), vad denna typ har för värde (*eventType*) samt vilken komponent som genererat händelsen (via parametern *Object*).



Figur 4.10: UML-klassdiagram över Controller.

SearchController lyssnar på *TextInputEvent*:s vilka genereras av *TextInput*. Denna information används för att tala om för modellen att det har kommit nya uppgifter om hur söksträngen kommer att se ut. På detta sätt kan modellen söka under tiden som användaren matar in söksträngen vilket snabbar upp sökningen avsevärt. *SearchController* lyssnar också på *Command*:s som talar om att användaren avslutat inmatningen, att användaren vill avsluta programmet samt att användaren vill hämta de poster som matchar söksträngen. För de mobiltelefoner vilka inte har någon naturlig knapp för att sudda ett tecken lyssnar *SearchController* också på ett *Command* för detta.

HitsController lyssnar på *HitsEvent*:s vilka genereras av *Record*:s (syns ej i figur). Detta talar om för HitsController att användaren valt en viss post i sökträfflistan och nu vill se full information om den posten. *HitsController* lyssnar dessutom på *Command*:s för att



Figur 4.11: UML-klassdiagram över Event.

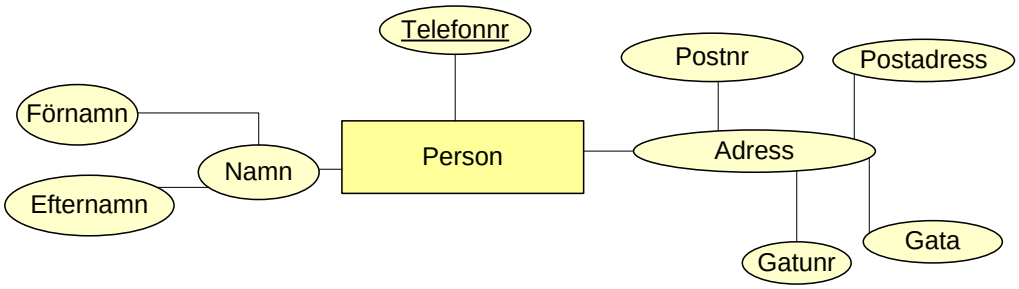
avsluta programmet samt att gå tillbaka till en *SearchForm* för att påbörja en ny sökning. *OneHitController* lyssnar inte på några *Event* utan endast på *Command*:s som talar om att användaren vill avsluta programmet eller gå bakåt till sökträfflistan.

4.5 Model

Paketet *model* innehåller logiken för systemet, vilket brukar omfatta det som handlar om att genomföra beräkningar, upprätta anslutningar, läsa och skriva data, etc. I vår modell återfinns de komponenter som realiserar t.ex. databasstrukturen och dess logiska komponenter. Det här innefattar bland annat att läsa från filer, söka i träd och organisera data. Delarna i paketet kommunicerar indirekt med *view* (Se avsnitt 4.3) via *controller* (Se avsnitt 4.4) enligt MVC-konceptet [4].
Nedan följer en beskrivning av designen över de viktigaste delarna i *model*-paketet, med tonvikt på databasen.

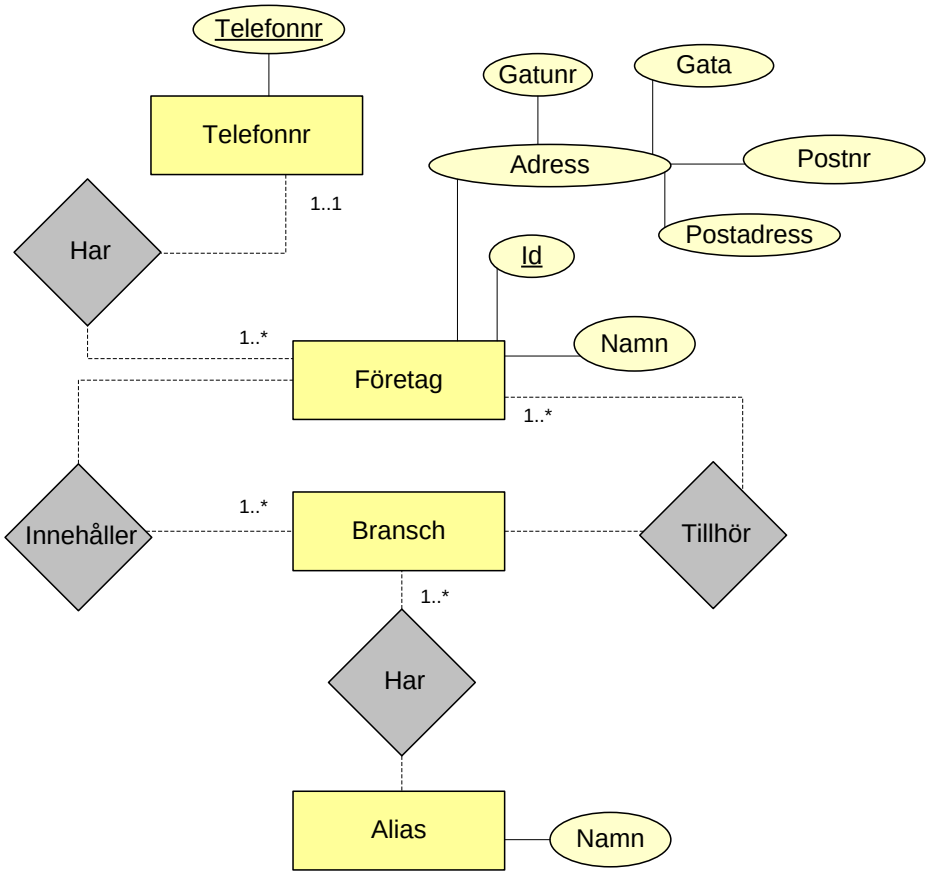
4.5.1 Entitetsrelationer

Detta avsnitt beskriver mycket kortfattat databasdesignen med hjälp av så kallade ER-diagram (där ER står för "Entity Relation") [6]. Tanken med avsnittet är att ge övergripande bild av hur informationen ligger organiserad, inte nödvändigtvis precis hur strukturen ser ut. Figurerna ska ses som konceptuella bilder och inte som exakta avbilder av systemet.
I figur 4.12 syns designen av de vita sidorna i sin enkelhet. Detta diagram består av en entitet, **Person**, och dess egenskaper/attribut. Huvudnyckeln för personer är personens telefonnummer, vilket alltså antas vara unikt för individen. Detta antagande stämmer i de flesta fall men det finns de som har exempelvis medabonnenter och alltså delar på samma telefonnummer. Detta är ett problem som i framtiden kan lösas med antingen ett extra attribut, medabonnent, eller en extra entitet. **Person** har i nuläget som synes inte några relationer till andra entiteter. Annorlunda då med **Företag**, vars relationsdiagram syns



Figur 4.12: ER-diagram över Vita Sidorna.

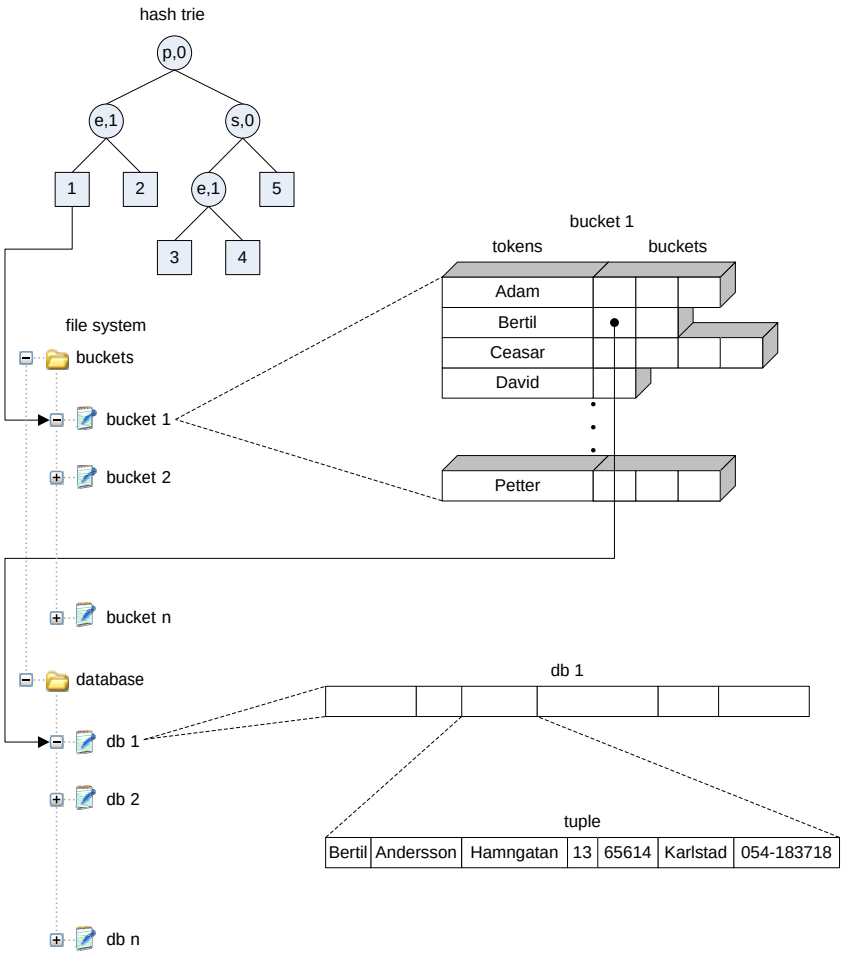
i figur 4.13. Här syns att **Företag** har relationer med ytterligare två entiteter, **Telefonnummer** samt **Bransch**. Anledningen till att dessa båda inte modelleras som attribut till **Företag** är för att ett företag kan ha fler än ett telefonnummer och kan dessutom tillhöra mer än en bransch. **Bransch** har en relation till entiteten **Alias**. Detta för att modellera att en bransch kan benämnas på en rad olika sätt. Som exempel kan nämnas att branschen "blommor" också kan benämnas med "florist" eller "blombindning".



Figur 4.13: ER-diagram över Gula Sidorna.

4.5.2 Databas

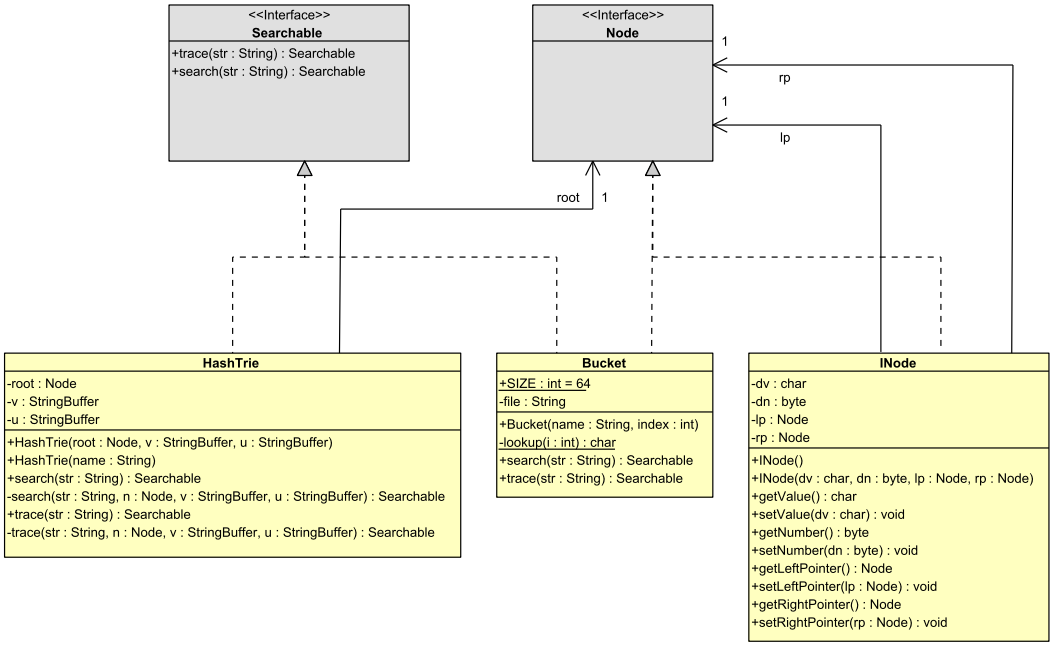
I enlighet med resultaten från avsnitt 3.4 så använder vi *trie hashing* och *binära sökträd* för att indexera data. Detta gör att vi på ett smidigt sätt kan koppla ihop det till ett system som använder många filer i sökningen, vilket var slutsatsen från avsnitt 3.3. En konceptuell bild över hur databasen, i sin enkelhet, fungerar illustreras i figur 4.14. Här ser vi hur lövnoderna i trädet, som i exemplet är ett *trie* men som kan vara ett *binärt sökträd* om sökningen sker över telefonnummer, pekar på filer som ligger i JAR-paketet. Dessa filer, som kallas *buckets* (hinkar), innehåller *tokens* tillsammans med index för de poster i databasen som innehåller detta token. Dessa två komponenter utgör tillsammans själva indexeringen för ett specifik fält i databasen. Detta förfarande sker alltså på liknande sätt



Figur 4.14: Konceptuell vy över databas.

för de fält som ska indexeras för snabb sökåtkomst. Själva "databasen", där all information för posterna finns lagrade, är distribuerad över ett antal mindre filer för att kunna hämta ut informationen snabbare då endast sekvensiell sökning är tillåten (se resultat från avsnitt 3.3).

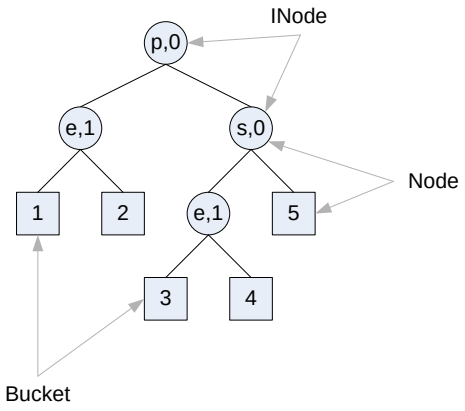
Databasen är i grunden uppbyggt av ett *trie* för att sköta indexeringen av data för strängar och av ett *binärt sökträd* för indexering av data för numeriska uttryck (telefonnummer). För att representera den abstrakta datastrukturen (ADT) trie hashing i Java har vi utvecklat ett antal klasser och interface. Deras relation till varandra illustreras i figur 4.15. Interfacet *Searchable* har en intressant funktion i hur sökningar i databasen sker. Meto-



Figur 4.15: UML-klassdiagram för ADT:n Hash Trie.

den *trace(str : String) : Searchable* utnyttjas för att kunna göra sökningar simultant som användaren knappar in söksträngar. Detta sker genom att traversera i trädet baserat på vad användaren hittills har matat in i sökfältet. Detta innebär alltså att det mesta av jobbet för att genomföra en sökning redan är avklarad innan användaren har hunnit trycka på sök-knappen.

Interfacet *Node* är ett "dummy-interface" som inte innehåller några metoder. Det är till för att på ett gemensamt sätt behandla inre noder och lövnoder. Exempelvis refererar *INode*, innernoder, till två barn som kan vara antingen inre noder eller lövnoder. Hur klasserna relaterar till de abstrakta modellerna av trie hashing från figur 3.4 i avsnitt 3.4 syns i figur 4.16, här syns även det som nämndes tidigare angående interfacet *Node* och hur det används av inre noder *INode*.



Figur 4.16: Design av Hash Trie.

5 Implementation

I detta kapitel kommer delar av implementationen att beskrivas. Detta är delar som varit mer problematiska än andra eller delar vilka är speciellt intressanta för applikationens funktionalitet.

5.1 Samtidig sökning och inmatning

I avsnitt 4.2 introducerades designmönstret consumer/producer eller konsument/producent på svenska. Detta mönster är en av hörnstenarna bakom applikationens sökfunktion. Då användaren skriver in sin söksträng är det önskvärt att söka under tiden så att det går fort då användaren väl trycker på sökknappen. Hastigheten förbättras eftersom en del av sökningen redan är utförd då användaren väl vill söka. Den verkliga vinsten ligger i att det är mycket dödtid för processorn då vi människor inte kan mata in text i samma hastighet som processorn kan ta hand om inmatningen. Denna tid kan användas för att söka. För att möjliggöra detta krävs två trådar, en som tar emot inmatningen från användaren och skriver ut ett tecken på skärmen samt en som tar emot tecknet och flyttar fram sökpositionen i sökträdet. (Det senare är möjligt då sökträdet är ett prefixträd.) Låt oss kalla tråden för inläsning för inmatningstråd och den andra för söktråd. Här kommer alltså inmatningstråden att producera ett antal tecken. Den måste också tillåtas producera dessa tecken *oavsett* hur många av tecknen som söktråden hinner konsumera. Detta innebär, förutom att söktråden och inmatningstråden inte kan vara samma tråd, att de båda måste samarbeta i ett konsument/producent-mönster. I sin tur innebär det att de delar på en buffert ur vilken konsumenten läser och producenten skriver. Bufferten är som beskrivits i avsnitt 4.2 endast ett tecken stort, dels för att spara utrymme men också för att mönstret inte kräver mer samt att detta var nog för att konsumenten ska hinna tömma bufferten. Detta innebär att användaren kan komma att uppleva förseningar i inmatningen eftersom sökningen kommer att låsa inmatningen om användaren hinner skriva fler än två

tecken.

Då applikationen startar, skapar *Main* de båda klasserna *SearchController* och *SearchForm* där den senare i sin tur skapar en *TextInput*. (Se figur 4.7.) *SearchController* och *TextInput* har de båda relevanta trådarna. Den förra, *SearchController*, agerar här konsument medan *TextInput* är producenten. Konsumenten startar i ett väntande, låst, läge och har med andra ord inte möjlighet att konsumera något förrän producenten säger till. Då användaren skriver in ett tecken tas detta emot av *TextInput* vilken informerar *SearchController* om detta (via observermönstret, se 4.2). *SearchController* reagerar genom att anropa den privata metoden *SearchController.setChar(char)* (se figur 5.1). Om inget teck-

```
1  //setChar(char c) är synkroniserad.
2  private synchronized void setChar(char c) {
3      while(occupiedBufferCount == 1) {
4          try {
5              wait();
6          } catch (InterruptedException exception) {
7              exception.printStackTrace();
8          }
9      }
10     this.c = c;
11
12     ++occupiedBufferCount;
13
14     notify();
15 }
```

Figur 5.1: Källkoden för setChar().

en tidigare tagits emot så är **occupiedBufferCount** (rad 3) lika med noll, eftersom inget skrivits till bufferten ännu. Därmed sätter *setChar(char)* den privata medlemmen **c** till det nyligen inmatade tecknet. Därefter ökas **occupiedBufferCount** med ett och den väntande tråden (konsumenten, i detta fall en speciell metod vilken kallas **run()** i Java) underrättas med ett anrop till *notify()*. Detta innebär att konsumenten försöker läsa ur bufferten vilket sker via ett anrop till *SearchController.getChar()* (se figur 5.2). Lägg märke till att detta anrop naturligtvis inte sker förrän söktråden - konsumenten - får tillgång till

```
1  private synchronized char getChar() {
2      while(occupiedBufferCount == 0) {
3          try {
4              wait();
5          } catch (InterruptedException exception) {
6              exception.printStackTrace();
7          }
8      }
9      --occupiedBufferCount;
10
11     notify();
12
13     return c;
14 }
```

Figur 5.2: Källkoden för getChar().

processorn, vilket inte nödvändigtvis är vid samma tillfälle som producenten kallar på *notify()*. *getChar()* kontrollerar om **occupiedBufferCount** är lika med noll. Eftersom så inte är fallet (*setChar()* ökade den med ett) hoppas **while**-satsen, **occupiedBufferCount** minskas med ett och den väntande tråden (producenten) blir notifierad via *notify()*.
Har ett tecken redan emottagits, det vill säga, **occupiedBufferCount** är lika med ett då *setChar(char)* anropas, så sätts producenten i vänteläge på rad 5, figur 5.1. Detta innebär att det redan finns ett tecken i bufferten och att *setChar(char)* har tillåtelse att köra. Så fort konsumenttråden får ta över kommer denna alltså att konsumera tecknet varpå producenten åter kan skriva till bufferten. På detta sätt hålls de båda trådarna synkroniserade. Då *getChar()* returnerar kommer den verkliga förtjänsten med denna lösning. Då har ett tecken konsumerats och kan nu användas för att stega fram i sökträdet. Samtidigt har låset på bufferten släppts vilket innebär att användare kan skriva in ytterligare ett tecken medan sökningen sker. Denna procedur exekveras i metoden *run()* i *SearchController* (se figur 5.3).

På rad 5 i figur 5.3 syns anropet till *getChar()*. De två variablerna *whitepages* och *yellowpages* är av typen *SearchItem* vilka är de objekt som utför framstegningen (alternativt bakåt) i sökträdet via anrop till bland annat *trace()*. Eftersom *run()* exekverar i en egen

```
1  public void run()
2  {
3      while(true)
4      {
5          char p = getChar();
6          switch(p) {
7              case REMOVECHAR:
8                  whitepages.backtrace();
9                  yellowpages.backtrace();
10                 break;
11             case REMOVEALL:
12                 whitepages.reset();
13                 yellowpages.reset();
14                 break;
15             case NEWSTRING:
16                 whitepages.newSearch();
17                 yellowpages.newSearch();
18                 break;
19             default:
20                 whitepages.trace(p);
21                 yellowpages.trace(p);
22         }
23     }
24 }
```

Figur 5.3: Källkoden för run().

tråd så utförs sökningen (konceptuellt) samtidigt som användaren kan mata in ett nytt tecken. På de modeller vi har testat har detta räckt - tiden det tar för en användare att mata in ett tecken är tillräckligt lång för att ett *SearchItem* ska hinna stega framåt respektive bakåt. Skulle det visa sig att någon modell exekverar för långsamt för att hinna stega i trädet så får en större buffert användas.

5.2 Inmatning utan J2ME

Ett problem närbesläktat med det beskrivet i avsnitt 5.1, så till vida att de båda handlar om samverkande trådar, är textinmatningen. Problemet bottnar i att en användaren måste kunna skriva hela alfabetet endast med hjälp av enhetens 12 knappar. Detta innebär att då en användare trycker två gånger på knapp två så ska ett “b” skrivas ut, om

knapptryckningarna kom tillräckligt snart efter varandra. (Tillräcklig tid är i applikationen 1000 ms.) Endast en knapptryckning ska generera ett “a”. Likaså ska två knapptryckningar, vilka inte ligger tillräckligt nära varandra i tid, på knapp två generera “aa”. Problemet är med andra ord att en sekvens av knapptryckningar som motsvarar exempelvis $\{2, 2, x, 2, 6, 2\}$ ska generera ordet “bana” och inte “aaana”. x står här för en sekunds uppehåll. Problemet kan tyckas trivialt vid en första anblick. Två trådar krävs där en tar tiden från varje knapptryckning medan den andra tar emot knapptryckningar. När en sekund gått från senaste knapptryckningen informerar tidtagartråden om detta och tråden som registrerar knapptryckningar skriver ut aktuellt tecken baserat på hur många gånger knappen tryckts ned. Problemet med denna lösning är att en användare som trycker sekvensen $\{2, 6\}$ i exemplet ovan förväntar sig att dessa tecken ska skrivas ut omgående, inte med en sekunds fördröjning. Annorlunda uttryckt, oavsett hur lång tid det är mellan 2 och 6 i en sekvens så ska den alltid generera strängen “an”. Innan en lösning på detta problem presenteras kan det vara värt att nämna att knapptryckningar skickas till en GUI-komponent via metoden *keyPressed(int key)* (se figur 5.4). Här är *key* ett heltal som motsvarar vilken knapp som blivit nedtryckt. För att veta vilken knapp heltalet motsvarar (vilket är olika på olika mobila enheter) jämförs heltalet med statiska sådana i J2ME-klassen *javax.microedition.lcdui.Canvas*. Metoden anropas av systemet.

I *keyPressed(int)* anropas först en metod vilken översätter heltalet till en position. Positionen pekar ut en sträng i en statisk vektor (eng. array). Denna sträng avgör vilka tecken som går att skriva med den knapp vilken motsvarar positionen. Som exempel ger position två strängen “abcåä2” eftersom knapp två förväntas ge någon av dessa symboler. (Beroende på hur många gånger knapp två trycks ned väljs alltså ett visst tecken. Två nedtryckningar ger ett “b”, fyra ger “å” osv.) En konstant (systemberoende minsta värdet för ett heltal) returneras om knappen som trycktes ned inte kan mappas mot någon av de 12 vanliga knapparna (0-9, stjärna samt brädgård).

Då ett objekt av typen *TextInput* skapas, bildas också en tråd - låt oss kalla den tidtagare

```
1  public synchronized void keyPressed(int keyCode)
2  {
3      int kc = reMapKeyCodes(keyCode);
4      if(kc != Integer.MIN_VALUE) {
5          notify();
6          try {
7              wait();
8          } catch(InterruptedException exception) {
9              exception.printStackTrace();
10         }
11         if(kc == BACKSPACE) {
12             counter = -1;
13             if(newChar != 0)
14                 newChar = 0;
15             else {
16                 removeFromInputString();
17             }
18         }
19         else {
20             if(prevKeyPress != kc && newChar != 0) {
21                 addToInputString(newChar);
22                 counter = -1;
23             }
24             if(keyMap[kc].length() > 0) {
25                 counter =  (counter + 1) % keyMap[kc].length();
26                 newChar = keyMap[kc].charAt(counter);
27             }
28         }
29     }
30     prevKeyPress = kc;
31 }
```

Figur 5.4: En tråd vars uppgift är att ta tid.

- vilken genast sätts i vänteläge. Den metod som motsvarar tråden syns i figur 5.5. Då det första tecknet skrivs in anropas *keyPressed(int)* för första gången. Här kontrolleras först om knappen som trycktes ned är någon av de 12 textinmatningsknapparna. Om så är fallet notifieras tidtagaren vilken nu går in i en oändlig loop samtidigt som *keyPressed(int)* går in i vänteläge. Tidtagaren sparar sedan undan systemtiden i millisekunder och notifierar *keyPressed(int)* varpå tidtagaren väntar i en sekund (eller till dess den blir notifierad). Nu körs alltså *keyPressed(int)* igång, då den precis blivit tillåten att göra så (samtidigt som den enda andra aktiva tråden satts i vänteläge). Här kontrolleras om knappen som

```
1  public synchronized void run() {
2      long time = 0;
3      try {
4          wait();
5      } catch(InterruptedException exception) {
6          exception.printStackTrace();
7      }
8      while(true) {
9          try {
10             notify();
11             time = System.currentTimeMillis() + 1000;
12             wait(1000);
13             if(time <= System.currentTimeMillis()) {
14                 counter = -1;
15                 if(newChar != 0) {
16                     char buf = newChar;
17                     newChar = 0;
18                     addToInputString(buf);
19                 }
20                 prevKeyPress = Integer.MIN_VALUE;
21                 repaint(); //Uppdaterar det grafiska gränssnittet.
22                 wait();
23             }
24         } catch(InterruptedException exception) {
25             exception.printStackTrace();
26         }
27     }
28 }
```

Figur 5.5: En tråd vars uppgift är att ta tid.

tryckts in var knappen för att radera ett tecken. Om så var fallet kontrolleras om den privata medlemmen *newChar* är skild från noll vilket i så fall indikerar att den sats av ett tidigare anrop till *keyPressed(int)*. Denna procedur är till för att användaren ska se vilket tecken som är på väg att skrivas ut, innan utskriften är verkställd. Om *newChar* har satts av ett tidigare anrop, tiden inte gått ut, och användaren tryckt på radera så ska inget verkställt tecken raderas utan endast det tecken som visas som hjälp ska raderas. Om däremot *newChar* är lika med noll så är detta en indikation på att den senaste knapptryckningen som gjorts är verkställd (tiden har gått ut) vilket innebär att ett tecken ska raderas från inmatningssträngen.

Om det inte var knappen för att radera görs följande kontroller:

- Är den nuvarande knapptryckningen skild från den förra knapptryckningen?
- Är *newChar* skild från noll?

Om båda dessa är sanna så indikerar detta att en ny knapp har tryckts ned samt att denna knapp inte är den första som tryckts ned. Detta i sin tur innebär att det tecken som motsvarar tidigare knapptryckningar (vilka alla varit på samma knapp) ska skrivas ut. Då utskriften har verkställts så sätts räknaren *counter* till minus ett. Vi återkommer till denna räknare. Om något av de båda uttrycken i listan ovan inte är sanna så verkställs inte utskriften av något tecken. Det som händer är att räknaren vi talade om tidigare ökas med ett. Denna räknare används för att avgöra vilket tecken i strängvektorn som är på väg att skrivas ut. Om användaren exempelvis tryckt tre gånger på knapp två så kommer *counter* vara lika med tre vilket indikerar att bokstaven på position tre i strängen "abcåä2" är på väg att skrivas ut. (Dock ej ännu verkställt.) Metoden avslutas med att lägga den aktuella knapptryckningen på minnet för att kunna jämföra med kommande knapptryckningar.
Vad händer då i *run*? För varje knapptryckning så kommer *keyPressed(int)* anropa *notify()* - *run()* kan fortsätta - vilket har två utfall:

- Tiden har inte passerats så *run* står på rad 12 och väntar. När notifieringen verkställs så kommer *run* med andra ord att vara på rad 13. Detta innebär en kontroll av att en sekund verkligen har förflutit. Detta är inte fallet nu och metoden tar tiden på nytt och väntar åter på rad 12. Varje knapptryckning inom tidsramen nollställer alltså tidtagningen.
- Tiden har passerat och användaren har med andra ord inte tryckt på någon knapp under en sekunds tid. Därmed går *run* in i *if*-satsen. Här verkställs utskriften av det nya tecknet om tecknet är skilt från noll. Enda gången *run()* tar sig till denna position och *newChar* är noll är då användaren tryckt på knappen för att radera. I detta fall

skall *run* endast nollställa medlemsvariabeln $prevKeyPress$ så att nästa anrop till *keyPressed* tolkar knapptryckningen oberoende av tidigare knapptryckningar. *run* ritar sedan om skärmen och ställer sig i vänteläge.

Sammanfattningsvis, för varje knapptryckning, skild från raderaknappen, startas en tidtagare vilken väntar i en sekund och sedan skriver ut aktuellt tecken. Varje knapptryckning ökar också en räknare som håller koll på vilket tecken som är aktuellt och huruvida användaren tryckt på samma knapp eller en ny knapp. I det förra fallet nollställs tidtagaren och användaren får ytterligare en sekund för att välja tecken. I det senare fallet skrivs tecknet ut direkt.

5.3 Fonetisk strängmatchning

Fonetisk matchning av strängar används (ofta vid hämtandet av information) för att identifiera ord som uttalas på liknande sätt, oavsett hur stavningen ser ut [28]. Vid hämtande av information är detta av intresse framförallt då användaren (den som vill hämta informationen) är osäker på stavningen av det eftersökta. Ett mycket bra exempel är då ett namn ska hämtas ur en databas, exempelvis "Rikard". Den information användaren är intresserad av kan mycket väl vara kopplade till namnet "Richard", vilket skiljer sig stavningsmässigt från det användaren angett som söksträng. En konventionell strängmatchning kommer skilja på dessa båda strängar och returnera informationen endast kring de poster som matchar "Rikard". För en telefonkatalogsapplikation är det alltså av stort intresse att ha någon form av fonetisk matchning som upptäcker likheterna mellan "Rikard" och "Richard" samtidigt som skillnaderna mellan exempelvis "Swensson" och "Swansson" bibehålls. Det finns en rad algoritmer för fonetisk matchning av strängar. De flesta tar dock endast hänsyn till det engelska språket och ger därför problem då användningsområdet huvudsakligen är svenska namn och efternamn. Dragen hos de vanligaste algoritmerna är dock snarlika och processen kan ofta generaliseras till följande:

- Transformera alla tecken till en kod vilken representerar tecknets uttal.
- Kontrollera kodernas inbördes beroenden. Vissa kodpar, kodtripplar och så vidare har ett speciellt uttal. Transformera beroendena till en kod för dess uttal. (Som exempel kan nämnas att "Ph" i svenska namn ofta blir "F".)
- Två likadana kodsträngar innebär att de ursprungliga strängarna uttalas på liknande sätt.

En äldre variant av fonetisk matchning är "Soundex" [13]. Algoritmen är snabb och relativt enkel. Olyckligtvis är den något förlåtande i den mening att många namn, vilka är uppenbart skilda, får samma soundexkod. Utan att i detalj förklara (den i övrigt, som sagt, enkla) algoritmen kan nämnas att namn som Lloyd och Liddy får samma kod (L300). Detta innebär att en sökning på Lloyd skulle ge oväntat många träffar och att många av träffarna skulle vara ointressanta för en användare. På grund av dess snabbhet och framförallt dess enkelhet har vi (trots moderna algoritmers överlägsenhet [28]) valt att använda en starkt modifierad version för att tillåta användaren att söka på "Eriksson" och få med de personer som stavar "Ericsson", "Erickson" eller till och med "Erixon". Likheterna mellan Soundex och implementationen beskriven nedan är framförallt att koden kan beräknas tecken för tecken[11]. Detta är en viktig egenskap om det ska vara möjligt att söka under tiden som användaren skriver in sin söksträng (vilket är ett krav, se bilaga B). I övrigt är likheterna av ringa karaktär.

Implementation Klassen som implementerar algoritmen kallas, trots de stora skillnaderna, för *Soundex*. Dess gränssnitt (med privata metoder synliga) syns i figur 5.6. Den utifrån sett viktigaste metoden är *soundex(char)* vilken använder sig av de privata metoderna för att beräkna koden för tecknet *c*. Då koden kan vara beroende av tidigare tecken finns de privata medlemmarna *previousSoundex* och *currentString*. Den förra är en

[11] Algoritmen behöver åtminstone inte hela strängen för att beräkna koden.

```
1  public class Soundex {
2
3      public Soundex()
4
5      //Privata medlemmar
6      private char[] mapping;
7      private String previousSoundex;
8      private String currentString;
9
10     //Privata metoder
11     private void init();
12     private String fixstart(String s);
13     private String fixend(String s);
14     private String fixH(String s);
15     private String fixDiphthongs(String s);
16     private boolean isVowel(char c);
17     private String transform(String s);
18     private String removeDoubles(String s);
19     private String soundex(String s);
20
21     //Publika metoder
22     public void newString();
23     public int remove();
24     public String soundex(char c);
25
26 }
```

Figur 5.6: Interfacet för klassen *Soundex* - med privata metoder synliga.

sträng innehållandes koden för den sträng som bildas av tidigare anrop till *soundex(char)*. Följande sekvens av anrop:

- soundex('p');

- soundex('h');

- soundex('i');

- soundex('l');

skulle alltså leda till att *previousSoundex* innehåller strängen "fil", vilket är koden för strängen "phil". *currentString* innehåller strängen bestående av alla tidigare inskickade

c till *soundex(char)*. Exemplet ovan skulle alltså generera *currentString* = "phil". Båda dessa strängar är nödvändiga för att avgöra vilken kod parametern *c* har. *currentString* används för att skapa en tillfällig kod av den hittills inmatade strängen. Denna kod jämförs sedan med *previousSoundex* för att avgöra om det senaste tecknet förändrar koden på något sätt. Vissa tecken kommer nämligen inte att förändra koden. Exempelvis så har "Kalleeee" och "Kalle" samma kod ("Kale") vilket innebär att de tre sista e:na inte förändrar koden. I figur 5.7 syns koden för *soundex(char)*. På rad sju syns jämförelsen av den tidigare koden och den nuvarande. Är dessa lika så returneras **null** vilket indikerar att tecknet som skickades in inte påverkar koden. Är de båda däremot olika (rad 13) så sätts *previousSoundex*

```
1  public String soundex(char c) {
2      if(Character.isDigit(c))
3          return new String("" + c);
4      currentString += c;
5      StringBuffer soundex = new StringBuffer(soundex(currentString));
6      if((soundex.toString()).equals(previousSoundex))
7          return null;
8      else if(currentString.length() <= 1)
9          return null;
10     else {
11         if(c == 'x' && soundex.length() != 0) {
12             previousSoundex = soundex.toString();
13             return new String("ks");
14         }
15         else {
16             String ret = "";
17             if(soundex.length() - previousSoundex.length() > 1)
18                 for(int i = ( previousSoundex.length() == 0 ?
19                     0 : previousSoundex.length() - 1 );
20                     i < soundex.length(); i++) {
21                     ret += soundex.charAt(i);
22                 }
23             else
24                 ret += soundex.charAt(soundex.length() - 1);
25             previousSoundex = soundex.toString();
26             return new String(ret);
27         }
28     }
29 }
```

Figur 5.7: Den publika metoden soundex(char).

till den nuvarande soundex-koden. Därefter returneras en sträng innehållandes den kod som lagts till sedan senaste anropet till *soundex(char)*. Det enda tecken som garanterat genererar fler än ett tecken i returströngen är "x" som ger koden "ks". Denna kontroll sker på raderna 14 och 18. På rad 10 undersöks längden på den hittills inskrivna strängen och om denna är 1 eller mindre så returneras inget tecken. Detta för att kunna returnera exempelvis "f" för "ph". I annat fall skulle "p" generera just "p" och då "h" anländer så kommer ett "f" returneras vilket leder till att klienten till soundex lever i tron att koden för "ph" är "pf" vilket alltså inte är sant. Det är också detta specialfall som leder till if-satsen på rad 20. På rad 2 kontrolleras om det inskickade tecknet är en siffra i vilket fall tecknet genast returneras, utan att påverka tillståndet hos soundexobjektet.

Det verkliga arbetet sker inte i den publika metoden utan i en privat *soundex(String)*. Metoden omvandlar en sträng innehållandes vanliga tecken till en sträng innehållandes koden för dessa tecken. Till sin hjälp har den en rad metoder. Figur 5.8 visar algoritmens gång. På rad 2 omvandlas strängen till gemener vilket är logiskt då "KeNT" och "kENt"

```
1  private String soundex(String s) {
2      s = s.toLowerCase();
3      s = fixstart(s);
4      s = fixend(s);
5      s = fixDiftongs(s);
6      s = transform(s);
7      s = fixH(s);
8      s = removeDoubles(s);
9
10     return s;
11 }
```

Figur 5.8: Den privata metoden soundex(String).

uttalas på samma sätt. Därefter (rad 3-4) korrigeras början och slutet på strängen. Detta görs separat då många specialfall förekommer just i början och slutet. På rad 5 görs samtliga kombinationer av bokstäver om till respektive kod, exempelvis är det här "ea" blir till "ä". Rad 7 gör därefter om alla tecken till sin kod, här blir alltså "ä" till "e". ("ae" genererar alltså koden "e".) Därefter tas specialfallet "h" om hand varpå alla koder som

förekommer fler än en gång i följd tas bort helt. Slutligen returneras strängen vilken nu innehåller koden för de tecken som inparametern innehåller.
Soundex innehåller ytterligare en publik metod, *remove()*. Denna är nödvändig för att kunna radera tecken ur det som hittills skickats in till *soundex(char)*. Returvärdet från *remove()* är ett heltal som beskriver hur många soundexkoder som ska raderas. Om tecknen "Peh" (början på Pehrsson) skickats in och ett tecken raderas så innebär detta inte någon förändring av soundexkoden varför remove kommer att returnera noll.
Resultatet av denna klass är alltså en implementation av en typ av fonetisk strängmatchning. Det ska dock tilläggas att transformationerna bygger på personlig intuition från författarnas sida, snarare än på lingvistiska sanningar. Metoderna är testade mot några av våra vanligare namn och efternamn och fungerar tillfredställande. Problem kan dock uppstå vid sökning på företag vilkas namn innehåller engelska ord eller ord med utländskt uttal. Detsamma gäller för namn med utländskt uttal. Problemen innebär dock aldrig att fullständigt rättstavade ord inte matchas - "Sound" kommer att matchas mot "Sound". Däremot kan felaktiga matchningar smyga sig in och matchningar som borde finnas kan tappas. Exempelvis kommer "sound" att matchas mot "sond" vilket stämmer ganska väl med svenskt uttal men mindre bra med engelskt.

5.4 Spara och ladda en trädstruktur

Det här avsnittet är ägnat åt att beskriva implementationen för att spara och ladda en trädstruktur. När databasen skapas sparas en trädstruktur ned till fil för att sedan ladda trädstrukturen från samma fil när applikationen väl startas i mobiltelefonen. Denna process kan naturligtvis ske på olika sätt, det enklaste är att låta de klasser som är nödvändiga för att spara trädet till fil implementera interfacet *Serializable* och sedan låta Java sköta biten med att skriva ned trädstrukturen till en fil. Problemet med denna metod är att J2ME inte har stöd för *Serializable*. Att spara objekt genom att serialisera dem är dessutom inte optimal vad gäller minnesutrymme, så vi sökte därför alternativa sätt att spara

trädstrukturen.

5.4.1 Traversera träd

Att traversera ett träd innebär att man "besöker" samtliga noder i trädstrukturen. Det finns två olika metoder för att göra detta; *depth-first traversal* och *breadth-first traversal*. Man kan kortfattat säga att *depth-first traversal* är ett rekursivt sätt att traversera i "djupet" på trädet medan *breadth-first traversal* är ett icke-rekursiv metod för att "besöka" noderna från vänster till höger. För vårt syfte så använder man sig av det förstnämnda *depth-first traversal*. *Depth-first traversal* kommer i tre olika varianter; *preorder*, *inorder* eller *postorder*. Skillnaden mellan dessa är ordningen för hur noderna "besöks". Nedan beskrivs alla tre tillsammans med pseudo-kod.
Preorder innebär att man "besöker" noderna enligt följande algoritm [20] [12]:

1. "Besök" rotnoden,
2. traversera vänstra subträdet,
3. traversera högra subträdet.

Namnet, *preorder*, kommer sig av att det faktum att man "besöker" rotnoden först. Pseudo-kod för algoritmen illustreras i figur 5.9.

Motsvarande algoritm för traversering i *inorder* blir:

1. Traversera vänstra subträdet,
2. "besök" rotnoden,
3. traversera högra subträdet.

Figur 5.10 visar pseudo-kod för inorder-traversering.

För *postorder* ser algoritmen ut som följer:

[12] Traverseringsalgoritmerna som beskrivs är för binära träd

```
1  private void preorder(Node node) {
2      visit(node);
3
4      if(node.getLeftChild() != null)
5          preorder(node.getLeftChild());
6      else
7          write(1); // "null"-nodes are marked as '1'
8
9      if(node.getLeftChild() != null)
10         preorder(node.getRightChild());
11     else
12         write(1); // "null"-nodes are marked as '1'
13 }
```

Figur 5.9: Pseudo-kod för metoden *preorder()* för noder i ett träd.

```
1  private void inorder(Node node) {
2      if(node.getLeftChild() != null)
3          inorder(node.getLeftChild());
4      else
5          write(1); // "null"-nodes are marked as '1'
6
7      visit(node);
8
9      if(node.getLeftChild() != null)
10         inorder(node.getRightChild());
11     else
12         write(1); // "null"-nodes are marked as '1'
13 }
```

Figur 5.10: Pseudo-kod för metoden *inorder()* för noder i ett träd.

1. Traversera vänstra subträdet,

2. traversera högra subträdet,

3. "besök" rotnoden.

I figur 5.11 syns pseudo-kod för postorder-traversering enligt algoritmen ovan.

```
1  private void postorder(Node node) {
2      if(node.getLeftChild() != null)
3          postorder(node.getLeftChild());
4      else
5          write(1); // "null"-nodes are marked as '1'
6
7      if(node.getLeftChild() != null)
8          postorder(node.getRightChild());
9      else
10         write(1); // "null"-nodes are marked as '1'
11
12     visit(node);
13 }
```

Figur 5.11: Pseudo-kod för metoden *postorder()* för noder i ett träd.

5.4.2 Uttryckskträd

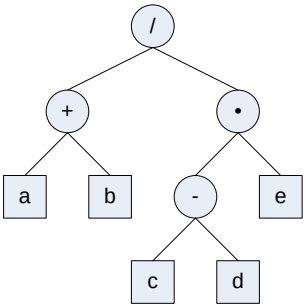
Uttryckskträd (eng. *expression trees*) innebär att man har en implicit trädstruktur i ett uttryck [20]. Detta illustreras lämpligast genom ett exempel med ett algebraiskt uttryck:

$$(a + b)/((c - d) \cdot e) \tag{5.1}$$

Uttrycket har en nedärvd trädstruktur där lövnoderna består av operander och de inre noderna av operatorer. Figur 5.12 är en trädrepresentation av uttrycket i ekvation 5.1. Här ser vi att parenteserna i uttrycket inte syns i trädstrukturen, detta beror på att deras funktionalitet ligger implicit i trädets struktur; t.ex. så ska $c - d$ evalueras innan multiplikationen med e kan ske. Om man traverserar trädet i *inorder* och skriver ut varje nod när den besöks så skulle vi få följande utskrift:

$$a,\ +,\ b,\ /,\ c,\ -,\ d,\ \cdot,\ e$$

Det är samma ordning på samtliga operander och operatorer som i ekvation 5.1, fast parenteserna fattas. Utskriften i *inorder* kallas *infix notation*, varje operator hamnar mellan sina operander [20]. Ett träd utskrivet i *infix notation* ger alltså upphov till tvetydighet eftersom att informationen som parenteserna gav har försvunnit. Utskriften ovan ska tolkas



Figur 5.12: Trädet som representerar uttrycket $(a + b)/((c - d) \cdot e)$.

som i ekvation 5.2, vilket inte är ekvivalent med uttrycket i ekvation 5.1.

$$a + b/c - d \cdot e \tag{5.2}$$

För att undvika denna sortens fel så måste man i utskriften för varje inre nod göra ett tillägg så att den först skriver ut en vänsterparentes och sist skriver ut en högerparentes. Ekvation 5.3 visar utskriften av ett sådant tillägg för en *inorder* traversering på trädet i figur 5.12.

$$((a + b)/((c - d) \cdot e)) \tag{5.3}$$

Detta uttryck är ekvivalent med uttrycket från ekvation 5.1.
Att däremot göra utskrift av trädet i *preorder*, vilket kommer generera ett uttryck i *prefix notation*, eller *postorder*, som genererar ett uttryck i *postfix notation*, kommer ge oss ett uttryck av trädet som inte behöver skriva ut parenteser. De uttrycken kommer alltså att kunna bygga upp samma träd igen. I litteraturen så beskrivs i synnerhet en metod för hur man m.h.a. en stack kan evaluera ett uttryck skrivet i *postfix* som vi har valt att använda oss av för att spara och ladda träd [20].

5.4.3 Spara trädstruktur

För att spara en trädstruktur så använder vi koden i figur 5.13 för inre noder och koden i figur 5.14 för lövnoder. Detta är ett polymorft anrop och den som anropar behöver alltså inte bry sig om vilken sorts nod som anropas.

```
1  public void writeToStream(DataOutputStream out) {
2  // Skriv ut vänster subträd till filen
3      if(left == null)
4          out.write((byte)1); // Ev. "null-nod" betecknas med 1
5      else
6          left.writeToStream(out);
7
8  // Skriv ut höger subträd till filen
9      if(right == null)
10         out.write((byte)1); // Ev. "null-nod" betecknas med 1
11     else
12         right.writeToStream(out);
13
14 // Skriv ut "mig" till filen
15     out.write((byte)2); // Inre nod betecknas med 2
16     out.writeInt(value); // Värdet på inre noden
17 }
```

Figur 5.13: Kod för utskrift av inre noder till fil i en trädstruktur.

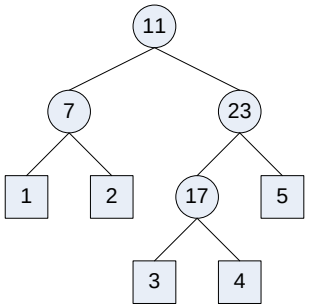
Koden för inre noder är en direkt implementation av en *postorder*-traversering. Lövnoderna skriver endast ut ett tecken som skiljer sig från "null"-noder för att indikera en riktig lövnod. Innehållet av den kan återskapas p.g.a att strukturen i trädet är sådan att lövnoderna ligger i nummerordning för traversering. Vid uppbyggnad av trädet från fil så kan man därför använda en räknare som inkrementeras för varje lövnod den stöter på. Detta kommer tydliggöras bättre i avsnitt 5.4.4 som handlar om att ladda träd.

```
1  public void writeToStream(DataOutputStream out) {
2      out.write((byte)0); // Lövnod betecknas med 0
3  }
```

Figur 5.14: Kod för utskrift av lövnoder till fil i en trädstruktur.

I Figur 5.15 så har vi ett exempel på ett litet, binärt sökträd. Notera numreringen på

lövnoderna.



Figur 5.15: Exempel på en trädsktruktur som ska sparas.

Om vi applicerar koden från figur 5.13 och 5.14 på trädet i figur 5.15 så erhåller vi följande utskrift, som för tydlighetens skull är komma-separerad:

$$\textit{0, 0, 2, 7, 0, 0, 2, 17, 0, 2, 23, 2, 11} \tag{5.4}$$

Denna sekvens av data, fast utan kommatecken, är alltså det enda som vi behöver spara ned i en fil för att sedan kunna bygga upp exakt samma trädstruktur som illustreras i figur 5.15.

5.4.4 Ladda trädstruktur

För att ladda in och rekonstruera ett träd från fil så använder vi en metod som m.h.a. en stack bygger trädet. För att kunna numrera lövnoderna så att trädet blir identiskt med ursprungsträdet så använder vi en räknare, som vi kan kalla *lövräknare*. Tecknen ifrån filen, som är i *postfix*-format, läses in från vänster till höger och processas så här:

1. Om nästa tecken är ’0’, så skapas en lövnod med värdet från *lövräknaren* och läggs på stacken.

2. Om nästa tecken är ’1’, så läggs en “null”-nod på stacken.

3. Om nästa tecken är ’2’, så skapas en inre nod som antar det värde som följer på ’2’:an. Två noder poppas från stacken och bildar barnen för denna nod, det översta blir högerbarn och det näst översta blir vänsterbarn.

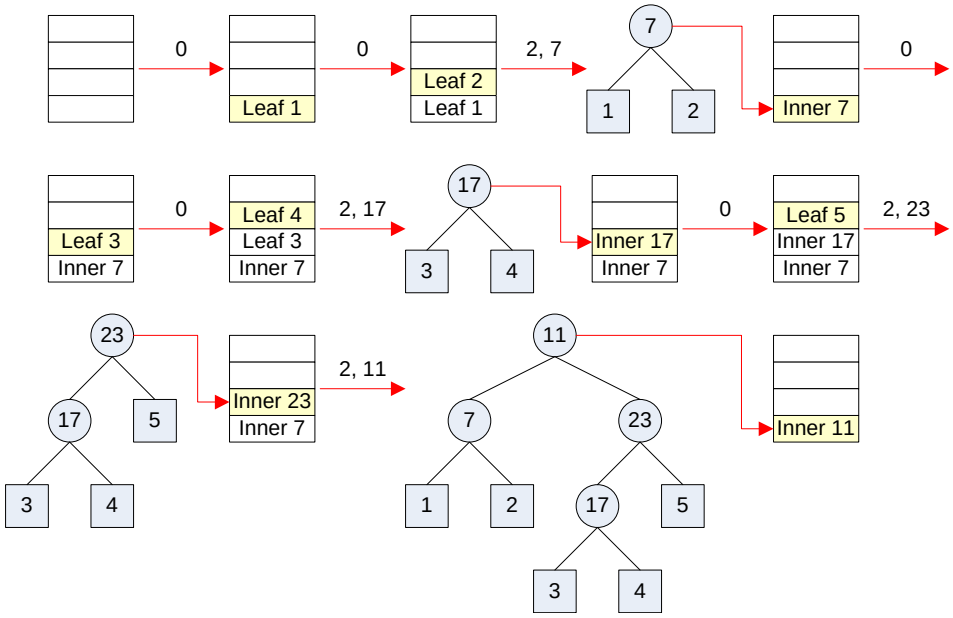
När sedan hela filen har processats så kommer root-noden för trädet ligga ensam på stacken. Implementation för algoritmen syns i figur 5.16.

```
1  public BST(InputStream in) {
2      DataInputStream d = new DataInputStream(in);
3      Stack s = new Stack();
4      int bucketCounter = 0; // Räknare för lövnoderna
5      int b = d.read();
6
7  // Läs tills slutet av filen
8      while(b != -1) {
9          switch(b) {
10             case 0: // Lövnod påträffades
11                 s.push(new BSTLNode(bucketCounter));
12                 bucketCounter++;
13                 break;
14             case 1: // ``null''-nod påträffades
15                 s.push(null);
16                 break;
17             default: // Inre node påträffades
18                 Node rNode = (Node)s.pop();
19                 Node lNode = (Node)s.pop();
20                 s.push(new BSTINode(d.readInt(),lNode, rNode));
21         }
22         b = d.read();
23     }
24     root = (Node)s.pop();
25 }
```

Figur 5.16: Kod för att ladda ett träd från fil.

Om vi använder datat från (5.4) i avsnitt 5.4.4 som genererades från trädet i figur 5.15 och använder detta som input till koden i figur 5.16 så kommer vi kunna återskapa trädet. Den processen beskrivs i detalj i figur 5.17. Datat som genererades, och som nu ska läsas in och återskapa trädet, hade följande utseende:

0, 0, 2, 7, 0, 0, 2, 17, 0, 2, 23, 2, 11



Figur 5.17: Process där träd byggs från datat i (5.4).

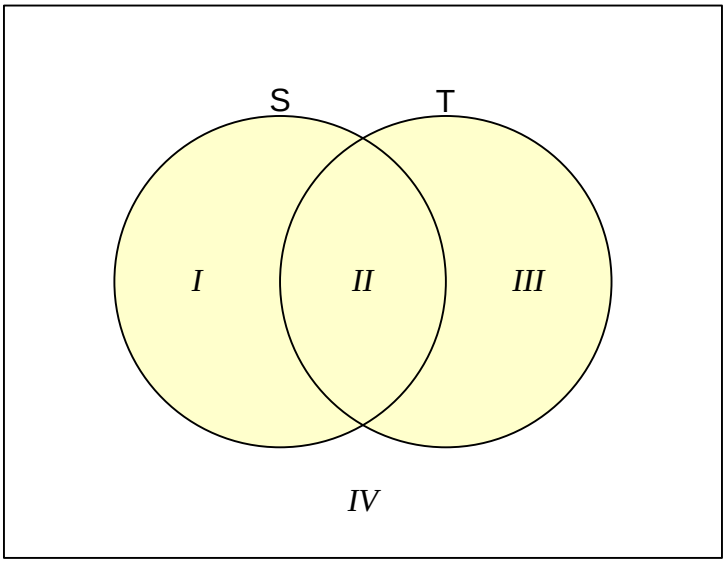
Här läses de första tecknen, som båda är nollor, in och vi skapar då lövnoder, baserade på *lövräknaren*, som vi sedan lägger på stacken. Den första lövnoden kommer skapas med värdet ’1’ från *lövräknaren*, sedan inkrementeras *lövräknaren* och nästa nod kommer få värdet ’2’. Nästa tecken som läses in är ’2’ och då ska två element poppas från stacken och bilda barn för den innernod som skapas. Innernoden antar det värdet som följer på ’2’:an, vilket i detta fall är 7. Den nya innernoden läggs sedan på stacken. Nästa tecken är en ’0’:a vilket kommer skapa lövnod nummer 3 och lägga den på stacken. Efter det skapas lövnod nummer 4 och läggs på stacken eftersom nästa tecken också är en ’0’:a. Nästföljande tecken är en ’2’:a, vilket innebär att de två översta elementen från stacken poppas och bildar barn till den nya innernoden med värdet 17 (som är nästa tecken i inläsningen) som sedan läggs på stacken. Sedan skapas ytterligare en lövnod och genom samma procedur som har beskrivits tidigare så skapas efter det en innernod med värdet 23 när ’2’:an processas. Sist så kommer den sista, rotnoden, att skapas när innernoden med värdet 11 skapas. Den kommer binda ihop hela trädstrukturen och sedan läggas på

stacken. När hela den här processen är avklarad så är det alltså bara att poppa det, enda, element på stacken för erhålla trädstrukturen som tidigare hade sparats.

5.5 Mängdlära och Relationsalgebra

I databasteknik så behöver man de grundläggande matematiska koncepten från mängdläran för att hämta viss typ av information ur databaser [6]. Det är aktuellt då man vill hämta information från en datakälla baserat på olika villkor. Avsikten med att ha flera villkor i en sökning är antingen för att expandera sökrymden eller begränsa den. De olika villkoren kommer medföra att flera mängder kommer genereras. En mängd kan beskrivas som en samling *element* som ofta har någon gemensam egenskap. En mängd skulle t.ex. kunna bestå av personer med förnamnet "Erik" eller av personer med efternamnet "Andersson". Anta att vi är intresserade av mängden som består av de personer som heter "Erik" i förnamn **och** "Andersson" i efternamn, hur ska det gå till? För att göra detta så måste man ha *operatorer* på mängder. Det existerar ett antal operatorer för mängder och vi tänkte ta upp *union*, $\cup$, och *snitt*, $\cap$ [13]. I figur 5.18 illustreras de två mängdoperationer som vi hade tänkt att ta upp i den här rapporten i ett *venn-digramm*. Ett venn-diagram beskriver mängder som regioner i ett plan. I figuren illustreras två mängder, S och T, som cirklar. Rektangeln (den svarta ramen) begränsar *universum* för exemplet. *Universum* är **alla** element i den domänen som är aktuell. Den mängd som innehåller alla element i universum kallas *universalmängden*, betecknad U. Den mängd som inte innehåller något element kallas den *tomma mängden* och betecknas $\emptyset$. I tabell 5.1 syns vilka regioner från venn-digrammet i figur 5.18 som olika mängder består av. Mängden som operationen $S \cup T$ (unionen) utgör är alltså de element som finns i S eller T (eller båda). För operationen $S \cap T$ (snittet) blir mängden de element som finns i S och T, d.v.s. de som är gemensamma för S och T.

[13] Det finns andra mängdoperationer, förutom de som vi har nämnt, men då vi inte använder dessa så har vi valt att inte ta upp dem här



Figur 5.18: Venn-diagram.

Tabell 5.1: Olika mängder och deras relation till venn-diagrammet i figur 5.18

Mängd	Region(er) i figur 5.18
U	I, II, III, IV
S	I, II
T	II, III
$S \cup T$	I, II, III
$S \cap T$	II

Vi återgår till frågeställningen om hur man hämtar ut de personer som heter “Erik” i förnamn **och** “Andersson” i efternamn, och definierar mängderna A och B som:

$$A = \{x | x \text{ heter “Erik” i förnamn}\}$$
$$B = \{x | x \text{ heter “Andersson” i efternamn}\}$$

Mängden, R, för de personer som både heter “Erik” och “Andersson” är alltså:

$$R = A \cap B$$

Av denna anledning så måste en implementation av *snitt* finnas i databasen.
Antalet sökfält i applikationen är endast ett, vilket innebär att söksträngen som matas in kan vara formaterad på en rad olika sätt. Användaren kan t.ex. mata in ett efternamn följt av ett förnamn eller tvärtom, enbart ett förnamn eller efternamn, ett förnamn eller efternamn som består av flera "namn". Nedan följer några exempel:
Andersson Erik
Erik Andersson
Jan Erik Svensson
Rupert
van der Ster
osv.
Vi har alltså ingen vetskap om vilken typ av information användaren avser att söka på när denne matar in sitt sökuttryck. En sökning måste därför ske på samtliga (sökbara) fält i databasen och returneras i resultatet. Låt oss säga att söksträngen är "Rupert". Alla som heter "Rupert" förnamn ska naturligtvis ge träff på sökningen och lika självklart ska alla som heter "Rupert" i efternamn ge träff på sökningen. Vi formulerar mängden A och B som:

$$A = \{x | x \text{ heter "Rupert" i förnamn}\}$$
$$B = \{x | x \text{ heter "Rupert" i efternamn}\}$$

Då ska resultatet, R, av sökningen vara:

$$R = A \cup B$$

5.5.1 Snitt och union på sorterad lista

I litteraturen beskrivs ett antal olika sätt att implementera mängder och metoder för union och snitt [20]. Samtliga lösningar baseras på att strukturen för att representera en mängd allokerar universalmängden och sedan markerar de element som ska ingå i den aktuella mängden från universalmängden. Om universalmängden t.ex. är heltalen $0, 1, 2, \cdots, 31$ så allokeras en array av typen *boolean* med storleken 32 element: *boolean[] array = new boolean[32];*. Samtliga element i *array* är från början satta till *false*, när sedan element läggs till i mängden så sätts de positioner som elementet motsvarar i *array* till *true*. Fördelen med en sådan lösning är att implementationen för union och snitt blir väldigt enkel då man kan använda de logiska operatorerna *och*, och *eller*, för varje element på två mängder för att genomföra snitt respektive union på mängderna. Nackdelen är att man måste allokera så mycket som hela universalmängden för varje mängd, även om bara ett element ingår i mängden. För att hålla nere minnesanvändandet så fick vi förkasta den implementationen och konstruera en annan.

När en sökning har genomförts på ett fält (t.ex. förnamn) så kommer en lista att returneras innehållande index för de poster som matchade sökningen. Har flera sökord angivits i sökfältet så ska snittet av de listor som returneras appliceras. Att ta snittet på två listor skulle innebära att varje element i den ena listan ska jämföras med alla element från den andra och de gemensamma elementen returneras i en tredje lista. Pseudo-kod för en sådan implementation illustreras i figur 5.19. En sådan algoritm har $O(n^2)$ och är därför väldigt ineffektiv i våra sammanhang. Tester av en sådan implementation visade att en sökning på ett vanligt förnamn tillsammans med ett vanligt efternamn tog hela 16 sekunder att genomföra i mobiltelefonen[14]. En sådan fördröjning är naturligtvis inte acceptabel och vi fick försöka optimera algoritmen. Algoritmen i figur 5.19 är en *allmän algoritm* som alltså fungerar på två listor som kan vara osorterade. I vår applikation har vi vetskap om att dessa listor är sorterade. Listorna är sorterade för när man hämtar information så vill vi ha

[14]I testet användes telefonen Sony-Ericsson W800i

```
1  private Set intersection(Set s1, Set s2) {
2      Set intersection = new Set();
3      foreach(element e in s1) {
4          foreach(element e in s2) {
5              if(s1 == s2)
6                  intersection.add(s1);
7          }
8      }
9
10     return intersection;
11 }
```

Figur 5.19: Pseudo-kod för snitt på två mängder

en bestämd sorteringsordning och vi kan därför "hårdkoda" den ordningen. Hur kan vi då utnyttja det faktum att listorna är sorterade? I figur 5.20 illustreras två sorterade listor, de gemensamma elementen är 12 och 20 vilket kan deduceras genom en effektiv men inte helt trivial algoritm. Algoritmen som vi har konstruerat bygger på att vi först väljer ut

Lista 1
7
12
17
20

Lista 2
1
5
12
15
16
20
27

Figur 5.20: Två sorterade listor.

den lista med minst antal element (lista 1 i figur 5.20). Denna lista kopieras till *resultatlistan*, den lista som kommer innehålla snittet. Sedan tar vi bort de element ur *resultatlistan* som inte finns representerade i den andra listan (lista 2 i figur 5.20). Borttagandet sker på ett sådant sätt att initialt väljs det första elementet i *resultatlistan* detta värde jämförs sedan, i sekvens från början, med den andra listan. Om det visar sig att värdet i *resultatlistan* är större än värdet i den andra listan så väljs nästa element ur *resultatlistan* och det föregående tas bort. Sökningen fortsätter nu på samma sätt som tidigare tills det att

alla element ur *resultatlistan* har traverserats. I Figur 5.20 skulle alltså först 7 från, lista 1, jämföras med $1, 5, 12$ eftersom att $12 > 7$ så skulle nästa element från lista 1 väljas och 7 tas bort. Nästa element är 12 och det ska då jämföras med det senaste från lista 2, vilket också råkar vara 12. Då tas 12 *inte* bort från lista 1, men däremot väljs nästa värde, 17. Proceduren fortsätter och till slut återstår endast $12, 20$ i lista 1, vilket också är snittet på de sorterade listorna.

Koden i figur 5.21 visar en implementation av algoritmen. Metoden i det exemplet tar emot en lista med sorterade listor och returnerar snittet för samtliga av dessa listor.

Algoritmen för union är sker på liknande sätt fast där väljs den lista med flest element istället och byggs på med de element ur den andra listan som inte redan existerar (för att undvika dubbla poster).

Efter att ha implementerat denna algoritm så reducerades söktiden från c:a 16 sekunder till endast 250-300 millisekunder för samma sökning.

```
1  private Vector intersection(Vector sets) {
2      switch(sets.size()) {
3          case 0:
4              return new Vector();
5          case 1:
6              return (Vector)sets.firstElement();
7          default:
8              Vector v = (Vector)sets.firstElement();
9              for(Enumeration e=sets.elements(); e.hasMoreElements();) {
10                 Vector u = (Vector)e.nextElement();
11                 if(u.size() < v.size())
12                     v = u;
13             }
14             sets.removeElement(v);
15
16             Vector intersection = new Vector();
17             for(Enumeration e=v.elements(); e.hasMoreElements();)
18                 intersection.addElement(e.nextElement());
19
20             for(int i = 0; i<sets.size(); i++) {
21                 Vector addresses = (Vector)sets.elementAt(i);
22                 int low=0;
23                 for(int j=0; j<intersection.size(); j++) {
24                     boolean hit = false;
25                     int pivot = ((Integer)intersection.elementAt(j)).
26                                                     intValue();
27                     for(int k=low; k<addresses.size() &&
28                             pivot >= ((Integer)addresses.elementAt(k)).
29                             intValue() && !hit; k++) {
30                         low = k+1;
31                         hit = pivot == ((Integer)addresses.elementAt(k)).
32                                                     intValue();
33                     }
34                     if(!hit)
35                         intersection.removeElementAt(j--);
36                 }
37             }
38
39             return intersection;
40     }
41 }
```

Figur 5.21: Snitt.

6 Slutsats

Uppgiften, som presenterades av uppdragsgivaren "The Phone Pages of Sweden AB", gick ut på att skapa en mobil telefonkatalogtjänst som skulle fungera fristående. Detta innebar att inga utomstående beräknings- eller datakällor fick användas vilket med dagens mobila enheter ställer stora krav på programvaran. Detta avsnitt beskriver vilka lösningar som använts och vilka problem som stötts på under projektets gång. Avslutningsvis finns ett avsnitt om framtida arbete.

6.1 Resultat och utvärdering

I kapitel 3 redogjordes för den inledande fasen av vårt arbete. Detta bestod huvudsakligen i att utvärdera vilka möjligheter som finns tillgängliga vad gäller olika aspekter av mjukvaran samt de resultat varje aspekt gav. Nedan tas samtliga aspekter och deras respektive resultat kortfattat upp. Här syns också vilka lösningar som slutligen användes.

Spara data Då det tar tid att sätta in nya element i en "record store" samtidigt som data i internminnet blir för resurskrävande så återstår två alternativ. Det ena innebär att tilläggspaketet FCOP används vilket leder till att applikationen måste signeras av tredje part för att användaren inte ska störas av säkerhetsfrågor från systemet. Det andra alternativet är att lägga data som resursfiler i JAR-paketet. Detta sista alternativ valdes då nackdelarna med FCOP undviks utan att införa nya, för applikationen relevanta, nackdelar.

Organisera data För att minimera access-tiden samtidigt som applikationen tar upp så lite minne som möjligt valdes en lösning med något vi kallar för konstgjord "random access" (Se avsnitt 3.3) genom att dela upp en logisk fil i flera mindre fysiska filer[15]. Anledningen till att vi var tvungna att simulera "random access" på det sättet var för att J2ME inte

[15]En fil är en logisk konstruktion vilket kan göra att läsaren opponerar sig mot uttrycket "fysiska filer". Här avses dock ytterligare en högre abstraktionsnivå vilket delvis rättfärdigar uttrycket.

har inbyggt stöd för att använda filpekare. Lösningen gjorde att information kan hämtas c:a 100 ggr snabbare än tidigare.

Indexera data Sökningen kan snabbas upp ytterligare genom att indexera token-tabellen. De alternativ som undersöktes för detta (indexering av strängar) var B-träd och trie hashing. Här valdes trie hashing vilket beror främst på dess enkla och "lättviktiga" struktur och att de kan beskrivas med en finit automaton. Dessutom behöver trädet skrivas till fil vilket i fallet med trie hashing låter sig göras relativt enkelt. För att indexera telefonnumren så användes ett binärt sökträd.

Presentera data J2ME har på grund av den stora diversiteten bland mobila enheter vissa problem med gränssnittsprogrammering. Av denna anledning valdes ett tredjepartsbibliotek för att underlätta skapandet av de komponenter som krävs för att applikationen ska kunna realiseras. Biblioteket finns i öppen källkod och släpps under GNU GPL. Det finns även kommersiella licenser och det blir upp till uppdragsgivaren att avgöra vilken licens som lämpar sig bäst. Bibliotekets namn är J2ME Polish.

Komprimera data Komprimering av data innebär dekomprimering för att återfå den ursprungliga informationen. Detta i sin tur leder till ökade resurskrav, främst i form av processorkraft. Utöver denna problematik är det för en databas av stor vikt att data är åtkomligt på godtycklig plats i databasen. Detta diskvalificerar många av dagens effektiva komprimeringsalgoritmer varför enklare, så som Byte Pair Encoding, lämpar sig bättre. Då den aktuella datamängden inte översteg de maxvärden som definierats i kravspecifikationen ansåg vi dock att komprimering inte var nödvändigt.

Stöd för kartor För att applikationen ska kunna stödja kartor måste någon annan teknik hittas än de som redogjorts för i denna rapport. Det är inte uteslutet att vektorbaserad grafik är lösningen på problemet. Detta lämnas dock som en öppen fråga och resultatet är

endast att det inte är möjligt att implementera stöd för kartor om GIF-filer används. [16]

Den produkt som slutligen överlämnades till uppdragsgivaren är i enlighet med kravspecifikationen och innehåller den funktionalitet som finns beskriven i bilaga B, krav märkta R1.0. Utöver de krav märkta R1.0 så har även krav nr.150 implementerats. Krav nr.150 säger att en sökning ska ha en "feltolerans" på ett tecken, vilket löstes m.h.a. fonetisk strängmatching (se avsnitt 5.3). För att produkten ska kunna sättas i skarp produktion krävs dock ytterligare en komponent för att generera databasfilerna utifrån befintlig data. Den testdatabas som byggts i detta projekt utgick från en xml-fil och applikationen som skapar databasen är därför hänvisad till inläsning från sådan fil. I en skarp produkt borde databasen-filerna genereras med hjälp av en JDBC-koppling till slutkundens[17] databas. Avslutningsvis bör det tilläggas att applikationen enligt författarna hade fungerat effektivare och varit enklare att implementera om den varit en tunn klient vilken använde sig av en utomstående databasserver.

6.2 Problem

Överlag har arbetet flutit på bra och de hinder som under tiden har uppstått har ofta varit av mindre art. De flesta kan klassas in under ovana med utvecklingsmiljön. Detta gäller främst arbetet med J2ME Polish vilket, på grund av sitt sätt att hantera skapandet av användargränssnitt, måste förkompilera vissa delar av koden. För att detta ska gå smidigt krävs en övergripande kunskap i hur Netbeans (via Apache Ant) hanterar kompilering. Hur detta går till ligger dock utanför rapportens omfång och vi hänvisar därför den intresserade till respektive webb-plats:

- Apache Ant - http://ant.apache.org
- J2ME Polish - http://www.j2mepolish.org

[16] "Möjligt" ska ställas relativt kravspecifikationen.
[17] Slutkund är exempelvis Lokaldelen, Eniro o.s.v, inte slutanvändaren med andra ord.

Ett problem av allvarligare grad var att analysfasen inte riktigt färdigställdes innan implementationsarbetet påbörjades. Detta ledde till vissa mindre lyckade implementationsval. Bland annat visade det sig att de mobiler som klarade av programmet ofta klarade sig nästan lika bra utan att söka samtidigt som användaren matar in text. Skillnaden känns och märks men är inte så pass påtaglig att det hade stört - och framförallt uppfylldes kravspecifikationen i båda fallen. Ytterligare ett större problem har varit angående avlusning (eng. debugging). Då applikationen byggs på en annan plattform än värdplattformen så krävs pålitliga emulatorer. Mobiltillverkarna har ofta egna emulatorer vilket fungerar bra så länge kompilering sker helt under Netbeans inflytande. Problemet var dock att få dessa emulatorer att fungera med J2ME Polish - vilket aldrig löstes. Istället fick vi förlita oss på den generiska emulator Sun[18] själva tillhandahåller vilket inte alltid resulterade i tillförlitligt beteende.

6.3 Framtida arbete

Arbete för framtiden innefattar bland mycket annat att vidare effektivisera algoritmerna. Ett sätt är att göra om varje hashtrie till en finit automaton och på så sätt hålla trädet i programkod. Detta liknar en parsingalgoritm och kan fungera eftersom den katalog som datat beskriver också beskriver ett språk med giltiga lexem och syntax. Om informationen i katalogen ofta ändras är detta inte möjligt vilket gjort att detta lagts på framtiden då kravspecifikationen innehåller krav som kan göra att informationen kan ändras av användaren själv.

För att underlätta sökningen implementerades i detta projekt en soundex-liknande algoritm. Denna tar dock ingen hänsyn till huruvida en viss stavning matchar bättre än en annan. Det innebär att oavsett om användaren söker på “Persson”, “Pehrsson” eller “Pearsson” så kommer resultaten att presenteras i bokstavsordning. Detta kan upplevas irriterande då en användare vilken söker på “Pearsson” troligen vill ha den med denna

[18]Sun är företaget bakom J2ME och Java.

stavning först. En förbättring vore alltså att ranka de olika stavningarna och presentera de som matchar bäst först.

Ytterligare effektiviseringar rör komprimering där det, som beskrevs i avsnitt 3.6, är passande med BPE.

Applikationen kan enkelt förberedas för tilläggspaketet FCOP (vilket beskrev i avsnitt 3.2) vilket skulle innebära att applikationen kan frigöras från informationen i en given katalog. Detta skulle innebära att användaren kan ladda hem applikationen och därefter de databasfiler som är relevanta för användaren. Detta kräver dock att programmet verifieras av en pålitlig tredjepart vilket ökar kostnaden för utvecklingen. Däremot skulle det underlätta uppdateringen av databasen vilket är ett annat steg som under alla omständigheter bör tas. Ett sätt är beskrivet i bilaga B, krav 136.

Sökning och implementering av kartor är ytterligare en komponent som bör ligga i ett framtida arbete liksom lokal sökstatistik där varje klient kommer ihåg vad användaren sökt på. Denna statistik kan sedan användas för att bygga upp ett T9-liknande system där användaren inte, via flera knapptryckningar, behöver “bokstavera” varje tecken i ett ord. Istället kan applikationen själv föreslå att användaren som hittills matat in “and” ska söka på “anders” eller “andersson”. Med hjälp av statistiken kan det vanligaste sökordet ges som första förslag.

Referenser

[1] Alfred V. Aho, John E. Hopcroft, Jeffrey Ullman, J. D. Ullman, and J. E. Hopcroft. *Data Structures and Algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[2] Rudolf Bayer and Karl Unterauer. Prefix b-trees. *ACM Trans. Database Syst.*, 2(1):11–26, 1977.

[3] Abdelghani Bellaachia and Iehab AL Rassan. Efficiency of prefix and non-prefix codes in string matching over compressed databases on handheld devices. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 993–997, New York, NY, USA, 2005. ACM Press.

[4] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture: a system of patterns.* John Wiley & Sons, Inc., New York, NY, USA, 1996.

[5] Harvey M. Deitel and Paul J. Deitel. *Java How to Program (6th Edition).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004.

[6] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems, Fourth Edition.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[7] Yu Feng and Jun Zhu. *Wireless Java Programming with J2ME.* Sams, Indianapolis, IN, USA, 2001.

[8] Robert E. Filman and Daniel P. Friedman. *Coordinated computing: tools and techniques for distributed software.* McGraw-Hill, Inc., New York, NY, USA, 1984.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[10] Ivor Horton. *Ivor Horton's Beginning Java 2, JDK 5 Edition.* Wrox Press Ltd., Birmingham, UK, UK, 2004.

[11] Sun Microsystems Inc. The cldc hotspot implementation virtual machine, 2002. Tillgängligt på Internet: `http://java.sun.com/products/cldc/wp/cldc_hi_whitepaper.pdf` [Hämtad: 06.10.02].

[12] Jonathan Knudsen and Sing Li. *Beginning J2me: From Novice To Professional (Beginning from Novice to Professional).* Apress, Berkely, CA, USA, 2005.

[13] Donald Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, 2nd edition, 1998.

[14] James F. Kurose and Keith W. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet Package*. 2nd edition.

[15] Witold Litwin. Trie hashing. In *SIGMOD '81: Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 19–29, New York, NY, USA, 1981. ACM Press.

[16] David J. C. MacKay. *Information Theory, Inference & Learning Algorithms*. Cambridge University Press, New York, NY, USA, 2002.

[17] Department of Defense. Department of defense world geodetic system 1984 its definition and relationships with local geodetic systems, third edition, 2000.

[18] E. Otoo and S. ah. Red-black trie hashing. In *Technical Report TR-95-03*, Carleton University, Ottawa, Canada, 1995.

[19] Pointbase. Product datasheet: Pointbase micro, 2004. Tillgängligt på Internet: `http://www.pointbase.com/resourcecenter/pdfs/micro.pdf` [Hämtad: 06.10.05].

[20] Bruno Richard Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*. Wiley, 2000.

[21] William Pugh. Compressing java class files. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 247–258, New York, NY, USA, 1999. ACM Press.

[22] Greg Riccardi. *Principles of Database Systems with Internet and Java Applications*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[23] W. Clay Richardson, Donald Avondolio, Joe Vitale, Scot Schrager, Mark W. Mitchell, and Jeff Scanlon. *Professional Java, JDK 5 Edition*. Wrox Press Ltd., Birmingham, UK, UK, 2005.

[24] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

[25] Claude E. Shannon and Warren Weaver. *A Mathematical Theory of Communication*. University of Illinois Press, Champaign, IL, USA, 1963.

[26] TeliaSonera. Topplista för augusti 2006. Tillgängligt på Internet: `http://www.teliasonera.se/externalarticlenm/?hier=13053&mainUrl=http%3A%2F%2Fwww.sonera.fi%2FpressProviderWeb%2Fresources%2Fjsp%2FProvider%`

2FgetArticle.do%3Flocale%3DSV%26articleId%3D228770 [Hämtad: 06.09.17] [Senast ändrad: 06.09.04].

[27] Robert Virkus. *Pro J2ME Polish: Open Source Wireless Java Tools Suite*. Apress, Berkely, CA, USA, 2005.

[28] Justin Zobel and Philip Dart. Phonetic string matching: lessons from information retrieval. In *SIGIR '96: Proceedings of the 19th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 166–172, New York, NY, USA, 1996. ACM Press.

A Projektbeskrivning

Examensarbete – Mobile Search - Java

Syfte

Syftet med detta examensarbete är att ta fram en mobil sökklient för lokala telefonkataloger. Tanken är att hitta en optimal lösning att ladda ner gula och vita sidor för begränsade geografiska områden.

Produkt

Projektet ligger inom området programmering och specifikt programmering i Java där vi använder J2EE i servers och J2ME i mobiler. Produkten innebär att en slutanvändare kan göra sökningar i sin mobiltelefon på följande:

- Privatpersoner (vita sidorna)
- Företag (rosa sidorna)
- Kategori (gula sidorna inklusive några bildannonser)
- Karta

Sökningen skall ske lokalt i telefonen och gälla specifika områden (tex Karlstad eller Uppsla), dvs all data finns på klienten.
Systemet skall köras på Linux servrar och vi använder MySQL databasserver.

Arbetsuppgift

Arbetet innebär att tillsammans med the PhonePages göra en kravspecifikation utifrån både tekniska och marknadsmässiga krav. Därefter skall produkten utvecklas, färdigställas och göras klar för en eventuell release. Mobilklienten har en begränsad storlek, vilket kommer att innebära att en del av arbetet går ut på optimering av katalogdata.
Målet är att klienten skall fungera för de tio mest vanliga mobiltelefontyperna på marknaden.

Projekt och Tidplan

Klienten skall finnas tillgänglig för release februari 2007. Det föreslås att projektet delas in i flera olika steg:

- Analys och kravspecifikation: 2 veckor
- Kravnedbrytning:2 veckor
- Systemdesign: 2 veckor
- Implementation: 8 veckor
- Test: 3 veckor
- Pilottest: 1 vecka
- Dokumentation: 2 veckor
- release

All nödvändig programvara och datorresurser kommer att tillhandahållas på plats.
Examensarbetet kan utföras av 1-2 teknologer

Handledare: Per-Åke Minborg, CTO, civilingenjör Elektroteknik CTH
per-ake.minborg@thephonepages.com tel: 0702-692402

Kontakt person: Carina Dreifeldt, CEO, civilingenjör Elektroteknik CTH
Carina.dreifeldt@thephonepages.com tel:0702-692401

Arbetet utförs på the PhonePages kontor
The PhonePages
Västra Hamngatan 21
411 17 Göteborg
031-7430700

Mer information om bolaget på: www.thephonepages.com

B Kravspecifikation

REQUIREMENT SPECIFICATION FOR LOCALLY ACCESSIBLE PHONE DIRECTORY (LAPD)

Abstract

This document specifies the requirements for the product Locally Accessible Phone Directory (LAPD) product.

Application

This document is valid for Locally Accessible Phone Directory release 1. Some requirements for release 2 have also been included.

Table of contents

1 Reader´s Guideline 3
2 Introduction 3
3 Releases 3
4 Reference Model 4
5 Functional Requirements 4
5.1 LAPD System 4
5.2 LAPD Interface 8
5.3 LAPD Security 9
5.4 LAPD Signaling 9
6 Characteristics 9
6.1 Performance 9
6.2 Reliability 10
6.3 Maintenance and Support 10
6.4 Changeability and Adaptability 10
6.4.1 Migration 10
7 Design Restrictions 10
7.1 Development Platform 10
7.2 Target Platform 11
7.3 Standards 11
7.4 Algorithms 11
8 References 11
8.1 Informative References 11
9 Terminology 12
9.1.1 Definitions 12
9.1.2 Abbreviations 12
10 Appendices 12
10.1 Revision History 12

1 READER´S GUIDELINE

Terminology and references have been but at the end of the document although they are valid throughout this paper.

2 INTRODUCTION

The LAPD service allows mobile users to lookup information about people and companies, such as names, phone numbers and addresses. The information is stored locally (hence the name) on the mobile device used, e.g. cellular phone, PDA, and covers the white and yellow pages for a given city or region.

3 RELEASES

In order to distinguish requirements from general or informative statements, all requirements are marked with an [R]. Requirements valid in release 1.0 of the LAPD are tagged with [R1.0] additional requirements supported in release 2.0 are tagged [R2.0] and for those supported in release 3.0 the tag [R3.0] will be used.

The LAPD release plan is defined as follows:

LAPD release 1.0 contains white and yellow pages for at least one phone directory. This release is for internal purpose only and not for end customer.

LAPD release 2.0 introduces map functionality.

4 REFERENCE MODEL

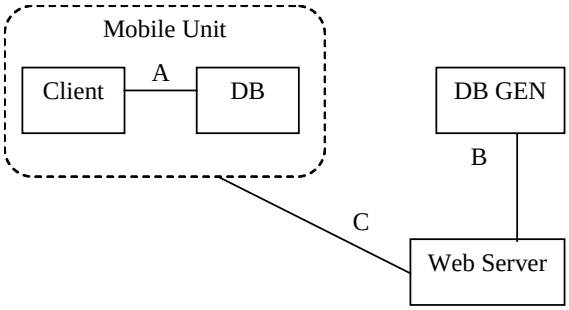


Figure 1 LAPD reference model

The LAPD reference model comprises three distinct parts:

1. Mobile Unit - E.g. a cellular phone or PDA. It Consists of two parts, which are bundled together in a Jar-file in the mobile unit:
   1.1. Client - The application running on the mobile unit.
   1.2. DB – The Database containing white and yellow pages for the mobile unit.

2. DB GEN – Database Generator – Generates the database file for a specific city or region. Information is originally collected from a third party provider.

3. Web Server - Hosting the content of the LAPD. A Web Server can be operated by a variety of providers, operators and ISPs.

Interface A represent the connection between the application and it's database.
Interface B illustrates the relationship between the DB GEN and the Web Server. The database is first generated and then bundled together with the application.
Interface C adheres to standard Internet protocols (HTTP).

The below figure illustrates how the three above entities can be interconnected in a real case.

5 FUNCTIONAL REQUIREMENTS

5.1 LAPD SYSTEM

The LAPD system enables the ability to search for contact information even if the mobile unit is not connected to Internet.

[100, R1.0] The database shall cover both white and yellow pages.

[101, R1.0] A white pages entry shall have the following fields:
- First name

- Surname
- Street address
- Street number
- Postal code
- City
- Phone number
- Map location WGS84 datum (R2.0)

[102, R1.0] A yellow pages entry shall have the following fields:
- Name
- Street address
- Street number
- Postal Code
- City
- A list containing an arbitrary number of phone numbers
- Logotype optionally
- Map location WGS84 datum (R2.0)
- Priority

[103, R1.0] A field category table is required to support searching in categories.

[104, R1.0] A field category entry shall have the following fields:
- Category name
- List of all companies belonging to the category

[105, R1.0] The fields that are searchable shall be:
- First name (White pages)
- Surname (White pages)
- Telephone number (White/Yellow pages)
- Field category (Yellow pages)
- Company name (Yellow pages)

[106, R1.0] If the number of search hits are 1, all information about the entry corresponding to the search hit shall be displayed.

[107, R1.0] If the number of search hits are more than 3, only information about name and address in the entries corresponding to the search hits shall be displayed.

[108, R1.0] The higher the priority field in a yellow pages entry is, the higher up in the search list it shall be shown.

[109, R1.0] When the client is started 1 of 4 advertisements are selected randomly and is shown below the search field, until the user begins to type in a search string. This is called a splash advertisement.

[110, R1.0] If a user clicks on the splash advertisement, all information about the advertiser shall be shown.

[111, R1.0] The client application shall be available in Swedish.

[112, R1.0] All documentation shall be in English.

[113, R1.0] The maximum number of keys pressed to commence a search must be less than two.

[114, R1.0] There should be exactly one search field.

[115, R1.0] When the user has commenced a search the application presents the search result as the number of search hits for the yellow and white pages respectively.

[116, R1.0] If no hits were detected a chord in A minor (Am) in octave 3 shall be played.

[117, R1.0] If hits were detected a chord in C major (C) in octave 5 shall be played.

[118, R1.0] A click on either the number of search hits for the yellow or white pages should take the user to a list of all the search hits on the chosen pages, showing 50 hits a time.

[119, R1.0] To navigate through the search hits, the user should not be constrained to a menu system but rather a movement of the joystick should show the next/previous 50 search hits.

[120, R1.0] A move to the right on the joystick should show the next 50 search hits, if more exists. A move to the left on the joystick should show the previous 50 search hits, if some exist.

[121, R1.0] To navigate among the 50 search hits, a scroll bar shall be used.

[122, R1.0] The user shall be able to call a located contact.

[123, R1.0] Detailed information on a particular subscriber shall be shown in a list so that different phone-numbers (e.g. fixed, mobile or fax) can be called.

[124, R1.0] The user shall be able to shut down the application at any time by pressing the keyboard one or several times.

[125, R1.0] The user shall be able to save a record in the Mobile Unit's phone book or SIM card, if and only if the users unit supports JSR 75, PIM optional package, otherwise the user shall not be presented with this option.

[126, R1.0] The user shall be able to make a new search at any time by pressing the keyboard one or several times.

[127, R1.0] If and only if the search string is spelled exactly as a field in the database, the entry corresponding to that field is returned as a search hit.

[128, R1.0] By pressing the designated <BACK> button, the user shall be able to stepwise go back from detailed search to search list to search (with latest search word(s) to empty search field.

[129, R1.0] On the main search page there shall be a menu button containing the options <Sök>, <Rensa>, <Hjälp>, <Om…>

[130, R1.0] The <Hjälp> menu shall contain information on how to download the program and other relevant help text.

[131, R1.0] The <Om…> menu shall contain program name, revision and a copyright notice.

[132, R1.0] The name of the Java application shall be “Lokaldelen” + “<name of local area>”.

[133, R1.0] Whenever the first character is numerical (0..9) then it shall be assumed that the following characters are also numerical.

[134, R1.0] It shall be possible to select “map” whenever detailed information is shown for a user (it is possible to show the map under the detailed information directly?)) The map shall be the same for all subscribers and can be provided as a static PNG picture.

[135, R1.0] An alias table is required to support searching in category aliases.

[136, R2.0] The user should, after a configurable time, get a question asking her to update the client and the database.

[137, R2.0] The database shall contain maps over the local area.

[138, R2.0] It should be possible to see the location of a certain address on the map when an entry is selected.

Note: The map should not be searchable in any other way than through a search in the yellow or white pages.

[139, R2.0] The search should begin as soon as the user begins typing in a search string.

[140, R2.0] The client application shall support other languages than Swedish.

[141, R2.0] The user shall be able to send an SMS to a located contact, if and only if the users unit supports JSR 120, WMA optional package.

[142, R3.0] It shall be possible to activate a second search page with detailed search options such as “begins with”, “sounds like”, etc.

[143, R3.0] Each client shall keep local search statistics.

[144, R3.0] If the client is updated, the local search statistics shall be sent to the Web Server, tracking global search statistics.

[145, R3.0] The maps shall be searchable using street addresses as input.

[146, R3.0] Clicking the map shall start a search for phone numbers of persons and companies situated in that area.

[147, R3.0] Search from the Internet shall be supported.

[148, R3.0] The user shall be able to get the search hit list presented on the map where each hit corresponds to a mark on the map and each mark corresponds the location of the search hit.

[149, R3.0] Travel instructions shall be supported.

[150, R3.0] The client shall find fields usually spelled in another way, for example, a search for the surname Svensson shall also find persons who's surname is Swensson. The fault tolerance is one character:

- Svensson – Swensson, shall find
- Svensson – Swenson, shall not find
- Karin – Carin, shall find
- Karin – Carina, shall not find
- Karin – Karina, shall find

This is true if and only if the number of search hits is less than 5.

[151, R3.0] It shall be possible for the user to insert new records or modify the current ones in the database.

5.2 LAPD INTERFACE

[200, R1.0] Downloading of a client and the bundled database, to an intermediate computer, takes place through HTTP.

[201, R1.0] To download a new database, the user must download a completely new client since clients are bundled with the database.

[202, R1.0] To transfer the client to the mobile unit, the unit's usual file transfer software is used, either through USB or Bluetooth. This is device dependent and up to the device manufacturer to support.

[203, R2.0] It shall be possible to download the client and the bundled database directly to the mobile unit through WAP.

[204, R2.0] The Web Server shall support download initiation through WAP push.

[205, R2.0] The Web Server shall, for each download, log:

- User Agent
- Date and time
- MSISDN
- Program variant (area)
- Program version

5.3 LAPD SECURITY

[250, R1.0] No security is needed.

[251, R2.0] It should be too costly, either through time or resources, for an attacker to corrupt or otherwise damage the integrity of the data contained in the Database Generator.

Note: An attacker is anyone who, deliberately or otherwise, attempts to damage the integrity of the system. This includes users that do so unknowingly of their actions.

[252, R2.0] It shall not be possible, with reasonable resources, to decompile the database content.

[253, R2.0] The Client shall be digitally signed.

5.4 LAPD SIGNALING

[300, R1.0] All signaling shall be based on the IP.

6 CHARACTERISTICS

6.1 PERFORMANCE

[350, R1.0] The Client shall count the number of entries matching a search string in no more than 500 ms.

[351, R1.0] The Client shall list the first 50 search hits in no more than 500 ms.

[352, R1.0] The Client shall, upon request from user, list the next/previous 50 search hits in no more than 500 ms.

[353, R1.0] The Client initial startup time shall take no more than 4 seconds longer than to start up a HelloWorld MIDlet.

[354, R1.0] The Database shall be able to handle 100 000 records.

[355, R1.0] The Client shall not occupy more than 5 MB of persistent storage memory. This is without counting any logotypes.

[356, R1.0] The Client shall not allocate more than 250 kB of RAM memory.

[357, R1.0] The Client shall not access the Web Server during any stage of a search.

[358, R2.0] The (download) Web Server shall respond within 2 seconds from a request.

6.2 RELIABILITY

[400, R1.0] The LAPD system shall pass a black box test suite in JUnit, designed to cover all the functional requirements stated in this document.

[401, R2.0] The web server shall have an 99.9% uptime.

6.3 MAINTENANCE AND SUPPORT

[450, R1.0] A word document shall be available describing how to change, build and deploy the application.

[451, R2.0] Documentation and Manuals shall be available in English electronically.

[452, R2.0] Tools for fast and accurate configuration of DB GEN shall be provided.

6.4 CHANGEABILITY AND ADAPTABILITY

6.4.1 MIGRATION

This section contains requirements regarding migration from previous system versions to subsequent ones.
The client application is a standalone program and do not need to be backward compatible.

[R2.0] The DB GEN generates a database structure that fundamentally conforms to that of release 1.0.

7 DESIGN RESTRICTIONS

7.1 DEVELOPMENT PLATFORM

[500, R1.0] The programming environment shall be NetBeans IDE 5.0 or higher.

[501, R1.0] The programming language of the DB GEN shall be Java (J2SE).

[502, R1.0] The programming language of the Client shall be Java (J2ME).

[503, R1.0] The programming language of the Web Server shall be Java (J2SE).

7.2 TARGET PLATFORM

The client application will run on all new devices supporting J2ME with the following configurations and profiles:

- MIDP 2.0 (JSR 118)
- CLDC 1.1 (JSR 139)

Containing at least 5 MB of memory for JAR-files and 250 kB of JVM memory.

7.3 STANDARDS

[550, R2.0] The LAPD system shall not require change of any underlying standards e.g. for MIDP 2.0 and Internet protocols.

7.4 ALGORITHMS

[600, R1.0] The Database is bundled with the client application into a JAR-file using the ZIP algorithm for compression.

[601, R2.0] Further compression shall be attained using the Huffman compression algorithm.

8 REFERENCES

8.1 INFORMATIVE REFERENCES

[JSR118] "JSR 118: Mobile Information Device Profile 2.0", Mark VandenBrink, Nov 2002. URL: jcp.org/en/jsr/detail?id=118

[JSR139] "JSR-000139 Connected Limited Device Configuration 1.1", Antero Taivalsaari, March 2003. URL: jcp.org/aboutJava/communityprocess/final/jsr139/

[JSR120] "JSR 120: Wireless Messaging API", Marquart C Franz, Aug 2002. URL: jcp.org/en/jsr/detail?id=120

[JSR75] "JSR 75: PDA Optional Packages for the J2METM Platform", Tom Chavez, Jun 2004. URL: jcp.org/en/jsr/detail?id=75

[WAP] "Wireless Application Protocol Architecture Specification", WAP Forum, April 30, 1998. URL: http://www.wapforum.org/

9 TERMINOLOGY

9.1.1 DEFINITIONS

See reference [PP Arch].

9.1.2 ABBREVIATIONS

JSR	Java Specification Request
HTTP	HyperText Transfer Protocol [RFC2068]
WAP	Wireless Application Protocol [WAP]
MIDP	Mobile Information Device Profile
CLDC	Connected Limited Device Configuration
J2ME	Java2 Micro Edition
J2SE	Java2 Standard Edition
J2EE	Java2 Enterprise Edition
JAR	Java ARchive
JVM	Java Virtual Machine (for J2ME JVM is actually called KVM)
PIM	Personal Information Management
ISP	Internet Service Provider

10 APPENDICES

10.1 REVISION HISTORY

Version	Changes	Changed by
1.0	Initial versions	Sebastian Skoglund Per-Erik Svensson
1.0a	Minor revision	Sebastian Skoglund Per-Erik Svensson