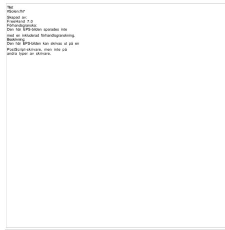


Datavetenskap

Per Davidsson och Patrick Jungner

Optimering av MIDletapplikationer

Examensarbete, C-nivå

2002:15

Optimering av MIDletapplikationer

Per Davidsson och Patrick Jungner

© 2002 Patrick Jungner, Per Davidsson och Karlstads universitet

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

Per Davidsson

Patrick Jungner

Godkänd, 2002-06-04

Handledare: Robin Staxhammar

Examinator: Stefan Alfredsson

Sammanfattning

Att ha en mobiltelefon är idag nästan lika självklart som att ha en vanlig telefon. I o m att mobiltelefoner blir kraftfullare finns nu möjligheten för tredjepartstillverkare att utveckla applikationer som kan laddas ned och köras i mobiltelefoner. En miljö, som tillåter att applikationer kallade MIDlets körs i en mobiltelefon, har utvecklats av Sun Microsystems och kallas för MIDP. Denna miljö finns för närvarande implementerad i två mobiltelefoner tillgängliga i Sverige. Dessa är Siemens SL45i och Motorola Accompli 008. En mobiltelefons minne och processorkraft är begränsade, interaktionen med användaren är långsam och besvärlig p g a de små knapparna och den lilla skärmen och kommunikationen är osäker och långsam. Därför finns det mycket man behöver tänka på när man skall utveckla en MIDlet.

Vårt arbete har bestått i att undersöka vad en programmerare, som utvecklat applikationer till vanliga datorer, behöver tänka på när han skall skapa en MIDlet till en mobiltelefon. Genom att söka information på Internet, i tidsskrifter och i böcker fick vi många råd och tips som kunde kompensera för de begränsningar vi nämnt ovan. I denna rapport finns denna information sammanställd tillsammans med våra egna tankar och rekommendationer som uppkommit när vi utvecklat och undersökt MIDlets. Det område som vi har funnit mest information om, är hur man kan optimera sin MIDlet för att den skall exekvera snabbare. Även för den som vill minska MIDletens minnesanvändande finns det mycket information i denna uppsats.

Det vi själva har upplevt som den största begränsningen med mobiltelefonerna vi testat är dess små knappar och lilla display. Detta område tar vi också upp, men här finns mer att göra för mobiltelefontillverkarna än applikationsprogrammerarna.Vissa optimeringar kan dock göras för att underlätta för användarna.

Vid arbetets slut sammanställde vi de resultat och rekommendationer vi funnit under arbetes gång. Vi är nöjda med resultatet som blivit en guide för de som skall utveckla MIDlets och hoppas att de som läser detta har nytta av denna uppsats. Vi upplever att vi har gjort en bra arbetsinsats och vi har lärt oss mycket under arbetets gång.

Optimization Of MIDlet Applications

Abstract

Having a cellular phone today is almost as usual as having a regular phone. As phones becomes more powerful it is now possible for third party developers to develop applications, which can be downloaded and executed in the cellular phone. An environment which allows applications, so called MIDlets, to execute in a cellular phone, has been developed by Sun Microsystems and the environment is called MIDP. At the moment this environment is implemented in two cellular phones available in Sweden. These are Siemens SL45i and Motorola Accompli 008. A Cellular phone has a limited memory and and a limited processor capacity, the interaction with the user is slow and cumbersome because of the small buttons and the small screen and the communication is slow and insecure. Because of this, there is a lot to think about when developing MIDlets.

In our work we have concentrated on what a programmer, who has developed applications for ordinary computers, should think about when developing MIDlets for a cellular phone. By searching the Internet and reading papers and books we got a lot of ideas on how to compensate for the limitations described above. In this bachelor project this information is put together along with our own thoughts and recommendations. The area in which we found most information is how to optimize the MIDlet to make it execute faster. The essay also contains lots of information on how to make the memory usage as small as possible.

The largest limitation that we experienced with the cellular phone is its small buttons and its small display. We also point out some of these limitations, but this is more of a problem for the manufacturers of the cellular phones than the application programmers, although some optimizations could be done to help the user.

At the final phase of the project we compiled the results and recommendations we found during our work. We are satisfied with the result, which has become a guide for those who shall develope MIDlets, and hope that those who read the essay find it useful. We feel that we

have done a good work and that we have learned a lot while working on the project.

Tack

Vi vill tacka våra handledare på universitetet och på Telia Mobile för den hjälp som vi fått.

Tack till: **Robin Staxhammar, Karlstads Universitet**
Håkan Blomkvist, Telia Mobile

Vi vill också tacka Telia Mobile för att vi fick chansen att göra vårt exjobb på företaget.

Innehållsförteckning

1 Inledning 1

1.1 Bakgrund 1

1.2 Syfte och mål 1

1.3 Uppsatsens upplägg 2

2 Java 2 Micro Edition och Mobile Information Device Profile 3

2.1 Java 2 Standard Edition - J2SE och Java 2 Enterprise Edition - J2EE 3

2.2 Java 2 Micro Edition – J2ME 4

2.3 Connected Limited Device Configuration – CLDC 5

2.4 Mobile Information Device Profile – MIDP 6

2.5 MIDP:s virtuella maskin 6

2.6 Application Management Software – AMS 7

3 Källkod 9

3.1 Testmetoder 9

3.2 Objekt 10
3.2.1 Rekommendationer
3.2.2 Analys

3.3 Klasser 12
3.3.1 Rekommendationer
3.3.2 Analys

3.4 Primitiva typer 13
3.4.1 Rekommendationer
3.4.2 Analys
3.4.3 Tester

3.5 Omvandlingar 15
3.5.1 Rekommendationer
3.5.2 Analys

3.6 Inlining 16
3.6.1 Rekommendationer
3.6.2 Analys

3.7 Generella lagringsklasser 17
3.7.1 Rekommendationer
3.7.2 Analys

3.8 Loopar 18
3.8.1 Rekommendationer
3.8.2 Analys
3.8.3 Tester
3.9 Metodanrop 21
3.9.1 Rekommendationer
3.9.2 Analys
3.10 Undantag 22
3.10.1 Rekommendationer
3.10.2 Analys
3.11 Inbyggda programsatser och operatorer 23
3.11.1 Rekommendationer
3.11.2 Analys
3.11.3 Tester
3.12 Beräkningar 24
3.12.1 Rekommendationer
3.12.2 Analys
3.13 Matriser, Arrayer och Länkade Listor 25
3.13.1 Rekommendationer
3.13.2 Analys
3.13.3 Tester
3.14 CharArrayer, SträngBuffertar och Strängar 27
3.14.1 Rekommendationer
3.14.2 Analys
3.15 Variabler 28
3.15.1 Rekommendationer
3.15.2 Analys
3.15.3 Tester
3.16 Blandade tips 30
3.16.1 Rekommendationer
3.16.2 Analys
3.17 Obfuscating 31
3.17.1 Rekommendationer
3.17.2 Analys
3.17.3 Tester
3.18 Starka kontrakt 34
3.18.1 Rekommendationer
3.18.2 Analys
3.18.3 Tester

4 GUI 39

4.1 Högnivå API 39
4.1.1 Rekommendationer
4.1.2 Analys
4.2 Översikt lågnivå API 40
4.2.1 Rekommendationer
4.2.2 Analys
4.3 En MIDlet som anpassar sig eller en MIDlet för varje mobiltelefonmodell? 41
4.3.1 Rekommendationer
4.3.2 Analys

4.4 Designtips gällande laddningstider, navigering mm 42
4.4.1 Rekommendationer
4.4.2 Våra rekommendationer
4.4.3 Analys
4.5 Information som en MIDlet kan få från en mobiltelefon 44
4.5.1 Analys
4.6 Siemens SL45i API 48
4.6.1 Analys
4.7 Motorolas API 49
4.7.1 Rekommendationer
4.7.2 Analys
4.8 Storlek på bilder och text 50
4.8.1 Analys
4.8.2 Tester

5 Kommunikation 53
5.1 Http-förbindelser 53
5.1.1 Analys
5.1.2 Tester
5.2 Sessioner 62
5.2.1 Analys

6 Exekvering 65
6.1 Synkronisering och trådar 65
6.1.1 Rekommendationer
6.1.2 Analys
6.1.3 Tester
6.2 Input / Output 66
6.2.1 Rekommendationer
6.2.2 Analys
6.3 Record Management System - RMS 67
6.3.1 Rekommendationer
6.3.2 Analys
6.3.3 Tester

7 Resultat och rekommendationer 75
7.1 Design 76
7.2 Användbarhet 76
7.3 Minnesoptimering 76
7.4 Tidsoptimering 77
7.5 Storleken på jarfilen 79

8 Slutsatser 81

Referenser 83

A Specifikation för Examensarbete 87
A.1 Sammanfattning 87

A.2 Grundläggande Information 87
A.2.1 Bakgrund ex-jobb/utredning
A.3 Mål 88
A.3.1 Ex-jobbets mål
A.3.2 Omfattning/avgränsning
A.3.3 Strategi
A.4 Rapportering 89
A.5 Projektets avslutande 90
B Brev 91
B.1 Fråga om GUI 91
B.2 Svar från Midletsoft 91
B.3 Svar från Uppli 92
B.4 Svar från Reqwireless 92
B.5 Svar från CoreJ2ME 94
C MyConnector-klass 95
D Metoder från klassen HttpConnection 99
E Http-connection 101
F Servlet 103
G Ordlista 105

Figurförteckning

Figur 2.1: Denna bild illustrerar sambandet mellan de olika Javateknologierna. 4

Figur 2.2: Figuren visar de byggblock som behövs för att exekvera Javaapplikationer i en mobiltelefon. 5

Figur 2.3: Beskrivning av de olika delar som finns i en mobiltelefon. [43] 5

Figur 5.1: Liten svartvit PNG-bild 53

Figur 5.2: Illustration av sambandet mellan en mobiltelefon och en serverapplikation som gör det möjligt att simulera bl a RMI. [18].. 54

Figur 5.3: Figurerna visar hur de olika skärmbilderna kan se ut. [16] 57

Tabellförteckning

Tabell 3.1: Exekveringstid för int 14
Tabell 3.2: Primitiva typers minnesutnytjande 14
Tabell 3.3: Tidsåtgång vid jämförelse 21
Tabell 3.4: Exekveringstid för en switchsats och en if-elsesats 24
Tabell 3.5: Accesstid för int, int-array och int-matris 26
Tabell 3.6: Arrayers och matrisers minnesutnytjande 27
Tabell 3.7: Defaultvärden 29
Tabell 3.8: Accesstid för objekt på olika minnesplatser 29
Tabell 3.9: Kodstorlek innan och efter obfuscering 33
Tabell 3.10: Minnesåtgång vid användande av starka kontrakt kontra undantag 37
Tabell 3.11: Tidsåtgång vid användning av starka kontrakt kontra undantag 37
Tabell 5.1: Nedladdning av 87 bytes till Siemens SL45i 59
Tabell 5.2: Nedladdning av 253 bytes till Siemens SL45i 60
Tabell 5.3: Nedladdning av 411 bytes till Siemens SL45i 60
Tabell 5.4: Nedladdning av 87 bytes till Motorola Accompli 008 61
Tabell 5.5: Nedladdning av 253 bytes till Motorola Accompli 008 61
Tabell 5.6: Nedladdning av 411 bytes till Motorola Accompli 008 62
Tabell 6.1: Tidsåtgång vid användande av RMS med Siemens SL45i 70
Tabell 6.2: Tidsåtgång vid användande av RMS med Motorola Accompli 008 72
Tabell 6.3: Tidsåtgång när data hämtas och bild skapas på Siemens SL45i 73
Tabell 6.4: Tidsåtgång när data hämtas och bild skapas på Motorola Accompli 008 73
Tabell 6.5: Tidsåtgång när data hämtas och bild skapas på Motorola Accompli 008 74

1 Inledning

1.1 Bakgrund

Under en relativt snar framtid kommer många av de mobila terminaler som släpps på marknaden att ha inbyggda virtuella Javamaskiner, vilka används för att kunna köra Javaprogram som är skrivna för J2ME som är Suns javateknologi för begränsade apparater som t ex mobiltelefoner och handdatorer. Detta gör det möjligt för användarna att hämta hem applikationer från Internet som exekveras lokalt i terminalen. Detta är en mycket intressant teknik med många möjligheter. Applikationerna kommer att kunna exekveras lokalt i telefonen och i viss mån utnyttja telefonimetoder såsom att ringa samtal, skicka SMS, komma åt adressboken o s v Applikationerna kan också med hjälp av de mobila näten ha Internetkoppling, d v s de kan skicka och hämta data från Internet, dock med de begränsningar som de mobila näten och J2ME har.

Att mobila terminaler öppnas för tredjepartsapplikationer på detta sätt är nytt. Traditionella applikationsutvecklare är inte vana att göra program för terminaler med liten processor, liten display, litet minne, små eller inga knappar och har en anslutning till Internet via en dyr och osäker förbindelse. Telia Mobile AB har därför gett oss i uppdrag att se över vad applikationsutvecklarna behöver tänka på då de utvecklar applikationer till dessa begränsade terminaler.

1.2 Syfte och mål

Vi ska undersöka vad man bör tänka på när man skall utveckla program till mobiltelefoner vad gäller källkod, GUI, kommunikation, exekvering, hårdvara och nätprestanda. Dessa områden har vi valt i samarbete med personal från Telia Mobile, se bilaga A för Krav-specifikationen. Vårt mål är att ta fram en guide/hjälpmedel för de mjukvaruutecklare som ska utveckla programmera Java-applikationer till mobiltelefoner, så kallade MIDlets. Vår guide riktar sig främst till personer som har programmerat i Java tidigare och som nu ska utveckla MIDlets. De bör då vara medvetna om mobiltelefonernas begränsningar, som t ex att minneskapaciteten och processorkraften är starkt reducerad jämfört med vanliga pc-datorer.

Detta leder till att man måste ta hänsyn till dessa begränsningar då man utvecklar applikationer till mobila enheter.

1.3 Uppsatsens upplägg

Vi börjar med att förklara de begrepp som vi använder oss av i denna rapport, vad J2ME är och hur exekveringsmiljön för Java är uppbyggd för en mobil plattform. När vi skriver mobiltelefoner i detta dokument menar vi mobiltelefoner som kan exekvera MIDlets. De som finns tillgängliga i dagsläget är Siemens SL45i [34] och Motorola Accompli 008 [25]. Vi går också igenom vad profiler och konfigurationer är.

Sedan kommer de kapitel som handlar om vår uppgift, att optimera en MIDlet. Vi har fyra huvudkapitel vilka tar upp olika områden. Dessa är källkodsoptimering, GUI, kommunikation och exekvering. I kapitlet om källkod ligger tyngdpunkten på att skriva koden så att en MIDlet exekverar snabbt och tar upp så lite minne som möjligt. Optimeringarna avvägs också mot den ökade komplexitet man får i koden. Vad som gör att användaren upplever applikationen som lättanvänd tar vi upp i kapitlet om GUI. Där finns också information om hur man anpassar sin applikation till den aktuella exekveringsmiljön. I kapitlet om Kommunikation kan man läsa om vilka operationer, vid kommunikation, som tar tid, och några sätt att minska denna tidsåtgång. I kapitlet om Exekvering kan man läsa om trådar och läs- och skrivoperationer till det beständiga minnet. I detta kapitel ligger tyngdpunkten på att begränsa tidsåtgången för operationer med trådar och skrivning/läsning till/från det beständiga minnet.

De två sista kapitlen beskriver de resultat, rekommendationer och erfarenheter vi fått och de slutsatser vi dragit under arbetets gång. Kapitlet Resultat och rekommendationer är ett bra minneskapitel för den som läst igenom rapporten en gång och vill använda de rekommendationer som föreslagits, utan att behöva läsa igenom alla kapitel igen.

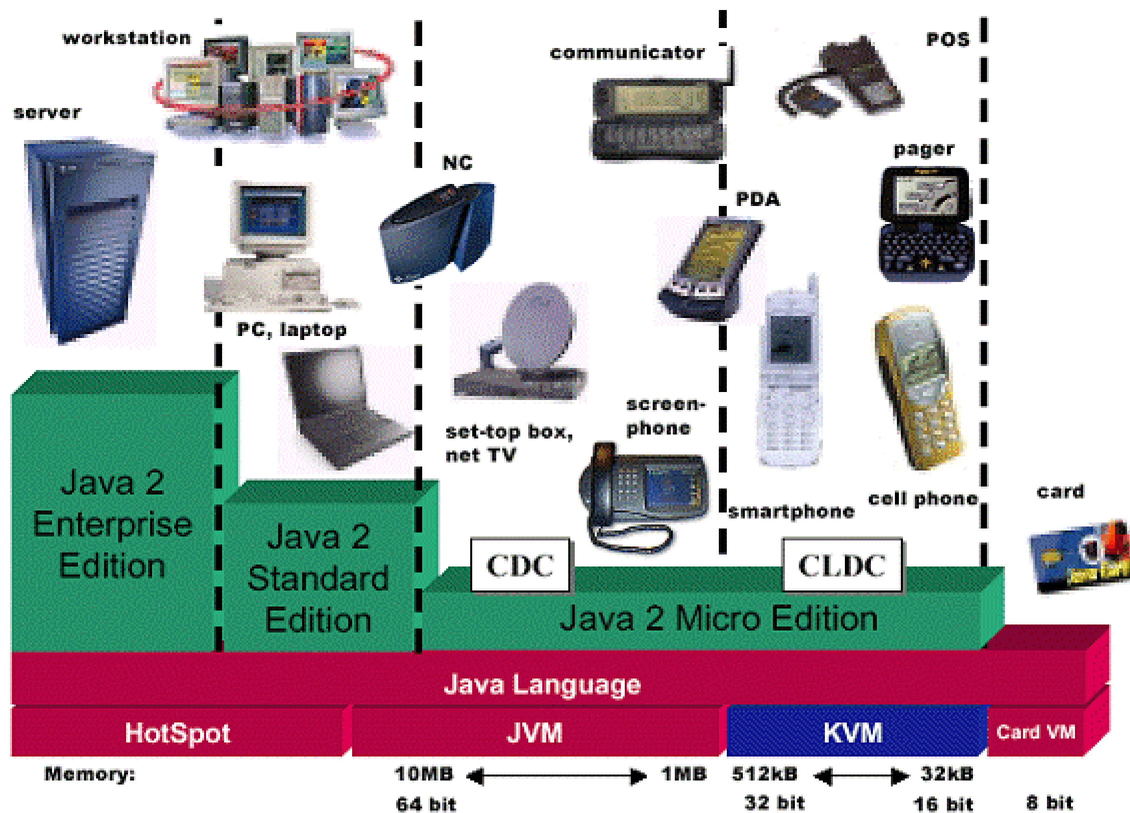
I bilaga G finns en ordlista där alla förkortningar i denna uppsats är beskrivna.

2 Java 2 Micro Edition och Mobile Information Device Profile

I detta kapitel går vi igenom vad J2EE, J2SE och J2ME är. Tyngdpunkten kommer att läggas på J2ME och en av dess profiler MIDP, därför att det är denna del vi undersökt i uppsatsen. Först ger vi en övergripande beskrivning av J2EE och J2SE. Därefter tar vi upp J2ME och ger en överblick av vad en konfiguration och en profil är för något. Sedan går vi igenom CLDC, Connected Limited Device Configuration, som är den konfiguration som används till mobiltelefoner. CDC är en konfiguration som är lik CLDC men den har inte samma hårda systemkrav som CLDC. CDC kan användas till programmering av något kraftfullare apparater. Denna konfiguration tar vi inte upp närmare. Därefter tas MIDP, Mobile Information Device Profile, som är den profil som finns i mobiltelefonen upp. Vi kommer sedan att gå igenom vad en virtuell maskin är och vilka skillnader som finns i J2SE:s virtuella maskin och den som används av J2ME. Sist går vi igenom AMS, Application Managment Software som är den mjukvara som är ansvarig för installation och exekvering av Javaapplikationen. Javaapplikationer till mobiltelefoner kallas för MIDlets.

2.1 Java 2 Standard Edition - J2SE och Java 2 Enterprise Edition - J2EE

J2SE, Java 2 Standard Edition, är den plattform som man använder för att utveckla applikationer till pc-datorer. Tyngdpunkten ligger på applikationer som ofta har en kort levnadstid och som oftast inte har krav på sig att alltid vara tillgängliga. Ett exempel på applikationer man utvecklar med J2SE är de Java-applets som ofta syns på diverse hemsidor. J2EE, Java 2 Enterprise Edition, används för att utveckla serverapplikationer som används i b l a webservrar. Med J2EE kan man utveckla system som använder databaser och servrar. Här ligger tyngdpunkten på att applikationen alltid skall vara tillgänglig och att ett grafiskt användargränssnitt oftast saknas. I figur 2.1 kan man se de hårdvaror som oftast används för att exekvera J2EE respektive J2SE. Som vi ser, och har nämnt ovan, används J2EE främst till servrar och J2SE till vanliga arbetsstationer.



Figur 2.1: *Denna bild illustrerar sambandet mellan de olika Javateknologierna.*

2.2 Java 2 Micro Edition – J2ME

J2ME, Java 2 Micro Edition, är till för produkter med relativt begränsat minnesutrymme och liten processorkraft. Huvudmålet för J2ME är att specificera en plattform som kan stödja en begränsad service för ett stort antal olika apparater som har många olika funktionaliteter.

En specifik konfiguration för J2ME definierar en Javaplatform för en viss familj av apparater. Alla medlemmar i en viss familj har liknande krav vad gäller minne och processorkraft. En konfiguration är egentligen en specifikation som identifierar vilka systemfaciliteter som är tillgängliga, som t ex vilken karakteristik den virtuella maskinen har och vilka Javabibliotek som minst stöds.

Konfigurationsspecifikationen kräver att alla Javaklasser som tagits med från J2SE ser exakt likadana ut eller är en delmängd av orginalklasserna från J2SE. Detta betyder att en klass i J2ME inte får innehålla några fler metoder än vad som finns i klassen med samma klassnamn i J2SE.

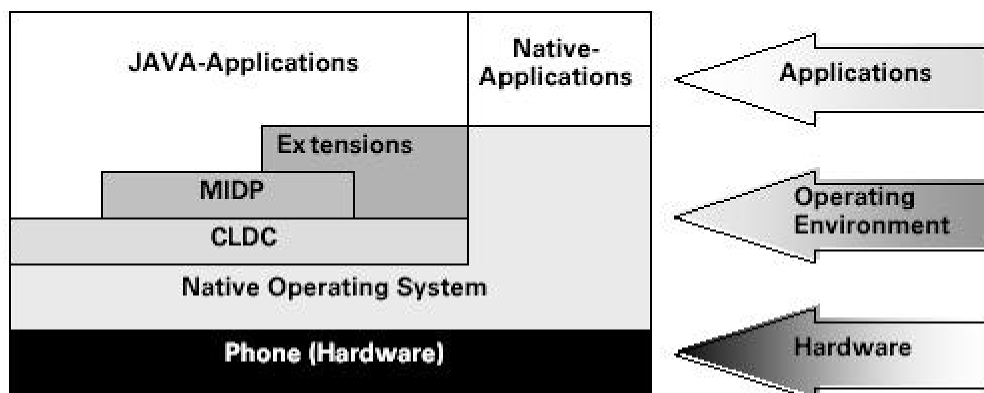
Det andra byggblocket i J2ME är profilerna. De specificerar applikationsgränssnittet för en viss klass av apparater. Exempelvis är MIDP skapad för mobiltelefoner. En profils

implementation består av en mängd klassbibliotek som specificerar applikationsgränssnittet. Huvudmålet för en profil är att garantera möjligheten att kunna flytta applikationer mellan apparater i samma kategori.

En profil är implementerad ovanpå en konfiguration. Programmeraren använder sedan profilens och konfigurationens klasser för att utveckla sin applikation. En profil innehåller klassbibliotek som är mer specifika för en viss kategori av apparater än de klassbibliotek som finns i en konfiguration. Applikationen byggs ovanpå konfigurationen och profilen. Man kan bygga flera profiler ovanpå varandra, medan konfigurationer inte får byggas ovanpå varandra. Figur 2.2 visar vilka byggblock som en MIDlet bygger på. Figur 2.3 visar hur dessa byggblock passar in i en mobiltelefon.

MIDlet
MIDP
CLDC
KVM
AMS (OS)

Figur 2.2: *Figuren visar de byggblock som behövs för att exekvera Javaapplikationer i en mobiltelefon.*



Figur 2.3: *Beskrivning av de olika delar som finns i en mobiltelefon. [43]*

2.3 Connected Limited Device Configuration – CLDC

Apparater i CLDC-kategorin har följande karakteristik:

- Mellan 160 och 512 KB minne är tillgängligt för Javaplatformen
- Apparaten använder sig av en 16 eller en 32-bitars processor
- Apparaten har liten strömåtgång. Oftast kommer energin från ett batteri
- Apparaten har möjlighet till begränsad nätverksuppkoppling

Målet för CLDC är att definiera en standard för dessa apparater. I specifikationen för CLDC görs minimalt med antaganden om miljön i vilken den existerar. CLDC specificerar vilka operationer och datatyper som måste stödjas, t ex stödjer CLDC inte flyttal, vilken funktionalitet som måste finnas i den virtuella maskinen, se avsnitt 2.5, samt vilka klassbibliotek som måste finnas.

2.4 Mobile Information Device Profile – MIDP

Eftersom kategorin av apparater som kan använda sig av CLDC är stor, krävs många olika profiler som kan stödja alla apparater. En av dessa är Mobile Information Device Profile, MIDP eller MID Profile som den ibland kallas. MIDP ligger ovanpå CLDC och definierar en mängd med gränssnitts-API:er speciellt designade för trådlösa apparater, t ex mobiltelefoner och tvåvägssökare (pagers) som ungefärligt har följande karakteristik:

- Skärmstorlek med minst 96*54 pixlar
- Display djup på 1 bit
- En- eller tvåhandskkeyboard, inmatning m h a touchscreen
- 128 KB flyktigt minne för MIDP-komponenter
- 8 KB beständigt minne för data som ska sparas mellan applikationexekveringar
- 32 KB runtimeminne för Javas heap
- Tvåvägs, trådlös uppkopplingsmöjlighet

2.5 MIDP:s virtuella maskin

Den virtuella maskinen är en mycket viktig del i alla Javamiljöer. Det är den virtuella maskinen som gör att man kan utveckla en applikation för att sedan kunna flytta applikationen till ett annat system, t ex kan man skapa en applikation för windows för att sedan kunna använda samma applikation till en dator med exempelvis linux som operativsystem. Den virtuella maskinen är ett mellanlager mellan applikationen och det underliggande operativsystemet. Den virtuella maskinen exekverar applikationens bytekod, den kod som

skapas när man kompilerar sina Java-applikationer. Denna kod läggs i en fil som har ändelsen *.class*. Den virtuella maskinen tar också hand om relaterade uppgifter som att hantera systemets minne, tillföra säkerhet mot elak kod och att hantera multipla trådar

I kraftfulla system, t ex persondatorer och servrar, används den virtuella maskinen JVM, Java Virtual Machine. När Sun skulle utveckla en Javamiljö för begränsade system, t ex handdatorer och mobiltelefoner, hade man några viktiga punkter som behövde lösas:

- Man behövde minska storleken på den virtuella maskinen och på de inbyggda klassbiblioteken.
- Man hade inte tillgång till särskilt mycket minne vid programexekveringen.
- Man ville att vissa komponenter i den virtuella maskinen skulle kunna konfigureras för att passa många maskiner. Olika mobiltelefoner kan ha olika processorer och olika tillgång till minnet.

Den nya virtuella maskinen, för bl a mobiltelefoner heter KVM, Kilobyte Virtual Machine. För att kunna uppnå de ovanstående punkterna skapade man en virtuell maskin med följande egenskaper:

- Reducerad storlek på den virtuella maskinen. KVM har mellan 50 och 80 Kb objektkod i standardkonfigurationen, beroende på målplatformen och kompileringsval.
- Minskat minnesutnyttjande. Förutom KVM:s minskade objektstorlek kräver den bara några tiotals Kb av det dynamiska minnet, vilket gör att även om det tillgängliga minnet bara är 128Kb kan användbara Javabaserade applikationer köras på mobiltelefonen.
- Ökad prestanda. Det är möjligt att köra KVM på 16-bitars processorer med en klockfrekvens så låg som 25 MHz.

2.6 Application Management Software – AMS

Alla Javaapplikationer oavsett om de är MIDlets, J2SE-applikationer eller applets körs under kontroll av den virtuella maskinen. När man kör en Javaapplikation på din dator startar man den virtuella maskinen m h a en kommandoprompt, eller om en applet körs, startas den virtuella maskinen av webbrowsern. Vad är det då som kontrollerar den virtuella maskinen på en mobiltelefon? Startande, avslutande och hantering av en J2ME-applikation kontrolleras av Application Managment Software, AMS som finns på apparaten. AMS är ansvarig för allt

från installation till uppgradering och borttagande av applikationen. AMS implementeras av apparatens tillverkare och tillhör därför mobiltelefonens operativsystem.

Källor [28], [45], [46]

3 Källkod

Detta kapitel handlar om att skriva källkod med inriktning på effektivitet. Effektivitet har många sidor. De vi har inriktat oss på i första hand är tidsåtgång och minnesoptimering. Effektivitet kan också innebära hur lång tid det tar att utveckla och hur komplex koden blir att läsa om man vill återanvända den. Flera rekommendationer i detta kapitel kommer att försvåra utvecklandet och återanvändningen av källkoden. Dessa kan dock vara nödvändiga för att programmen skall fungera på en plattform med begränsat minne och CPU-kraft som dagens mobiltelefoner.

När man skall göra en applikation bör man först göra applikationen utan att optimera den för att sen gradvis optimera den. Sen testar man om optimeringen är tillräcklig för att programmet skall fungera tillfredställande. Om programmet fortfarande är för långsamt eller kräver för mycket minne får man fortsätta att optimera. Hos referenserna som är upplistade i slutet av varje avsnitt finns information om det som tas upp i avsnittet. Hos de referenser som står inuti ett avsnitt finns information som endast tar upp den specifika delen.

3.1 Testmetoder

För att mäta hur mycket minne olika programmeringssätt upptar har vi använt den fördefinierade klassen *Runtime*[39] och dess metod *freeMemory*. *freeMemory* returnerar det lediga minnet på heapen. Vi har anropat denna metod direkt före och efter testkoden, för att sedan jämföra resultatet och se hur mycket minne som gått åt. Här följer ett exempel:

```
int before = runtime.freeMemory();
// Gör minnnetest
int after = runtime.freeMemory();
// Jämför before och after
```

För att mäta hur lång tid olika programmeringssätt tar har vi använt den fördefinierade klassen Date och dess metod *getTime*. *getTime* returnerar antalet millisekunder sedan 1 januari, 1970, klockan 00:00:00. Tiderna i tabellerna är alltså mätta i millisekunder. Vi har

anropat denna metod direkt före och efter testkoden för att se hur mycket tid som har gått åt. Vi kan inte garantera att skräpsammlaren, vilken är en automatisk avallokerare som letar efter orefffererade objekt och avallokerar deras minne, körs av den virtuella maskinen under testkörningen, men eftersom alla testkörningar ger liknande resultat är sannolikheten väldigt liten. För att minska sannolikheten ytterligare anropas skräpsammlaren explicit innan varje test m h a metoden *gc*, som finns i klassen *System*.

3.2 Objekt

När vi undersöker objekt gör vi det med avseende på att effektivisera objekthanteringen i Java för att koden skall exekveras snabbare men också för att spara på det dynamiska minnet, heapen. Alla objekt som skapas m h a nyckelordet *new* skapas på heapen. Det andra minnesutrymmet som finns är stacken. I stacken lagras alla lokala variabler som inte skapats m h a nyckelordet *new*. Slutligen finns det statiska minnet. Här lagras alla konstanter som skapas m h a nyckelordet *final*.

3.2.1 Rekommendationer

Använd få temporära objekt, eftersom varje objekt som skapas tar upp extra minne som inte tas bort direkt av skräpsammlaren. Dessutom tar skapandet av objekt en viss tid. Använd metoder som ändrar objekt direkt istället för att göra kopior.

Det är bra om man gör konstruktorerna små för att minska tiden för skapandet av objektet. Det går då också snabbare för skräpsamlaren. Skräpsamlaren sköts normalt av den virtuella maskinen men kan också anropas manuellt. Använd inte för djupa arvshierarkier. Om man kan undvika att ärva i flera generationer är det bra, eftersom varje förälders konstruktor anropas vid konstruktionen av ett objekt.

Det kan också vara smart att minska flaskhalsar genom att skapa objekt vid bra tillfällen, t ex innan man startar eventuella trådar som kan ta upp cpu-tid, eller när processorn har lite att beräkna. När applikationen väntar på händelser från användaren är ofta ett bra tillfälle att skapa objekt, eftersom det alltid blir fördröjningar när människan skall interagera med applikationen. Då kan det också vara bra att explicit anropa skräpsammlaren.

Alla objekt som skapas har en negativ effekt på applikationens exekveringstid såväl som på minnesutrymmet. Om man inte behöver skapa ett objekt är det bättre att inte göra det, t ex är:

```
public void setSize(int width, int height);
```

bättre att använda än:

```
public void setSize(Dimension  size);
```

eftersom det, i andra fallet skapas ett onödigt objekt av klassen *Dimension*.

Använd tekniken *Lazy Instantiation* genom att bara instansiera objekten när de behövs. Tekniken går ut på att man letar efter nollreferenser. Det görs på det här sättet:

```
public Vector getVector()
{
        if(v == null)
        {
                v = new Vector();
        }
        return v;
}
```

Vektorn skapas bara om det inte tidigare finns en vektor.

Lita inte på skräpsammlaren. Om man allokerar för många objekt för snabbt, kan skräpsammlaren ha problem att hinna med att avallokera orefererade objekt. Ett bra sätt att hjälpa skräpsammlaren är att sätta objektreferensen till *null* när man är klar med objektet. Då ser skräpsamlaren lättare den referens som tidigare refererade till det aktuella objektet inte längre behöver objektet. Anropa skräpsammlaren manuellt, genom metoden *Runtime.gc()*, för att vara säker på att man har tillräckligt med fritt minne i heapen när man behöver det, men också för att den inte skall sättas igång när man är inne i en beräkningsintensiv period eftersom systemet då kan få mycket att göra och applikationen blir långsammare.

Om man enbart skall anropa statiska metoder i en klass bör man aldrig skapa ett objekt av klassen utan alltid anropa metoden med hjälp av klassnamnet. Detta fungerar eftersom statiska variabler och metoder är gemensamma för alla objekt av klassen, vilket betyder att man inte behöver skapa ett objekt av klassen. Det är alltså bättre att skriva:

```
MyClass.staticFunction();
```

än att skriva:

```
MyClass myClass = new MyClass();
myClass.staticFunction();
```

Alla metoder som är statiska variabler har kortare accesstid än de dynamiska.

Källor: [5], [14], [21], [22], [26], [29], [30], [31], [37]

3.2.2 Analys

När det gäller att minska på användandet av arvshierarkier finns nackdelen att det förstör en del av objektorienteringen i Java. En av idéerna med objektorientering är dock att det skall finnas ett visst släktskap mellan liknande objekt, t ex om man har klasserna *fordon* och *bil*, låter man bil ärva från fordon, dels för att man ska kunna återanvända kod men också för att man skall kunna visa på ett släktskap mellan klasserna. Avväg vad som är viktigast, att applikationen är välstrukturerad och bra uppdelad i klasser, eller att optimera koden.

När man vill anropa skräpsammlaren manuellt måste man avväga hur ofta det kan göras. Om man anropar skräpsammlaren ofta kan exekveringen saktas ner. Om man alls inte anropar skräpsammlaren finns det dock risk att skräpsammlaren måste köras då applikationen är inne i en cpukrävande del.

Tester som skulle kunna göras under utvecklandet av enskilda applikationer är att man testar hur ofta man kan anropa skräpsammlaren utan att exekveringen saktas ned.

3.3 Klasser

I det här kapitlet undersöker vi klasser i Java och deras egenskaper, med avseende på effektivitet och objektorientering.

3.3.1 Rekommendationer

Gör om generella klasser, d v s klasser som är definierade i Javas J2ME-API så att de är specifika för de objekt som applikationen använder. Exempel på sådana klasser är *Hashtable*, *Stack* och *Vector* som finns i paketet *java.util*. Generella klasser är normalt mycket långsammare än specialiserade klasser. Ett skäl är att de inte är optimerade för någon speciell typ. I generella klasser lagras objekt av typen *Object*. Eftersom alla objekt ärver denna klass kan man lagra alla typer av objekt. Om man lagrar ett objekt av typen *MyObject* i ett generellt objekt, t ex av typen *Vector*, lagras det som om det vore ett objekt av typen *Object*. När man vill använda sitt lagrade objekt måste man omvandla objektet till den aktuella typen.

Undvik inre klasser. När man kompilerar en Java-applikation skapas en egen fil för varje klass, inkluderat inre klasser, vilket leder till att mer minne krävs. Därför är det bättre att använda få större klasser än många små. Om man bara vill använda en liten del av funktionaliteten från en klass får man avväga vad man tjänar mest minne på, att skapa en stor klass eller dela upp klassen i fler klasser.

Källor: [13], [29], [38]

3.3.2 Analys

Eftersom Java är objektorienterat är det meningen att man skall dela upp applikationerna i flera klasser. För vanliga persondatorer är detta inget problem, men när man programmerar små enheter med mycket begränsad processor- och minneskapacitet måste man ibland frångå idén med objektorientering för att effektivisera koden. Detta är ett sådant tillfälle. Eftersom overheaden för att skapa objekt utifrån klasser är stor är det ofta nödvändigt att göra avsteg från objektorienteringen för att anpassa sig till de begränsade resurserna.

Att skapa specialiserade klasser ökar oftast exekveringshastigheten på applikationen. Problemet är att man då får en extra klass som tar upp minne. Om man skapar applikationen till en speciell mobiltelefonsort kan man ta reda på dess begränsningar och då lättare bestämma sig för vad som är viktigast att optimera.

3.4 Primitiva typer

I MIDP-profilen[39] finns inte alla primitiva datatyper som finns i J2SE[40]. De som finns är *boolean*, *byte*, *char*, *int*, *long* och *short*. Här undersöker vi de primitiva datatyperna med avseende på både effektivitet och objektorientering.

3.4.1 Rekommendationer

Utnyttja de primitiva datatyperna. Det är bättre att använda de primitiva typerna, t ex *int*, om det är möjligt istället för att använda deras objektmotsvarigheter, t ex *Integer*, eftersom primitiva typer inte kräver lika mycket minne och har snabbare access än vad objekt har. Det är alltså bättre att skriva

```
int i = 10;
```

istället för

```
Integer i = new Interger(10);
```

När man ska använda primitiva typer i en loop är *int* alltid att föredra. Processorn har speciella operatorer för att operera på *int*. Detta gör att det går snabbare att operera på *int* än t ex *byte*, *short* och *long*.

Källor: [15], [31]

3.4.2 Analys

Det enda skälet till att man ska använda objektet *Integer* istället för den primitiva datatypen *int* är om man vill att värdet på variabeln ska kunna anta ett nytt värde när metoden returnerat. Värdet på variabler, primitiva typer, som skickas med som argument kan inte ändras i den anropande funktionen eftersom det är en kopia av variabeln som skickas istället för variabeln själv. Skickar man med objekt kan man dock ändra i dessa eftersom man skickar en kopia på den referens som pekar på objektet.

3.4.3 Tester

Vi har testat hur mycket längre tid det tar att komma åt ett värde i ett objekt av klassen Integer än ett värde i en lokal variabel av typen *int*. Detta test gick till så att vi anropade klassen *Integers getValue*-metod, vilken returnerar *int*-värdet i objektet, och tilldelade detta värde till en lokal variabel. Detta gjorde vi i en loop som gick 10 000 varv. Sedan gjorde vi samma sak med en *int*, alltså tilldelade en lokal variabel en annan lokal variabels värde, i en loop som gick 10 000 varv. Innan och efter varje loop sparade vi tiden i millisekunder m h a Date.getTime(). Vi subtraherade de båda tiderna och fick fram hur lång tid loopen tagit.

Vi testade också hur mycket plats i minnet som de olika datatyperna och ett objekt av klassen Integer tog. Det gjorde vi genom att först anropa Runtime.freeMemory(). Sedan skapade vi ett nytt objekt och anropade Runtime.freeMemory() igen. Genom att subtrahera de båda minnesvärdena med varandra fick vi ut hur mycket minne varje objekt behövde.
Testet utfördes med Motorolas Accompli008, [25].

Försök	1	2	3	4	5	6	7	8	9	10	medelvärde
Int	389	385	388	389	390	390	389	390	389	389	388,8
Integer	425	400	427	426	400	400	426	401	405	400	411

Tabell 3.1*: Exekveringstid för int.*

I tabell 3.1 ovan visas hur mycket längre exekveringstid den primitiva typen *int* tar upp kontra klassen *Integer.* Värden visas i antal millisekunder.

Typ	byte	short	int	long	Integer
Antal byte	1	2	4	8	16

Tabell 3.2*: Primitiva typers minnesutnytjande.*

I tabell 3.2 visas hur mycket minne olika primitiva typer tar upp. Värden visas i antal byte.

Vi ser att skillnaden i tid var ungefär 5 % (från tabell 3.1) vilket inte är en jättestor skillnad, men att skillnaden i minnesutrymme (tabell 3.2) är betydligt större. Det är inget stort avbräck på objektorienteringen att använda primitiva typer istället för objekt. I o m att en wrapperklass, en klass som är till för att omsluta en primitiv typ, tar både mer minne och längre tid att använda, rekommenderas att använda primitiva typer framför deras respektive wrapperklasser. Det kan dock vara bra att testa hur det förhåller sig med andra wrapperklasser jämfört med deras primitiva motsvarigheter (t ex Char och char, Byte och byte etc.).

3.5 Omvandlingar

Här tittar vi på omvandling av objekt, på engelska kallat *cast*, med avseende på effektivitet.

3.5.1 Rekommendationer

Undvik omvandlingar genom att använda en variabel av rätt typ. Det är bättre att skriva:

```
MyObject myObject = (MyObject)Vector.elementAt(1);
myObject.metod();
myObject.metod2();
```

istället för:

```
Object myObject = Vector.elementAt(1);
((MyObject)myObject).metod();
((MyObject)myObject).metod2();
```

eftersom man slipper omvandla objekten flera gånger. Att omvandla objekt och variabler är långsamt, eftersom kompilatorn testar så att det går att omvandla objektet från den gamla typen till den nya. Det tar ännu längre tid att omvandla ett objekt till ett *interface* än till andra objekt. När man omvandlar ett objekt till ett interface måste kompilatorn exekvera mer kod än om man skulle omvandla objektet från en klass till en annan klass. Alla objekt ärver från klassen *Object* vilket gör att det går enklare att omvandla ett objekt från en klass till en annan. Interface ärver inte från klassen objekt vilket gör att det är svårare för kompilatorn att omvandla mellan klasser och interface.

T ex om man har:

```
Interface MyInterface {}
class MyClass {}
class MySubClass extends MyClass implements MyInterface {}
```

Det tar betydligt längre tid att omvandla så här:

```
MyClass myClass = new MySubClass();
MyInterface myInterface  = (MyInterface) myClass;
```

än att omvandla *myClass* till ett annat objekt.

För längre arvshierarkier mellan ursprungsklassen och den nya klassen tar det ännu längre tid för omvandlingen eftersom man då måste gå i fler steg innan man kan se om det går att omvandla från klassen till interfacet.

Källor: [5], [33], [36]

3.5.2 Analys

Undvik omvandlingar. Kan man använda objektreferenser av rätt typ direkt finns det inte någon anledning att använda sig av omvandlingar.

3.6 Inlining

Inlining betyder att kompilatorn byter ut metodanrop till metoder som utför lite kod, oftast getXXX och setXXX-metoder som hämtar/sätter en privat variabel, mot koden för metoden. Detta sker vid kompileringen. På så sätt slipper man göra onödiga metodanrop. Inlining kommer från C++ där det finns ett nyckelord som heter inline.

3.6.1 Rekommendationer

Använd inlining så ofta som möjligt. Detta kan göras med hjälp av nyckelordet *private*. Alla metoder i klassen som är deklarerade m h a nyckelordet *private* är kandidater till inlining. Detta p g a att de inte kan anropas utifrån av andra klasser. Om man deklarerar en klass till *final* kommer kompilatorn också att kunna göra vissa optimeringar, eftersom man inte kan ärva sådana klasser. Beroende på vilken kompilator man använder kan inliningen ske automatiskt, eller så får man ange att inlining ska utföras, m h a en flagga. För att göra denna optimering i Suns™ standardkompilator[42] skriver man javac –O MyClass.java. Ett exempel på inlining, om man själv skriver så här:

```
MyClass myClass;
...
int i = myClass.getTheValue();
```

så ersätter kompilatorn koden så att det står så här:

```
MyClass myClass;
...
```

```
int i = myClass.variable;
```

så här är metoden implementerad:

```
getTheValue()
{
        return variable;
}
```

Källor: [4]

3.6.2 Analys

Inlining bör man aldrig göra själv, eftersom det förstör objektorienteringen. Om man vill göra inlining själv ersätter man funktioner, t ex getVariable med variablen själv. Variabeln måste då göras till public för att kunna användas av andra klasser. Detta är dock inte rekommenderat, då en av grundidéerna i objektorienteringen är att man skall kapsla in data.

Ett test som kan göras är att jämföra hur mycket snabbare det blir med att låta kompilatorn optimera koden.

3.7 Generella lagringsklasser

I Java finns det ett antal generella lagringsklasser som lagrar objekt av typen *Object*. Klassen *Object* är en klass som alla andra klasser automatiskt ärver ifrån. Det gör att man kan utnyttja detta till att lagra objekt av alla klasser genom att skapa en referens av typen *Object* som kan referera till alla objekt. Detta ser ut så här:

```
Object objekt;
```

I detta avsnitt undersöker vi generella samlingar med avseende på effektivitet.

3.7.1 Rekommendationer

Förbestämning av storleken av vektorer och andra mängder ökar accesshastigheten för lagringsklasserna, eftersom de i MIDP[39] är implementerade som arrayer. När man skapar ett objekt av en generell lagringsklass, t ex en vektor blir storleken initierad till noll. När man lägger in ett objekt i vektorn måste vektorn skapa en ny array av någon blockstorlek. Den metod som returnerar antalet element returnerar 1. Lägger man till element utöver den nya arraystorleken måste vektorn skapa en ny array och kopiera över alla värden till den nya arrayen. Den gamla arrayen måste sedan tas om hand av skräpsammlaren.

Om man från början skapar en vektor med en större storlek uppstår inte detta problem lika snabbt. Klassen *Vector* kan man återanvända genom att använda metoden *.removeAllElements()*. Undvik generella mängdklasser, som t ex *Vector* och *Stack*. Använd istället mängder som är skapade för just den objekttypen. Nackdelen med detta förslag är att man då måste skapa ytterligare en klass, istället för att använda sig av standard biblioteket.

Ytterligare ett skäl att inte använda klassen *Vector* är att vissa metoder är synkroniserade vilket minskar exekveringshastigheten för metodanropen avsevärt. Om objektet av en lagringsklass är väldigt stor är det bättre att skapa ett nytt objekt istället för att använda klassens inbyggda ”töm alla element”-metod. Detta på grund av att det tar längre tid att rensa alla objekt ur tabellen än det tar att skapa en ny tom tabell.

Källor: [29], [30],[37]

3.7.2 Analys

När man ska lagra objekt måste man avväga vad som är viktigast för den applikation man utvecklar. Fördelen med att skapa egna lagringsklasser är att man kan optimera metoderna, t ex en sorteringsfunktion, för att få snabbare exekvering. Problemet är att för varje extra klass man skapar krävs extra minne för klassfilerna. När man utvecklar en applikation bör man först testa programmet om det blir tillräckligt snabbt med generella klasser. Om man tycker att det går långsamt och applikationen inte är för stor för den tänkta målplattformen kan man göra optimerade lagringsklasser.

3.8 Loopar

Här undersöker vi loopar och hur dessa kan effektiviseras.

3.8.1 Rekommendationer

Försök att undvika upprepade metodanrop och skapande av objekt inuti en loop. Det ar bättre att skapa objektet utanför loopen istället, eftersom man då slipper skapa nya objekt för varje varv i loopen.

Det är alltså bättre att skriva:

```
StringBuffer sb = new StringBuffer();
for (int i = n; i >= 0; i--)
{
        // gör något med sb.
```

```
}
```

än att skriva:

```
for (int i = n; i >= 0; i--)
{
        StringBuffer sb = new StringBuffer();
        // gör något med sb.
}
```

Om man inte använder loopvariabeln inuti loopen kan man göra själva loopen effektvare genom att jämföra med noll istället för t ex *size()* eller någon annan siffra än noll. Det finns oftast inbyggda operationer i hårdvaran som snabbt jämför med noll. Man bör alltid använda datatypen *int* som varvräknare eftersom det är den datatyp som processorn snabbast kan öka/minska värde. Det kan se ut på det här sättet:

```
for(int i=n;  i>=0;  i--)
```

Om man vill iterera genom en array kan man använda en alternativ metod:

```
try
{
        for(i=0; ;  i++)
        {
                variable = array[i];
        }
}catch(ArrayIndexOutOfBoundsException){}
```

Detta fungerar p g a att varje access i en array automatiskt kontrolleras så att inte indexet är felaktigt. D v s Javas virtuella maskin undersöker alltid om ett index ligger inom arrayens storlek. Om indexvärdet är större än vad arrayen har element eller om indexvärdet är mindre än noll kastas ett undantag. Det här alternativet skall bara användas om man har stora *for*-satser, eftersom det är kostsamt att kasta undantag.

I loopar skall endast nödvändig kod stå, det som kan göras utanför ska göras utanför. Det är också bra att göra testerna i loopen så små som möjligt. När man använder sig av variabler i en loop är det bättre att använda lokala variabler än t ex klassvariabler.

Om man använder sig av klassvariabler i en loop kan det vara en bra ide att lagra värdet lokalt i metoden och sedan använda den lokala variabeln. Det kan också vara bra att använda en lokal variabel vid användning av en array. Det är bättre att skriva:

```
int length = buf.length;
```

```
for(int=0; i <length; ++i)
{
          char ch = buf[i];
          if( ch >='0' && ch <= '9' )
                    {…}
          else if( ch == '\r' || ch == '\n' )
                    {…}
}
```

än:

```
for(int i=0; i<buf.length, i++)
{
          if( buf[i] >='0' && buf[i] <= '9' )
                    {…}
          else if( buf[i] == '\r' || buf[i] == '\n' )
                    {…}
}
```

Källor: [5], [29], [32], [37]

3.8.2 Analys

Att skapa objekt utanför loopen bör man alltid göra, även när man programmerar J2SE-applikationer eller om man programmerar i t ex C++. Problemet med att jämföra med 0, är att det kan vara svårt att göra detta om man måste använda sig av variabeln inuti loopen, t ex om man vill iterera genom en lista från första till sista elementet. Ibland har det inte någon betydelse i vilken ordning man går igenom positionerna i t ex en array. Då är det utmärkt att använda sig av denna teknik. Om man t ex ska sortera en array spelar det inte någon roll från vilken ända man börjar. Om man vill använda sig av undantag för att avbryta loopen måste man ha en ganska stor *for*-sats. Exakt hur stor *for*-satsen bör vara kan man ta reda på genom att utföra ett test. Oberoende av vilken version av Java man använder bör man alltid använda sig av lokala variabler för att iterera genom en loop.

3.8.3 Tester

Vi ville testa om det tog kortare tid att testa mot noll i en loop jämfört med att testa mot ett annat tal. Först kontrollerade vi tiden innan loopen med metoden *Date.getTime.* Sedan fick

loopen snurra 10000 gånger. Loopen hade inget innehåll. När loopen var klar mätte vi tiden igen. Vi testade både med att jämföra med noll och med 10000.

Test nr	1	2	3	4	5
Jämförelse med 0 (ms)	498	507	498	508	494
Jämförelse med 10000 (ms)	678	651	697	687	683

Tabell 3.3*: Tidsåtgång vid jämförelse.*

Tabellen ovan visar hur stor skillnad det är att jämföra med 0 och med 10000.

Vi får från tabell 3.3 att det är drygt 20 % skillnad i tid mellan att jämföra med noll och att jämföra med 10000. Det är alltså en relativt stor tidsvinst att jämföra med noll. Testen har utförts på Siemens SL 45i, [34].

3.9 Metodanrop

Här undersöker vi hur man kan effektivisera metodanrop.

3.9.1 Rekommendationer

Man kan undvika onödiga metodanrop genom att göra variabler direkttillgängliga. Man gör då om variablerna så att de blir publika. Använd bara denna metod om inte kompilatorn använder sig av inlining. Då kan man skriva:

```
public myClass.variable = 1;
```

istället för

```
myClass.setVariable(1);
```

om det enda funktionen gör är att sätta den privata variabeln till det värde som argumentet har.

Metoder som inte skall anropas utifrån bör alltid deklareras som private så att man lätt kan döpa om metoden utan att riskera att andra klienter har anropat den. Om en metod kan ta emot argument, som t ex ett objekt, och om objektet inte skall förändras bör det deklareras till *final*. Detta gör man för att kompilatorn skall kunna optimera. Metoder som är deklarerade till *final* är inte virtuella och kräver därför inte lika mycket overhead som de virtuella metoderna. Virtuella metodanrop är långsammare än ickevirtuella metodanrop eftersom virtuella

metodanrop inte binds till rätt metod vid kompileringen vilket de ickevirtuella gör. Att det är så beror på att kompilatorn inte vet vid kompileringstillfället exakt vilken funktion som ska anropas vid arv. Det beror på vilken typ av objekt funktionen tillhör.

Källor: [5], [32], [37]

3.9.2 Analys

Att ha direktaccess till variablerna är ett allvarligt brott mot objektorienteringen och bör bara användas som en sista utväg om applikationen fortfarande inte är tillräckligt snabb och den kompilator man använder inte använder sig av inlining.

3.10 Undantag

Undantag är till för att hantera fel som kan uppkomma vid exekveringen av applikationen, t ex vid filinläsning eller kommunikation över nätverk. I J2SE använder ofta programmeraren undantag för att inte behöva returnera om en operation lyckas eller inte. När vi skriver om undantag vill vi sätta fokus på att applikationen skall kunna ha en hög exekveringshastighet.

3.10.1 Rekommendationer

I MIDP-applikationer bör undantag inte användas om det är möjligt att göra på något annat sätt. T ex, om man implementerar en stack bör man alltid först anropa en metod som visar om stacken är tom innan man försöker hämta ett element. Om man använder sig av undantag kan man försöka hämta elementet direkt. Om det inte finns något element i stacken kastas ett undantag. Genom att undvika undantag kan man reducera klassfilernas storlek samt antalet objekt som allokeras eftersom att kasta ett undantag innebär att ett undantagsobjekt kastas. Objektskapandet gör att det tar lite längre tid att använda undantag.

Källor: [13], [21]

3.10.2 Analys

Om man programmerar med starka kontrakt, som beskrivs i avsnitt 3.18, är det lättare att undvika undantag. Om man skall spara eller accessa information m h a RMS[39] i mobiltelefonens minne eller om man gör uppkopplingar mot en server måste man använda sig av undantag för att ta hand om fel som kan uppstå. T ex kan förbindelsen brytas mellan mobiltelefonen och servern eller kan minnet ta slut när man försöker skriva ner data till en fil.

3.11 Inbyggda programsatser och operatorer

I Java, och i många andra programspråk, finns det oftast flera olika sätt att göra samma sak. I detta avsnitt tittar vi på några olika sätt att utföra operationer och vilka programsatser som tar minst tid att exekvera.

3.11.1 Rekommendationer

Det går snabbare att använda en switch-sats än if-else satser. Den snabbaste operatorn av alla är *int*-inkrementeraren, e x *int i; i++*, vilket leder till att int alltid är att föredra framför t ex short, char och long i loopar etc. Operatorn a += b är snabbare att skriva a = a + b eftersom det är snabbare med en operation, +=, än två, + och =. Om man vill utföra multiplikation eller division med tal på formen 2^n kan man använda de mycket snabbare shiftoperatorerna <<, >>. Shiftoperatorerna opererar på bitnivå.

Ex: Om man har talet 10, som binärt är 1010 och vill shifta det ett steg åt vänster shiftas bitarna och en nolla läggs till i slutet, 10100. Översatt till det decimala talsystemet blir de 20. Man har alltså utfört en operation som är likvärdig med att multiplicera med två. Om man vill multiplicera med 4, som är 2^2 shiftar man talet 2 steg åt vänster.

Källor: [5], [10], [13], [15], [22]

3.11.2 Analys

Har man fler än en if-elsesats bör man byta ut den mot en switchsats om möjligt. Man bör också använda så få operatorer som möjligt. När det gäller shiftoperatorerna bör man inte använda högershiftoperatorn för att öka exekveringshastigheten vid division. När man shiftar åt höger tappar man information från den minst betydande biten. Om man shiftar åt vänster så tappas ingen information. Det enda sätt man kan tappa information med vänstershift är om det nya värdet är för stort för att lagras i den aktuella datatypen, men den informationen skulle även förloras om man använder sig av multiplikationsoperatorn. Eftersom shiftoperatorerna bara kan användas för vissa speciella tal ska man undvika att använda sig av denna operation om det inte blir väldig skillnad i tid vid exekveringen. Om man tycker att applikationen är för långsam kan man alltid testa hur stor skillnad det blir.

3.11.3 Tester

Vi har testat om det går snabbast att välja rätt programväg med switch eller med if-else. För att ta tiden använde vi Date.getTime(). Vi skrev en switch-sats med tre olika val och en if-

else med tre olika val. Sen lät vi programmet välja väg nummer 2. Vi gjorde detta i en loop som snurrade 1000 gånger.

Test nr	1	2	3	4	5	6
Switch	148	138	143	148	157	147
If else	217	216	217	213	240	217

***Tabell 3.4**: Exekveringstid för en switchsats och en if-elsesats.*

Tabellen visar skillnaden i exekveringstid för en switchsats och en if-elsesats. Alla tider är i millisekunder.

Från tabell 3.4 får vi att det är betydligt snabbare att använda switch än att använda if-else, cirka 30%. Använd alltså switch om det är möjligt. Testen har utförts på Siemens SL 45i, [34].

3.12 Beräkningar

Här går vi igenom hur man kan effektivisera beräkningar i Java.

3.12.1 Rekommendationer

Beräkna så få gånger som möjligt. Det är exempelvis bättre att skriva:

```
Double depth=d*(lim/max);
double x = depth * sx;
double y = depth * sy;
```

istället för:

```
double x = d * (lim / max) * sx;
double y = d * (lim / max) * sy;
```

Källor: [5]

3.12.2 Analys

Om samma beräkning görs flera gånger och variablernas värde inte skiljer sig åt mellan gångerna bör man lagra resultatet i en lokal variabel och använda variabeln istället.

3.13 Matriser, Arrayer och Länkade Listor

I detta avsnitt undersöker vi hur man på ett effektivt sätt kan lagra information.

3.13.1 Rekommendationer

Matriser ger mer overhead och har längre åtkomsttid än arrayer. Använd därför en array istället för en matris. Om man har en matris, int matris[a][b] innebär det att man har en array med a stycken element. Varje element är i sin tur en array av b stycken *integers*. Om b är ett stort tal blir den procentuella skillnaden mindre än om b är liten eftersom det extra minnesutrymmet som krävs för att peka på en array blir en mindre del av det sammanlagda minnesutrymmet som upptas av hela matrisen. Om man vill använda sig av små matriser är det bättre att skapa en array och istället simulera matrisen, t ex genom att abstrahera matrisen m h a en metod. Man kan då göra en metod som ser ut så här:

```
int getElement(int rad, int kolumn)
{
    return array[rad+kolumn*STORLEKENPAENRAD];
}
```

Länkade listor bör man inte använda alls, dels eftersom de skapar objekt och dels för att de är långsamma. Om man vet ungefär hur många platser man behöver i en lagringsklass är det klart bättre att använda sig av arrayer än av listor. Listor bör man bara använda om man inte vet hur många element som ska lagras och om man inte utför många accessoperationer. En accessoperation är t ex att man anropar metoden *getElement*.

Använd inte heller rekursion. Det är bättre att använda en loop om detta är möjligt. Att använda rekursion tar längre tid och mera minne eftersom varje gång ett anrop görs måste nytt minne allokeras på stacken. Det tar 8 gånger längre tid att använda rekursion och varje metodanrop allokerar ca 40 bytes på stacken. Vissa kompilatorer ersätter rekursion med loopar. Använder man en sådan kompilator behöver man inte tänka på detta råd.

Källor: [1], [30]

3.13.2 Analys

Om man vill implementera en matris, *matris[a][b]* där *a* är ett stort tal och *b* är ett mindre tal är det en fördel att implementera matrisen som en array eftersom overheaden tar upp en större del av minnesutrymmet än om förhållandet vore tvärtom. Om man har små matriser är

det bättre att implementera den som en vanlig matris eftersom varje metodanrop också har en overhead som gör att man inte vinner något på att implementera den som en array.

När det gäller länkade listor kontra arrayer är det generellt bättre att implementera lagringsutrymmet som en array om man ofta behöver utföra accessoperationer eftersom arrayer har kortare accesstid, tack vare direktaccess till varje position, än vad länkade listor har. Om man skall lägga till väldigt många element i en lista kan det dock löna sig att implementera den som en länkad lista. När det gäller enskilda mobiltelefoner kan det dock hända att man behöver anpassa sig på grund av dess specifika svagheter/styrkor. T ex kan en viss mobiltelefon ha en snabb processor men väldigt lite minne vilket kan leda till att man måste ändra på applikationen. Detta får man testa på den mobiltelefon som man huvudsakligen utvecklar applikationen till.

Att inte använda rekursion kan innebära att koden blir lite mer komplex, t ex vid trädtraversering. Men med dagens mobiltelefoner så måste man ändå undvika rekursion om inte kompilatorn kan optimera detta.

3.13.3 Tester

Vi har testat åtkomsttider i en array, en matris och en lokal variabel. Vi mätte tiden med *Date.getTime()*. Vi gjorde en loop där vi varje varv hämtade ett värde i matrisen och lade det i en lokal variabel. För varje varv hämtade vi värdet från olika positioner. Vi lät loopen snurra i 10000 varv. Sedan gjorde vi samma sak med en array och en lokal variabel. Datatypen vi använde i försöket var *int*. Detta test utfördes på en Motorola Accompli008 (se [25] för information om telefonen).

Vi testade också hur mycket minne en array tar upp och hur mycket minne en matris, med samma antal platser som arrayen, tar upp. Detta test utfördes på en Siemens SL45i, [33].

Försök	1	2	3	4	5	6	7	8	9	10	medelvärde
int	389	385	388	389	390	390	389	390	389	389	388,8
Array	414	417	414	415	410	415	414	413	415	415	414,2
Matris	489	515	489	493	514	489	491	514	515	514	502,3

Tabell 3.5*: Accesstid för int, int-array och int-matris.*

Tabell 3.5 visar accesstiden för en *int*, en *int*-array och en *int*-matris mätt i millisekunder.

Typ	Tar upp (bytes)
int[1]	20
Int[1][1]	60
Int[100]	416
Int[10][10]	640
Testet kördes 10 ggr med samma resultat	

Tabell 3.6*: Arrayers och matrisers minnesutnyttjande.*

Tabell 3.6 visar hur mycket minne arrayer och matriser tar upp. Värdena är mätta i bytes.

Från tabell 3.5 och 3.6 får vi att det både är snabbare och tar mindre plats i minnet att använda arrayer istället för matriser. Tidsåtgången är ca 17 % mindre med arrayer. Hur mycket minne man tjänar i procent beror på hur stor b i int[a][b] är. Alltså, om b är liten tjänar man mer. Då kan man med fördel använda en lång array istället för en matris.

3.14 CharArrayer, SträngBuffertar och Strängar

Detta avsnitt handlar om hantering av strängar och hur man kan göra för att effektivisera dessa.

3.14.1 Rekommendationer

För kopiering av arrayer från t ex en *char*-array till en annan *char*-array bör man använda metoden *arraycopy( )* eftersom denna funktion är den effektivaste när det gäller att kopiera arrayer, som finns i klassen *System.* Om man ska ändra i en sträng bör man alltid använda sig av klassen *StringBuffer*, som huvudsakligen är till för att lagra föränderliga textsträngar, istället för klassen *String*, som lagrar strängar som inte kan ändras. Bäst av allt är om man kan använda *char[ ]* istället, eftersom det är ännu effektivare än *StringBuffer*. Man bör undvika all onödig strängmanipulering eftersom det kan vara svårt att veta när nya strängobjekt skapas. T ex när man vill sätta ihop strängobjekt m h a + operatorn, skapas flera onödiga objekt. Som exempel, om man vill sätta ihop flera strängar:

```
String x = "my" + "String";
```

så motsvarar det detta:

```
x = new Stringbuffer().append("my")
  .append("String").toString();
```

Vid en strängkonkatenering skapas först ett nytt *StringBuffer*-objekt, objektets *append*metod anropas, och tillslut anropas metoden *toString* som skickar tillbaks ett nytt *String*-objekt. Det är därför bättre att använda sig av *StringBuffer*. Vid skapande av en *StringBuffer* kan man skicka med den förväntade storleken, detta gör att *StringBuffer*-objektet inte behöver expanderas. När man skickar med en förväntad storlek skapas en *StringBuffer* direkt med den storleken. Om man vill lagra en sträng med större storlek, skapas ett nytt objekt med större storlek. Den gamla strängen kopieras över och blir till slut avrefererad, så att den kan tas omhand av skräpsamlaren. Om man skapar ett *StringBuffer*-objekt utan storlek kommer man att kasta objekt tidigare vilket leder till att mer minne blockeras i väntan på att skräpsamlaren hinner avallokera objektets minne.

Källor: [5], [15], [21],[22],[29],[37], [40]

3.14.2 Analys

Det effektivaste sättet att lagra strängar är att lagra dem i arrayer. Nackdelen med detta är att de då tolkas mer som en array av enskilda tecken än en textsträng. Normalt är det bäst att använda klasserna *String* och *StringBuffer* för att lagra strängar eftersom de konceptuellt representerar strängar. Det extra minne som det kostar är oftast värt att använda för att få en tydligare förståelse av källkoden.

3.15 Variabler

I detta avsnitt kommer vi att gå igenom olika variabeltyper och hur man kan göra för att minska accesstiderna till dessa.

3.15.1 Rekommendationer

När variabler instansieras sätts de implicit till defaultvärden, t ex *int i*, sätts till 0. Om man också initierar variablerna explicit kommer kompilatorn att generera mer bytekod samt att det tar längre tid eftersom den gör initieringen två gånger. Tabell 3.7 visar vilka defaultvärden olika typer får.

Variabel typ	Initierade värden
Objekt	Null
boolean	False
Byte, short, int long	0
char	Null

Tabell 3.7: *Defaultvärden.*

Använd lokala variabler, det tar generellt längre tid att accessa klassmedlemmar än att accessa lokala variabler, eftersom klassvariabler lagras i heapen till skillnad från lokala variabler som lagras i stacken. Läs mer om variabler i loopar i avsnitt 3.8.

När man skall använda arrayer av heltal kan man spara minne. Om man t ex bara ska spara resultat mellan 1 och 10, är det bättre att använda byte än short, int och long.

Källor: [5], [10], [15], [21], [23], [29]

3.15.2 Analys

Det är generellt god programmeringsstil att initiera variabler explicit, men detta rekommenderas inte av effektivitetsskäl.

3.15.3 Tester

Vi har testat vad som är snabbast att komma åt, lokala variabler, publika objektmedlemmar eller privata objektmedlemmar (åtkomst genom metodanrop). Den variabel vi accessar är av typen *int*. För att mäta tiden använde vi metoden Date.getTime(). Testet gick ut på att lägga in värdet från de olika variablerna i en lokal variabel, i en loop som snurrade 10000 gånger.

Testerna har utförts på Siemens SL 45i, [25].

Försök nr	1	2	3	4	5	6	7	Medel
Lokal	136	124	129	120	120	120	120	124
Medlem (direkt åtkommst, public)	157	180	185	171	180	185	185	178
Medlem (åtkomst med get)	397	392	406	388	420	388	383	396

Tabell 3.8*: Accesstid för objekt på olika minnesplatser.*

Tabell 3.8 visar vilken accesstid objekt som är lagrade på olika ställen i minnet har. Tiderna är i millisekunder.

Vi ser i tabell 3.8 att man kan spara mycket tid genom att göra klassmedlemmar publika istället för att hämta deras värden med en metod. Det förstör dock inkapslingen och gör det möjligt att ändra värdet på medlemmen. Med en get-metod kan man bara läsa värdet, inte ändra på det. Om man skall använda ett värde flera gånger i en metod är det dock bättre att spara värdet i en lokal variabel och operera på denna istället. Detta är 30 % snabbare än att använda en klassmedlem, och 69 % snabbare än att använda en get-metod. Det är 55 % snabbare att använda publika klassmedlemmar än att använda en get-metod.

3.16 Blandade tips

I detta avsnitt finns några rekommendationer som inte passade in i något anat avsnitt. De har ingen direkt anknytning till varandra förutom än att de gör koden effektivare.

3.16.1 Rekommendationer

Ta bort redundant kod. Detta kan göras m h a kontrakt, läs mer om kontrakt i avsnitt 3.18 Använd plattformsspecifika metoder, av typen *System.metod*, t ex *System.arraycopy*. Dessa utnyttjar de speciella förhållanden på den maskin som den virtuella maskinen exekverar på vilket leder till att dessa metoder vanligtvis utförs snabbare än någon annan metod med samma uppgift.

Tänk på i vilken ordning AND och OR villkor förekommer, p g a s k short-circuit. Short-circuit innebär att om man har ett villkor, t ex a OR b, där a och b kan vara sanna eller falska, utvärderas först a. Om a är sant vet man att hela uttrycket kommer att vara sant. Detta betyder att man inte behöver utvärdera b, vilket inte en kompilator gör. Om processorn kan utvärdera uttrycket utan att gå igenom hela så gör den det. Se till att programmet kontrollerar det uttryck som kan avbryta utvärderingen så snabbt som möjligt.

Minimera anrop till klassen *Date*, eftersom metoden kräver ett I/O-anrop. Ta bort onödiga finesser. Om man till exempel ska koppla upp sig mot en server, är det bra om man skriver ut något på skärmen för att tala om för användaren vad som händer. Det finns inte någon anledning att ha en animation, liknande timglaset i Windows, det räcker med en textsträng. Om man bara behöver använda vissa finesser då och då, som t ex text för att presentera programmet eller instruktioner, flytta dem till egna applikationer så användaren kan ta bort dem om han/hon vill. Om man gör ett spel kan man lägga hjälptext och regler till spelet i en egen applikation som man inte behöver ladda ner i mobiltelefonen om man inte vill.

Återanvänd gränssnittet om det är möjligt. Detta gör inte bara applikationen mindre, det gör också att användaren lättare kan lära sig applikationen, d v s vilka knappar som finns var och så vidare. Den största delen av interaktiva applikationer går åt till det grafiska gränssnittet.

När man programmerar MIDP-applikationer är det viktigt att dela på GUI-kod och applikationens logik eftersom alla mobiltelefoner ser olika ut med olika, skärm, knappar, touchscreen etc. Det blir då lättare att skriva olika versioner till olika mobiltelefoner, eftersom endast GUI-delen behöver skrivas om.

Källor: [3], [21], [29]

3.16.2 Analys

När man ställer villkor i t ex en *while*-sats är det bra att se till att de villkor som ändras oftast undersöks först så att villkoren snabbt kan utvärderas. Problemet med detta kan vara att man inte alltid vet intuitivt vilka villkor som troligast avbryter utvärderingen snabbast.

Vissa finesser finns det ingen anledning att använda sig av i applikationen. Det gäller att avväga vad som ökar upplevelsen av applikationen och vad man klarar sig utan.

När man utvecklar en applikation bör man veta vilka begränsningar som finns i den exekveringsmiljön som applikationen skall exekveras i. I vanliga fall är det bra att dela upp applikationen i olika delar, t ex programlogik och GUI. Problemet med att göra det är att det blir extra mycket kod. Det blir fler klasser och mer skapande av objekt som har negativa effekter på minne och exekveringshastighet.

Som test kan man jämföra hur stor skillnad det blir i minnesutnyttjandet och exekveringshastigheten om varje del i programmet ligger i en egen klass eller om man har allt i en klass.

3.17 Obfuscating

I detta avsnitt undersöker vi obfuscation. Obfuscation är en teknik för att minska storleken på klassfilerna genom att byta ut namnen på privata metoder och variabler. Oftast använder man sig av speciella program som gör detta automatiskt direkt på klassfilerna, men man kan även göra detta för hand.

3.17.1 Rekommendationer

När man kompilerar koden i en Java-fil behåller klassfilen alla namn på metoder och variabler vilket gör att Java filer där metoder och variabler har långa beskrivande namn tar upp

mer minne än om alla namn var korta. Detta leder till längre nedladdningstider, men det är också lätt att dekompilera klassfiler och kopiera källkod. Detta kan avhjälpas med en Obfuscerare. Obfusceraren ändrar alla namn på alla variabler och metoder och ersätter dem med kortare varianter. Förutom att alla filer blir mindre blir det också svårare att stjäla koden, eftersom koden inte beskriver vad som sker i programmet.

Det finns obfuscerare som gör mer än att bara byta variabelnamn. De kan också modifiera bytekoden i klassfilerna för att göra det svårare att dekompilera klassfilerna till källkod. En nackdel med obfuscering är att om en användare upptäcker en bugg i programmet och skickar en utskrift på det undantag som kastats är det inte lätt att veta vart felet är. När ett undantag kastas står det i meddelandet som skrivs ut vilka metoder som anropats från den metod som kastat undantaget. Om man då har obfuscerat koden kommer metodnamnen bestå av korta obeskrivande ord, vilket gör det svårare att se vad som är fel.

Som regel döper obfusceraren inte om *public*-deklarerade metoder och variabler, eftersom andra klasser kan försöka anropa dessa metoder. De enda metoder/variabler som är det är helt säkert att ändra på är de som är privata.

Exempel:

Utan obfuscering

```
public void calcPayroll(RecordSet rs)
{
     while(rs.hasMove())
     {
          employee = rs.getNext(true);
          employee.updateSalary();
          distributeCheck(employee);
     }
}
```

Med obfuscering

```
public void a(a rs)
{
     while(rs.b())
     {
          b = rs.c(true);
          b.d();
          d(b);
```

```
    }
}
```

Problemet med många obfuscerare är att de kan ändra för mycket vilket innebär att när en obfuscerad fil blir dekompilerad går det inte att kompilera om den igen. Mer avancerade obfuscerare byter metodnamnen, vilket innebär att det blir svårt för dekompileraren att veta vilka metoder som ska anropas. Koden kommer då att se ut så här:

```
public void a(a rs)
{
    while(rs.a())
    {
        a = rs.a(true);
        a.a();
        a(a);
    }
}
```

Vissa obfuscerare ändrar om bytekodinstruktioner för att göra det mycket svårare att dekompilera klassfilerna.

Källor: [7], [10], [47]

3.17.2 Analys

När man ska obfuscera bör man inte göra det för hand. Det finns bra obfuscerare på Internet som automatiskt obfuscerar koden. Om man vill obfuscera för hand bör man göra det efterdet att man är klar med programmeringen av applikationen, eftersom obfuscering innebär att man ersätter metod/variabelnamn till korta och otydliga namn. Det kan då vara svårt att förstå sin egen kod. Man bör spara den vanliga koden ifall användaren skulle hitta en bugg, eftersom det är lättare att se i den vanliga koden vart felet finns.

3.17.3 Tester

Vi testade att obfuscera en liten MIDP-applikation. Testen gjordes i en PC.

Innan obfuscering	Efter obfuscering
2417 bytes	2379 bytes

Tabell 3.9*: Kodstorlek innan och efter obfuscering.*

Den fil vi använde gav inte någon större minnesförtjänst, som man kan se i tabell 3.9, eftersom den var väldigt liten. För större filer tjänar man mycket mer på att obfuscera.

3.18 Starka kontrakt

Detta avsnitt beskriver en metod för programmering som innebär mindre tester och framhåller objektorienteringens fördelar.

3.18.1 Rekommendationer

Att programmera med starka kontrakt innebär att man sätter upp ett kontrakt för hur en klass får användas. Att programmera med starka kontrakt innebär också att man litar på att användaren av en klass använder klassens metoder på rätt sätt, och alltså inte skickar in felaktiga värden till metoderna. Detta medför att dokumentationen för användandet av klassernas metoder måste vara tydlig.

Det faktum att man litar på att användaren av klasserna inte skickar in fel värden i metoderna, medför att man inte behöver testa de formella variablerna inne i metoderna, d v s de argument som finns i den anropade metoden. Det gör också att man kan garantera att metoden utför sin uppgift och att metoden returnerar ett korrekt värde, såvida användaren respekterar kontraktet.

En viktig del i kontraktet utgörs av för- och eftervillkor. Dokumentationen av vad man får skicka in till en metod och vad metoden utför skall se ut på nedanstående sätt:

```
//Förvillkor: 0 <= pos < arraySize()
//Eftervillkor: Värdet på position pos returnerat
int getValue(int pos)
{
        return array[pos];
}
```

Till metoden *getValue(int pos)* får man alltså skicka med ett värde som är större eller lika med noll och mindre än storleken på arrayen. Om man uppfyller dessa krav vet man att man får värdet på positionen pos i arrayen returnerat. Om man skickar in ett annat värde är resultatet odefinierat. För att anropa denna metod måste man alltså testa det värde man tänker skicka med för att vara säker på att det ska stämma överens med förvillkoren. Ett anrop av metoden skulle alltså gå till på följande sätt:

```
...
if(pos >= 0 && pos < arraySize())
{
        variable = getValue(pos);
        ...
}
...
```

Detta resulterar alltså i ett test innan metoden anropas. Om man inte skulle använda sig av för- och eftervillkor, utan låta testet ske inne i metoden skulle ovanstående kod istället se ut på följande sätt:

```
int getValue(int pos)
{
        if(pos >= 0 && pos < arraySize())
        {
                  return array[pos];
        }
        else
        {
                  errorFlag = true;
                  return 0;
        }
}
```

Ett anrop till metoden skulle se ut ungefär såhär:

```
...
variabel = getValue(pos);
if(isErrorFlagSet())
{
        //Felhantering
        ...
}
else
{
```

```
        //Gör något med variabeln
}
...
```

Man skulle också kunna använda sig av undantag, som man kan läsa om i avsnitt 3.10

Att inte använda för och eftervillkor resulterar i två tester, ett för att undersöka att rätt värde skickats med och ett för att undersöka om fel-flaggan är satt. Det resulterar också i att metoden *getValue(int pos)* anropas även i de fall då metoden inte kan utföra sin uppgift.

Om man jämför de två sätten att programmera, med eller utan för- och eftervillkor, ser man att det blir färre tester med för och eftervillkor plus att man inte behöver anropa metoden om man inte har rätt värde på *pos*.

Källa: [26]

3.18.2 Analys

Att inte testa villkor lika många gånger gör koden effektivare med avseende på tidsåtgång. När man skapar MIDlets är det dock oftast viktigt att veta hur en metod utför en uppgift, för att man skall veta vilken tid det tar och hur mycket minne metoden kräver. Om man använder detta programmeringssätt kan man införa ett till dokumentationsfält under förvillkor och eftervillkor, där man skriver hur mycket minne metoden behöver för att exekvera och hur lång tid den tar. På det sättet behöver man bara sätta sig in i hur metoden fungerar om man tycker att den tar för mycket tid eller minne och vill ändra på den.

3.18.3 Tester

Vi har testat hur mycket minne det tar att kasta ett undantag jämfört med att testa att värdet är rätt innan man anropar en metod. Innan och efter testkoden undersökte vi minnet med *Runtime.freeMemory()*. Vi testade minnesåtgång i fyra olika fall. Det första fallet var att inte anropa någon metod alls. Det motsvarar att metodens förvillkor ej är uppfyllt. Det andra fallet var att anropa en funktion vars förvillkor är uppfylt och som inte kastar något undantag. Det tredje fallet var att anropa en funktion som kan kasta undantag, men inte gör det. Det fjärde var att anropa en funktion som kastar ett undantag

Vi har också testat tidsåtgången för de ovan beskrivna fyra fallen. Vi mätte tiden för en loop som snurrade 10 000 gånger, samt exekverade testkoden. För att mäta tiden använde vi *Date.getTime()*.

Testen har utförts på Siemens SL 45i, [34].

Test nr	1	2	3	4	5	6	7	8	9	10	Medelvärde
Förvillkor ej uppfylt	0	0	0	0	0	0	0	0	0	0	0
Förvillkor uppfylt	0	0	0	0	0	0	0	0	0	0	0
Inget undantag kastat	84	84	84	84	84	84	84	84	84	84	84
Undantag kastat	84	84	84	84	84	84	84	84	84	84	84

Tabell 3.10*: Minnesåtgång vid användning av starka kontrakt kontra undantag.*

Test nr	1	2	3	4	5	6	7	8	9	10	Medelvärde
Förvillkor ej uppfylt	1034	1025	1025	1047	1047	1020	1029	1033	1006	937	1020,3
Förvillkor uppfylt	3277	3258	3364	3281	3275	3291	3314	3194	3286	3332	3287,2
Inget undantag kastat	3558	3531	3545	3572	3540	3540	3544	3535	3525	3517	3540,7
Undantag kastat	19055	19365	19189	19014	19028	19341	19217	19056	18553	19074	19089,2

Tabell 3.11*: Tidsåtgång vid användning av starka kontrakt kontra undantag.*

Av resultaten i tabell 3.10 framgår att det tar betydligt mer minne att kasta ett undantag än att testa värdet innan metoden. Eftersom endast minnet på heapen kan mätas är dessa resultat inte exakta. Det tar ju mer än inget minne att anropa en funktion. Man kan dock se att det tar upp minne på heapen att kasta ett minne. Förklaringen till att minne tas upp även när ett undantag inte kastas är antagligen det behöver allokeras minne för att testa om ett undantag kastats.

Från tabell 3.11 framgår det att det tar betydligt mera tid att programmera med undantag än att använda starka kontrakt. Om rätt värde används till funktionen är tidsskillnaden mellan att använda starka kontrakt eller undantag bara ca 7% till starka kontrakts fördel, men om fel värde används tar det ca 20 gånger längre tid att kasta ett undantag än att undersöka värdet innan en metod anropas.

Från ovanstående tester ser vi att man tjänar på att programmera med starka kontrakt. Dock är oftast det vanligaste fallet att man har rätt värde och inget undantag behöver kastas. I dessa

fall blir tidsvinsten inte så stor. Dock tjänar man inget på att använda undantag. Det finns alltså ingen anledning att använda undantag, om man kan testa på andra sätt.

4 GUI

Detta kapitel handlar om en MIDlets interaktion med användaren och vilka möjligheter det finns för att anpassa en MIDlet till den mobiltelefon den körs på. Hos referenserna som är upplistade i slutet av varje avsnitt finns information om det som tas upp i avsnittet. Hos de referenser som står inuti ett avsnitt finns information som endast tar upp den specifika delen.

4.1 Högnivå API

Meningen med Java är att en applikation skall fungera på alla sorters mobiltelefoner, oavsett vilka egenskaper som mobiltelefonen har. Eftersom mobiltelefoner ser olika ut, de kan b l a ha olika stora skärmar, olika antal knappar och vissa har touchscreen, är inte ovanstående alltid sant. I detta avsnitt presenterar vi några klasser som fungerar till alla mobiltelefoner. Samlingsnamnet för dessa är Högnivå API. I avsnitt 4.2 beskriver vi två klasser som har metoder som inte fungerar på alla mobiltelefoner. Samlingsnamnet för dessa är Lågnivå-API. Vid användning av högnivå-API:et är mobiltelefonen själv ansvarig för att placera ut komponenterna. De kommer alltid att finnas tillgängliga för användaren. När man använder lågnivå-API:et är programmeraren själv ansvarig för att placera ut komponenterna vilket kan leda till att bl a ett textfönster kan hamna utanför skärmen.

4.1.1 Rekommendationer

Högnivå-API:et är designat för applikationer som skall fungera på alla sorters mobiltelefoner som kan köra MIDlet:s. De klasser som hör till högnivå API:et är *Screen*, de fyra subklasserna *Alert*, *Form*, *List* och *TextBox* samt *Item* och de 6 subklasserna *ChoiceGroup*, *DateField*, *Gauge*, *ImageItem*, *StringItem* och *TextField*. För att åstadkomma denna portabilitet får man ge upp kontrollen över hur användargränssnittet kommer att presenteras på mobiltelefonen. Man kan heller inte bestämma på vilket sätt som information skall hämtas från användaren. Eftersom alla telefoner ser olika ut kommer en applikation inte att se ut på samma sätt på alla mobiltelefoner. Så länge det är mest textinformation och enkla bilder som skall visas fungerar de ovanstående klasserna bra.

Källor: Bilaga B

4.1.2 Analys

Använd detta API för att göra menyer och för att presentera textinformation och bilder där bildernas placering inte är viktig. Man vet nämligen inte exakt vart bilderna kommer att hamna. Det kan se ut på olika sätt och att navigera mellan menyerna kan vara tidskrävande på vissa mobiltelefoner. Använd därför inte detta API för applikationer som har krav på att användaren snabbt ska kunna mata in information. Användaren kan nämligen behöva gå igenom flera menyer för att mata in information.

4.2 Översikt lågnivå API

Om man vill ha mera kontroll över hur applikationen kommer att presenteras, räcker inte högnivå-APIet till. I detta avsnitt beskrivs två klasser som ger mer kontroll till programmeraren.

4.2.1 Rekommendationer

Med lågnivå-API:et har man mer kontroll över hur informationen kommer att visas på skärmen, och hur användaren kommer att få mata in information. De klasser som hör till lågnivå-API:et är Canvas och Graphics. Detta API passar bäst när man skall göra spel eller av annan anledning vill visa text och bilder på ett bestämt ställe på skärmen och uppdatera skärmen ofta. Nackdelen med detta API är att applikationen måste ta reda på vilka egenskaper som stöds av den mobiltelefon applikationen befinner sig på. Detta leder till att koden blir större och mer komplex. Ett alternativ är att utveckla applikationer som bara fungerar på en viss mobiltelefonmodell. Mer om detta i avsnitt 4.3.

Källor: [48]

4.2.2 Analys

Använd detta API för att göra applikationer där det behövs kontroll över var bilder och text hamnar och på vilket sätt som information från användaren skall hämtas samt där bilder uppdateras ofta. Man kan dock inte räkna med att MIDlet:en kommer att fungera på alla plattformar. Man kanske använder knappar i programmet som inte finns representerade på mobiltelefonen, eller ritar ut för stora bilder etc.

4.3 En MIDlet som anpassar sig eller en MIDlet för varje mobiltelefonmodell?

När man utvecklar MIDlets vill man oftast att den skall fungera på så många mobiltelefonmodeller som möjligt för att få störst lönsamhet. Frågan är vilket sätt som är det enklaste att utveckla dessa MIDlets på och hur bra applikationen fungerar.

4.3.1 Rekommendationer

- Att göra en applikation som skall fungera på en viss modell av mobiltelefon.

Detta kan vara bra om man skall skriva en applikation till ett företag som bara köper en viss modell av mobiltelefoner eller om man behöver använda sig av de mobiltelefonspecifika klasser som ofta finns (t ex fler ljud, vibration, tända/släcka ljus mm). Fördelarna med att utveckla till en viss modell är att man vet hur mycket minne som finns tillgängligt, vilken typ av knappar som finns, hur stor skärmen är etc. Då kan man undvika en massa tester och programmet blir inte lika stort. Nackdelen med detta tillvägagångssätt är att man begränsar sig till en viss modell av mobiltelefon. Om marknaden för denna modell inte är tillräckligt stor riskerar man att applikationen inte lönar sig. Man kan då dela upp programmet i de olika delarna GUI, mobiltelefonspecifika komponenter och övrig kod. Med denna uppdelning behöver man bara ändra i GUI och mobiltelefonspecifika komponenter för att applikationen skall fungera till en annan mobiltelefonmodell.

- Att göra en applikation som fungerar om mobiltelefonen har vissa egenskaper (t ex vissa knappar, touchscreen, visst minne, färgskärm etc.)

På detta sätt utvidgar man den mängd mobiltelefoner som kan använda applikationen. Samtidigt vet man vilka metoder som dessa mobiltelefoner stöder. Man behöver då inte testa vilka metoder som stöds och programmet blir mindre och snabbare. Det man förlorar är de komponenter som är specifika för vissa modeller av mobiltelefoner. Om man tror att man kan klara sig utan dessa är detta utvecklingssätt att föredra.

- Att göra en applikation som beter sig olika beroende på vad mobiltelefonen har för metoder

M h a de metoder som beskrivs i avsnitt 4.5 kan en applikation anpassa sig till den mobiltelefon som applikationen befinner sig på. På detta sätt kan man få en applikation som fungerar på alla mobiltelefoner. Nackdelen är, i och med att många tester måste göras och

olika kod skall exekveras beroende på testernas utfall, att applikationen blir större, långsammare och mer komplex.

Källor: [11]

4.3.2 Analys

Det finns inte ett sätt som är bäst utan man får utreda vilket sätt man vill utveckla på vid varje nytt projekt. I framtiden kommer det dock att finnas många fler mobiltelefoner med stöd för MIDlets. Vill man då ha en applikation som fungerar för alla plattformar kan det bli en väldigt stor uppgift att skriva en applikation till varje.

4.4 Designtips gällande laddningstider, navigering mm

Vi tar här upp hur det grafiska gränssnittet skall se ut för att en applikation skall vara lättillgänglig och lätthanterlig. När man kör en MIDlet i en mobiltelefon är det vanligt med väntetider. Att sitta och vänta är inte den roligaste sysslan. Här tar vi också upp vad man kan göra för att få laddningstiderna att kännas mer uthärdliga.

4.4.1 Rekommendationer

Om flera instruktioner utför liknande operationer skall dessa instruktioner aktiveras och användas på samma sätt. Det gör att användaren snabbare lär sig hur en applikation fungerar, och att applikationen blir mer lättanvänd.

Användaren skall bestämma över applikationen, inte tvärt om. T ex skall hjälp bara visas om användaren ber om det och man skall inte avbryta nedladdning, minnesskrivning eller andra tidskrävande operationer med timeout utan upplys användaren att det operationen tagit ovanligt lång tid och ge honom möjlighet att avbryta.

Användaren skall kunna se var i menyerna som han befinner sig och dessa skall vara uppbyggda i en hierarkisk struktur, så att användaren lätt kan ta sig till det ställe han befann sig på tidigare t ex om han tryckt på fel knapp. Eftersom telefonmenyer oftast är uppbyggda på detta sätt kommer användaren att känna igen sig och snabbare lära sig applikationen.

Applikationer som härstammar från kända PC-applikationer skall likna dessa applikationer. Detta gör att användaren känner igen sig och snabbare lär sig applikationen. Om en köpare väljer mellan två olika applikationer är det sannolikt att han väljer den han känner igen.

Källor: [44]

4.4.2 Våra rekommendationer

För att göra laddningstiderna mindre tråkiga, när man väntar på att något skall hämtas på eller skickas till en server eller när man skriver till eller läser från minnet, kan man visa någon information t ex en rolig bild, något som rör sig etc. En bra information att visa är instruktioner över hur man använder programmet. Det skall också alltid finnas en mätare som visar hur lång tid det är kvar att vänta. Om man kan förutsäga den ungefärliga laddningstiden, är det bra att visa en tid som räknar ned. Då kan användaren avbryta direkt om han tycker att väntan blir för lång. Om inte en sådan mätare finns är risken stor att användaren tror att programmet låst sig och börjar knappa på tangenterna eller stänger av programmet. Om man har en mätare som går långsammare i början för att sedan accelerera ger detta en illusion av att väntan blir mindre.

4.4.3 Analys

Det är bra om användaren känner igen sig från tidigare applikationer han använt. En PC är dock ganska annorlunda i skärmstorlek, processorkraft, minne och inmatningshastighet. Det kan vara bättre att strukturera en applikation på ett nytt sätt när den skall användas till en mobiltelefon, jämfört med dess motsvarighet till PC. Om användaren känner igen sig men har svårigheter med att använda applikationen är kommer han antagligen att byta till en mer lättanvänd applikation.

Det är bra att ge användaren kontroll över applikationen. Fönster som öppnar sig själv med hjälp och tips man inte behöver är väldigt frustrerande. Att ge möjlighet att avbryta skrivning till det beständiga minnet är dock inte alltid att föredra. Om det är viktig data riskerar denna att förstöras.

Att ha instruktioner och mätare vid laddning är bra för att användaren skall vara kunna följa händelseförloppet i programmet. Det gör dock att programmet blir långsammare eftersom det måste uppdatera mätarna, vilket tar lite tid. Ibland kanske det är bättre att bara låta programmet ladda utan grafiska finesser. Det beror på hur mycket längre laddningstiden blir. Det är ett test som man får göra för sin MIDlet. Då kan det också vara bra att ta reda på ungefär hur lång tid det tar innan användaren tröttnar på att vänta och avslutar programmet. Då kan man också testa om det blir någon skildnad om man har en mätare mot om man inte visar någon information om kvarvarande laddningstid.

4.5 Information som en MIDlet kan få från en mobiltelefon

Om man vill ha en MIDlet som skall anpassa sig till mobiltelefonen behöver man metoder för att ta reda på vad mobiltelefonen har för egenskaper. I detta avsnitt beskriver vi vad man kan och vad man inte kan ta reda på om mobiltelefonen.

Med MIDP:s fördefinierade metoder kan man ta reda på följande om mobiltelefonen:

- Färgskärm eller inte
- Antal färger/gråskalor
- Hur stor skärmen är
- Om doublebuffring utförs av mobiltelefonen
- Om mobiltelefonen har touchscreen
- Om mobiltelefonen känner av att en knapp hålls nedtryckt
- Hur mycket minne finns tillgängligt
- Hur mycket beständigt minne finns tillgängligt
- Vilka klasser som finns tillgängliga

Nedan följer en närmare beskrivning av hur man tar reda på ovanstående.

-färgskärm eller inte

Detta görs med metoden *isColor* som finns i klassen *Display*.

-antal färger/gråskalor

Med metoden *numColors* som finns i klassen *Display*, kan man ta reda på antalet färger, om man har en färgskärm, eller antalet gråskalor, om man har en svartvit skärm, som skärmen kan visa.

-hur stor skärmen är

Genom att anropa metoderna *getHeight* och *getWidth* i Klassen *Canvas* kan man få reda på hur stor skärm som mobiltelefonen har.

-om doublebuffering utförs av mobiltelefonen

Att rita upp enskilda streck, figurer och text är långsamt, vilket leder till att skärmen flimrar eftersom uppdateringarna sker oftare än man hinner rita ut de olika figurerna på skärmen. För att förhindra detta använder man sig av dubbelbuffring. Dubbelbuffring innebär att man ritar alla bilder som skall synas på skärmen till en grafikbuffer, för att sedan rita ut grafikbuffern på skärmen genom att kopiera över grafikbuffern till skärmbuffern. Eftersom kopiering är väldigt snabbt visas förändringen nästan omedelbart. Att göra dubbelbuffring i MIDP är enkelt, man använder sig av *Image*-klassen för att skapa en grafikbuffer, sedan använder man sig av *Graphics*-klassen för att rita till/från grafikbuffern. Till slut använder man samma klass för att föra över grafikbufferten till displayen. I implementeringen i vissa system finns det automatiskt stöd för doublebuffering vilket gör att man inte själv behöver andvända det. Att testa om ett system har automatisk doublebuffering gör man med metoden *isDoubleBuffered()* som finns i klassen Canvas.

Ett exempel:

```
public class MyCanvas extends Canvas {

    private Image offscreenImage = null;
    private int   height;
    private int   width;

    public MyCanvas(){
        height = getHeight();
        width = getWidth();

        if( !isDoubleBuffered() ){
            offscreenImage = Image.createImage( width, height );
        }

      ...
    }

    ...
}
```

Ett exempel på *paint* utan doublebuffering:

```
public void paint( Graphics g )
{
    g.setColor( 255, 255, 255 );
    g.fillRect( 0, 0, width, height );
}
```

Ett exempel på *paint* med explicit doublebuffering:

```
public void paint( Graphics g )
{
    Graphics offScreengraphics;
    if( offscreenImage != null )
    {

        offScreengraphics = offscreenImage.getGraphics();
    }
    offScreengraphics.setColor( 255, 255, 255 );
    offScreengraphics.fillRect( 0, 0, width, height );
    g.drawImage(offscreenImage);

}
```

Nackdelen med doublebuffering är att man måste ha en extra bild (grafikbufferten) laddad i programmet vilken kan ta upp stor minnesplats. För att inte ha en offscreenbuffer i onödan, t ex när ett objekt av klassen Canvas inte visas på skärmen, kan man omdefiniera Canvas *showNotify* och *hideNotify*-metoder. Dessa anropas när Canvas visas respektive inte visas på skärmen.

- Man kan också ta reda på om mobiltelefonen har touchscreen

Detta kan göras m h a metoderna *hasPointerEvent* och *hasPointerMotionEvent*. Om minst en av dessa returnerar true har mobiltelefonen touchscreen. hasPointerEvent() betyder att mobiltelefonen kan känna om användaren trycker på skärmen och hasPointerMotionEvent() betyder att mobiltelefonen kan känna om användaren drar något på skärmen.

-om mobiltelefonen känner av att en knapp hålls inne

När man håller inne en knapp anropas *keyRepeated* i *Canvas*-klassen. För att se om mobiltelefonen stöder denna metod kan man anropa metoden *hasRepeatEvent* i klassen Canvas.

-hur mycket minne som finns tillgängligt

Med metoderna *freeMemory* och *totalMemory* i klassen *Runtime* kan man få reda på hur mycket minne som finns ledigt och hur mycket minne som mobiltelefonen erbjuder totalt.

-hur mycket beständigt minne finns tillgängligt

Om man anropar metoden *getSizeAvailable* i klassen *RecordStore* får man reda på hur mycket beständigt minne som finns kvar för applikationen att använda.

-vilka klasser som finns tillgängliga

Om man försöker anropa en klass från ett mobiltelefonspecifikt API och applikationen inte exekverar på rätt mobiltelefon kastas ett *ClassNotFoundException*. Detta kan man använda för att undersöka vilken typ av telefon som applikationen ligger på då. T ex kan man anropa metoden *Class.forName(”com.siemens.mp.game.ExtendedImage”)* och om denna metod kastar ett undantag vet man inte befinner sig på en Siemens SL45i, eftersom *com.siemens.mp.game.ExtendedImage* finns tillgänglig i denna telefon.

Man kan inte ta reda på följande:

- Vilka ljud som finns
- Vilka knappar som finns

-vilka ljud som finns

I MIDP-API:et finns fem ljud definierade. Dessa behöver dock inte låta olika, på vissa mobiltelefoner hörs samma ljud oberoende av vilket man väljer att spela upp. Det finns inget sätt för en applikation att ta reda på om det blir samma ljud för alla eller om det blir olika ljud.

-vilka knappar som finns

Det går inte för en applikation att se vilka knappar som finns tillgängliga på telefonen. Därför är det bäst att vid inmatning använda klassen TextBox eller List. Dessa klasser låter mobiltelefonen bestämma hur inmatningen skall gå till. Om man måste använda sig av knapptryckningar vid inmatning (vissa mobiltelefoner använder främst touchscreen) är det bäst att använda knapparna som i MIDP döpts till UP, DOWN, LEFT, RIGHT, FIRE. Dessa kallas för *gameAction* och finns på de flesta mobiltelefoner. Alla knappar representeras av ett heltal vilket kallas keycode. För att få deras keycode anropar man metoden *getKeyCode* som finns i klassen *Canvas*.

Källor: [34], [39]

4.5.1 Analys

Om man vill ha en anpassningsbar MIDlet kan det vara bra att spara all information om mobiltelefonen, som MIDleten befinner sig på, i variabler. Det kommer att bli många tester mot denna information och att anropa ovanstående funktioner kan ta lång tid.

4.6 Siemens SL45i API

Om man har tänkt sig att utveckla applikationer för Siemens SL45i, får man tillgång till fler bibliotek. I detta avsnitt beskrivs vilka extra metoder man kan använda på Siemens SL54i om man använder deras egna bibliotek.

Siemens har definierat tre bibliotek. Dessa är *com.siemens.mp.game*, *com.siemens.mp.gsm* och *com.siemens.mp.io*.

I *com.siemens.mp.game* finns klasser och metoder för att tända och släcka skärmbelysningen, starta och stanna vibratorn, skapa och spela upp melodier, spela enskilda ljud samt flera klasser för att rita bilder och enskilda eller rader av pixlar till skärmen. Att rita enskilda pixlar istället för hela bilder kan göra att applikationen blir snabbare. Man kan även sätta enskilda pixlar. Denna funktionalitet gör att det är möjligt att förstora och förminska bilder på egen hand. Om man t ex vill ha en bild som är hälften så stor som ursprungsbilden kan man skapa en tom bild med önskad storlek, läsa varannan pixel från ursprungsbilden och lägga in dessa pixlar i den nya bilden. Om en bild med storleken 12 består av pixlarna 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 kan man plocka ut 1, 3, 5, 7, 9 och 11 för att få en bild som är hälften så stor som den ursprungliga.

I *com.siemens.mp.gsm* finns klasser och metoder för att ringa telefonsamtal och skicka SMS.

I *com.siemens.mp.io* finns klasser och metoder för bl a filhantering och för att öppna datakanaler för SMS och en infraröd port.

Källor: [6], [24]

4.6.1 Analys

Med Siemens API kan man använda många funktioner på mobiltelefonen som inte finns tillgängliga om man endast använder MIDP API:et. Användande av detta Siemens-specifika API kommer dock att göra MIDlet:en exklusiv för Siemens SL45i. Om MIDlet:en kommer att fungera på Siemens kommande mobiltelefoner vet man inte i dagsläget, men det verkar troligt. Men om man vill ha tillgång till SMS, vibrationer och dylikt är det enda man kan göra att använda sig av Siemens API. Det kan dock vara riskabelt att tillåta program att skicka SMS och ringa telefonsamtal. Ett elakt program skulle kunna ringa samtal och skicka SMS som man inte vill skall bli ringda eller skickade. Användaren måste dock tillåta varje SMS genom att klicka på OK innan meddelandet skickats.

4.7 Motorolas API

Om man har tänkt sig att utveckla applikationer för Motorolas Acompli008 får man tillgång till några extra bibliotek. I detta avsnitt beskrivs vad dessa bibliotek innehåller.

4.7.1 Rekommendationer

Motorolas API, som kallas för LWT (Leightweight Windowing Toolkit) innehåller klasser för att lägga till knappar, checkboxar, bilder med bildtext, justerbara mätare och textfält. LWT:s klasser är menade för Canvas vilket ger programmeraren mer kontroll över var komponenterna skall hamna, än om man skulle använda högnivå API:et. Det ger dock inte samma kontroll över mobiltelefonen som man kan uppnå med Siemens SL45i, där man kan skicka SMS, ringa, tända och släcka lampan och få telefonen att vibrera etc.

Källa: [24]

4.7.2 Analys

Motorolas API ger ingen kontroll över funktioner på mobiltelefonen som man inte kan använda med MIDP API. Det finns därför ingen större anledning att använda detta API och därmed göra MIDlet:en Motorola-exklusiv. Men om man ändå har tänkt göra applikationen Motorola-exklusiv (t ex om man vill använda touchscreen, det bara Motorola Accompli 008 som har en touchscreen-skärm) kan man med fördel använda ovanstående klasser.

4.8 Storlek på bilder och text

När man visar bilder vill man ofta ha en text som följer med bilden. Problemet är att denna text blir olika stor på olika mobiltelefoner. Nedan presenteras en lösning på detta problem. Ett annat problem är att olika telefoner har olika storlek på skärmarna. På vissa mobiltelefoner är en bild för stor, samtidigt som den kan vara för liten på andra.

Oftast vill man att texten skall ha samma eller kortare längd än bilden. För att ta reda på bildens längd finns metoden *getWidth* i klassen Image. Sen kan man skapa en *Font*, vilken bestämmer textens storlek och om texten skall vara fet, kursiv understruken eller vanlig. Det finns tre textstorlekar man kan välja på, liten mellan eller stor. När man skapat en *Font* kan man skicka in sin textsträng till metoden *charsWidth.* Denna metod returnerar hur lång texten kommer att bli (i pixlar) om man använder nuvarande textinställning. Om man tycker att den är för stor eller för liten ändrar man bara på storleken i *Font*. När man är nöjd med storleken gör man den valda *Font*:en till den som gäller med metoden *setFont* som finns i klassen *Graphics*.

Det finns inga metoder i MIDP för att förminska eller förstora bilder. Det man får göra istället är att ladda ned flera storlekar av samma bild, undersöka vilken storlek skärmen har och sen rita ut rätt version av bilden. Ett annat sätt är att öppna en nätverksförbindelse och ladda ned bilden efter att programmet startat och läst av storleken på skärmen. Alternativt kan man förstora och förminska bilders utseende på skärmen genom att anropa repaint(int x, int y, int width, int height) många gånger. Detta tar dock en väldig tid. Mer om detta i avsnitt 4.8.2. Där beskrivs också vad som händer om man ritar ut en bild som är för stor för skärmen.

För att få bilder som passar displayen på en viss mobiltelefon skulle man kunna packa MIDlet:ens jar-fil efter att någon har begärt att få den nedladdad. Då vet man vilken telefon som begär MIDlet:en och man kan packa ned de bilder vars storlek passar mobiltelefonen.
Källor: [39]

4.8.1 Analys

Räkna inte med att det går att få en text som passar perfekt till bilden. Eftersom det bara finns tre storlekar att välja på kommer man inte alltid att kunna välja en storlek som man blir nöjd med. Testa alltid programmet på flera mobiltelefonplattformar. Det kommer att se olika ut på olika plattformar.

Vilket sätt man skall välja när det gäller nedladdning av bilder beror på hur mycket tid och plats det tar. Om man skall ha flera storlekar av samma bild eller om man laddar ner rätt bild i efterhand, avgör vilket alternativ som blir bäst. Mer om detta i avsnitt 6.3.3 och 5.1.2.

4.8.2 Tester

Vi har testat ett sätt att rita ut förstorade eller förminskade bilder på skärmen. Låt två nästlade for-loopar gå lika många varv som bilden skall vara hög och bred, och anropa repaint(int x, int y, int width, int height) i den innersta for-loopen varje varv. På detta sätt kan man styra till vilken pixel på skärmen som man vill skriva till. Sen kan man flytta bilden för varje varv så att man ritar ut en pixel av bilden ha på ett visst ställe på skärmen. På detta vis kan man förminska eller förstora en bild genom att inte rita ut vissa pixlar eller rita ut vissa flera gånger.

```
//De nästlade looparna för att få bilden hälften så stor som
//originalet
for(width=0; width<(image.getWidth()/2);width++)
{
          for(height=0; height<(image.getHeight()/2);height++)
          {
                    repaint(width,height,1,1);
                    serviceRepaints();
          }
}

//skall ligga i paint-metoden
g.drawImage(image, (0-width), (0-height), 0);
```

Vi har också testat hur man kan skrolla en bild på mobiltelefoner om man använder *Canvas*. Använder man *Form* och *ImageItem* istället, skapas en scrollbar automatiskt. Om man kör programmet i en emulator, en miljö på en dator som skall likna miljön i en

mobiltelefon, är det enkelt. När man ritar ut en för stor bild i emulatorn visas bara den del av bilden som får plats på skärmen. I emulatorn kan man också rita ut bilden till minuskoordinater för att på så sätt se de delar av bilden som hamnat utanför bilden. För att skrolla en bild är det bara att ändra de koordinater som bilden ritas ut med. Tyvärr fungerar denna strategi inte till Siemens SL45i eller Motorola Accompli008 där man inte kan rita ut bilder till minuskoordinater. På Siemens SL45i verkar den förstora bilden förminskas till en bild som får plats på skärmen vilket gör att man förlorar detaljer och på Motorola Accompli008 kommer ingen bild upp alls. För att skrolla bilder på dessa mobiltelefoner får man skapa många små bilder och sen byta ut de som skall ritas på skärmen så att det ser ut som om det är en enda stor bild som skrollar.

Att dela upp bilden i flera delar är en bra strategi eftersom man då bara behöver de delar av bilden som får plats på skärmen. Om man t ex vill ha en applikation som visar en karta över den plats som man befinner sig på kan man ladda ned en ny del av kartan varje gång som man flyttar sig till ett nytt område.

När man ritar ut en bild som är för hög m h a *Form,* kommer en pil upp nederst på skärmen. Trycker man på nedåttangenten skrollas skärmen nedåt, och man får se resten av bilden. Om bilden är för bred kommer ingen högerpil upp på skärmen, och trycker man på högertangenten skrollas inte skärmen åt höger. Det innebär att den högra delen av bilden ej kan ses, utan försvinner.

5 Kommunikation

I detta kapitel går vi igenom vad man bör tänka på när man utvecklar MIDlets som skall kommunicera med serverapplikationer. Hos referenserna som är upplistade i slutet av varje avsnitt finns information om det som tas upp i avsnittet. Hos de referenser som står inuti ett avsnitt finns information som endast tar upp den specifika delen.

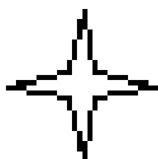
5.1 Http-förbindelser

I detta avsnitt går vi igenom vad man bör tänka på vid kommunikation med en server. Undvik att köra beräkningsintensiva processer direkt på mobiltelefonen utan kör dem på servern istället. Istället för att ladda ner data till mobilen och sortera den där är det bättre att servern sorterar innan den skickar datan eftersom den kan göra det mycket snabbare. Nedladdningstiden kommer inte att minskas, men man slipper utföra en tidskrävande sortering på mobiltelefonen. När man vill koppla upp sig mot en server bör man använda sig av en bakgrundstråd, för att inte riskera att applikationen låser sig medan den försöker koppla upp sig mot servern.

Vid kommunikation med en server, tänk då på att:

- Det finns inget stöd för sockets, utan kommunikationsmetoden som används är http-request/response

- Kommunikation mellan en mobiltelefon och en server tar lång tid. Att ladda ned en liten svartvit PNG-bild (se Figur 5.1) tar ca 20 sekunder.



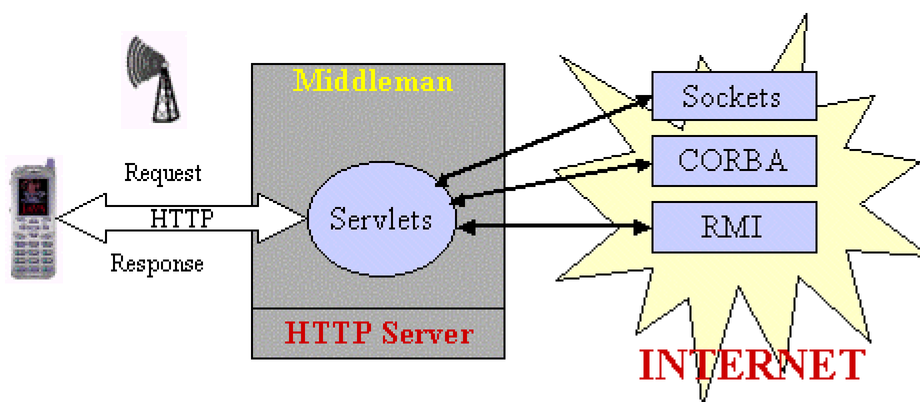
Figur 5.1: *Liten svartvit PNG-bild*

- Om flera värden måste skickas mellan client/server bör detta göras i ett enda objekt istället för att skicka ett värde i taget. Det går också bra att sätta ihop informationen till en sträng. Det viktigaste är att man skickar allt på samma gång för att det ska bli billigare.

- Undvik att skicka information som inte behövs, som t ex periodvisa uppdateringar även om objektet inte har ändrats.

MIDP har inget direkt stöd för sockets, datagramförbindelser eller RMI. RMI står för Remote Method Invocation och är ett system för att anropa funktioner på en serverapplikation. Därför är det inte möjligt att använda socket och RMI-baserade applikationer direkt, utan man måste använda en servlet. En servlet är en applikation som körs på en server. Servleten tar hand om midp-applikationens metodanrop och sköter sedan kommunikationen med eventuella rmi- och socketförbindelser. Mellanvaran ansvarar för att skicka tillbaka resultatet till klient-applikationen.

Det finns stöd för uppkoppling via Http-protokollet. Alla förbindelser öppnas m h a klassen *Connector*s *open*-metod. Lyckas metoden returneras ett objekt som implementerar interfacet *HttpConnection*.



Figur 5.2*: Illustration av sambandet mellan en mobiltelefon och en serverapplikation som gör det möjligt att simulera bl a RMI. [18]*

I bilaga E finns kod för att skapa en HTTP-förbindelse mot en servlet, skriva ett meddelande till servern och ta emot ett svar som den skriver ut på displayen. För att göra en servlet-applikation använder man sig av J2EE-API:t, [41]. I bilaga F finns kod för

kommunikation i en servlet, där servleten tar emot en HTTP-uppkopplingsbegäran av typen POST. Typen POST använder datafältet i det paket som skickas mellan klienten och servern för att överföra information till klientapplikationen. Detta gör att informationen inte behöver ha en maxstorlek. Headrarna i ett paket får bara vara av en viss storlek, medan storleken på informationen i datafältet kan variera. Man skulle också kunna använda sig av en GET-metod som innebär att serverns svar skickas direkt i headern. Sedan ändras svarsmeddelandet till typen *plaintext*, som är en textsträng. En skrivare till en *HttpServletResponse*-klass skapas så att man kan skriva tillbaks till MIDlet-applikationen.

Man kan använda metoden *setRequestProperty* från interfacet *HttpConnection* för att sätta vissa egenskaper som förbindelsen skall ha. Ex:

```
connection.setRequestProperty("IF-Modified-Since",
  "20 Nov 2001 16:33:19 GMT");
```

Denna metod talar om att servern inte skickar tillbaka data om den inte har modifierats sen ett visst datum.

Vissa plattformsspecifika API:er kan ha tillgång till TCP/UDP sockets. Då behövs inte mellanvaran, men fördelen med mellanvaran är att man kan använda den till att generalisera applikationen så att samma applikation fungerar på olika plattformar.

När man gör en nätverksuppkoppling bör man tala om för användaren vad som händer, så att de inte tror att programmet har låst sig. Detta beror på de ofta långa väntetider som kan uppstå när man måste skicka information via luften. När man väljer att utföra ett kommando eller ett menyval anropar systemet metoden *commandAction*. Metoden är en tråd som tillhör systemet och om man använder den tråden för att skapa en nätverksförbindelse kan det ta lång tid, vilket innebär att systemtråden måste vänta. En regel för MIDlet-trådning är att de enda trådar som tillhör en är de man skapar själv. De trådar som skapas av systemet och inte explicit av programmeraren bör inte användas för att utföra långsamma operationer. Detta är en skillnad från J2SE/J2EE-applikationsprogrammering där den applikationen exekveras i en virtuell maskin som kan köra parallellt med andra applikationer. T ex i Windows kan man köra en Javaapplikation samtidigt som man skriver i ett dokument i Word.

Skriver man enligt följande exempel kommer systemtråden att fastna i *connect*-metoden medan uppkopplingen görs.

```
public void commandAction(Command c, Displayable s)
{
    if (c == mExitCommand)
        notifyDestroyed();
```

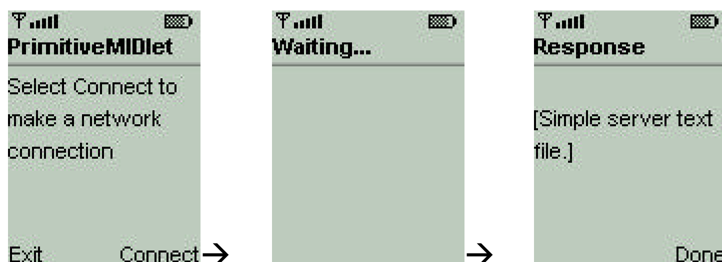
```
    else if (c == mConnectCommand)
    // Uppkoppling görs
      connect();
  }
```

När man använder *Form*-klassens *addCommand*-metod lägger man till kommandon till systemet. Kommandon används för att svara på knapptryckningar från användaren av applikationen. Dessa kommandon kommer att följa med i variabeln *c* när metoden *commandAction* anropas av systemet. Variabeln s talar om på vilken del av användargränssnittet som händelsen skett. Det är bättre att låta *connect*-metoden att köra i en egen tråd. Ett problem som då kan uppstå är att en otålig användare kan trycka på *connect*-knappen upprepade gånger vilket leder till att applikationen skapar flera förbindelser. Detta är dock lätt att lösa genom att man under uppkopplingen visar en ny skärm. På den nya skärmen kan man också skriva ut statusinformation om uppkopplingen så att användaren vet vad som händer. Koden kan då se ut så här:

```
public void commandAction(Command c, Displayable s)
{
    if (c == mExitCommand)
    notifyDestroyed();
    else if (c == mConnectCommand)
    {
      // Visar statusfältet
      mDisplay.setCurrent(mWaitForm);

      Thread t = new Thread()
      // Definierar en inre klass samtidigt som ett objekt av
      // klassen skapas.
      {
        public void run()
        {
          connect();
        }
      };
      // startar tråden
       t.start();
     }
 }
```

Så här skulle det kunna se ut på skärmen:



Figur 5.3*: Figurerna visar hur de olika skärmbilderna kan se ut. [16]*

Nu har vi skapat grunden för en nätverksapplikation. Det saknas dock ett par saker för att den skall vara riktigt bra. När man har startat uppkopplingen kan det hända att något inträffar och man vill avsluta uppkopplingen innan den är klar. Vi vill alltså lägga till en *cancel*-knapp som läggs i samma skärmyta som visar *"Waiting..."*. Dess kommando bör då läggas i samma kommandolyssnare som vi tidigare skapat till huvudytan. För att inte göra MIDlet-klassen för stor och komplicerad flyttar vi ut allt som har med uppkopplingen att göra till en egen klass. Det gör att MIDleten snabbare startas när inte lika mycket data behöver laddas in. När detta är gjort ser MIDlet-klassens *commandAction*-metod ut så här:

```
public void commandAction(Command c, Displayable s)
{
    if (c == mExitCommand)
       notifyDestroyed();
    else if(c == mConnectCommand)
    {
       mDisplay.setCurrent(mWaitForm);
       // Skapar formulär som visar status till användaren

       myConnector = new MyConnector(this, "http://...");
       //Startar den nya klassen MyConnectors tråd
       mWorker.start();
    }
    else if (c == mCancelCommand)
    {
       mDisplay.setCurrent(mMainForm);
       // Visar huvudformulär
```

```
        myConnector.cancel();
        // Avbryter uppkopplings-försök

        myConnector = null;
      }
  }
```

Den nya klassen, *MyConnector*, som tar hand om uppkopplingen finns i bilaga C. Klassen tar hand om all kommunikation. Om användaren har startat uppkopplingsfasen, och den inte är färdig kan användaren avbryta uppkopplingen genom att anropa *cancel*-metoden. När man sen trycker på *connect*-knappen anropas klassens *go*-metod som väcker tråden, som sätter igång att koppla upp sig.

När man skapar en förbindelse med en server kan man skriva så här:

```
OutputConnection connection =
   (OutputConnection)Connector.open(URL);
OutputStream os = connection.openOutputStream();
```

Om man inte behöver använda sig av variablen connector från exemplet ovan kan man göra så här istället:

```
OutputStream os = Connector.openOutputStream(URL);
```

Fördelen med detta är att man skapar ett objekt mindre. Vid kommunikation med en server bör man tänka på att det kan vara svårt att veta om man fortfarande har en förbindelse mellan servern och mobiltelefonen. Om man t ex åker tåg och kommer in i en tunnel finns det en risk att man förlorar förbindelsen. Möjliga lösningar för detta problem är att man använder sig av ”acknowledgement”-meddelanden för att tala om att man mottagit ett meddelande. Nackdelen är då att man måste vänta på meddelandet vilket gör att applikationen blir långsammare.

Källa: [12] , [18], [19]

5.1.1 Analys

När man skickar information mellan mobiltelefonen och serverapplikationen kan man skicka datan i ett enda objekt. Nackdelen med detta är att man kanske måste sätta samman

information som annars finns i olika objekt till ett nytt objekt för att undvika att skicka flera objekt.

Det är också värt att tänka på hur mycket kommunikationen kommer att kosta användaren i form av avgifter för användande av nätverket. Skicka därför så lite information som möjligt och upplys användaren om kostnaden innan kommunikationen genomförs.

5.1.2 Tester

Eftersom kommunikation är en tidskrävande operation på en mobiltelefon har vi testat hur lång tid det tar att hämta en bild från en webbserver till en mobiltelefon. Vi testade tre olika bilder med storlekarna 87, 253 respektive 411 bytes. Vi utförde testerna på två olika mobiltelefoner. Dessa två var Siemens SL45i och Motorola Accompli 008. Vi mätte tiden genom att anropa metoden *System.getCurrentTime* direkt före och efter testkoden. Nedan ses resultatet av testerna. OpenCon motsvarar *Connector.open(url)*, OpenIn motsvarar *c.openInputStream()*, Read motsvarar *is.read(data)*, CloseIs motsvarar *is.close()* och CloseCon motsvarar *c.close()*.

Test nr	1	2	3	4	5	6	7	8	9	10	Medel
OpenCon	180	180	171	152	161	180	166	170	161	171	169
OpenIn	6217	7407	6193	6064	6198	6272	6350	6198	6354	6609	6386
Read	4	4	5	5	5	4	5	4	5	5	5
CloseIs	0	5	0	0	4	0	0	4	9	0	2
CloseCon	794	817	813	854	762	905	868	798	923	2316	985

Tabell 5.1: *Nedladdning av 87 bytes till Siemens SL45i.*

Tabell 5.1 visar hur lång tid olika operationer, vid nedladdning av 87 bytes, tar på en mobiltelefon av typen Siemens SL45i[34]. Alla tider är räknade i millisekunder.

Test nr	1	2	3	4	5	6	7	8	9	10	Medel
OpenCon	152	161	166	157	166	157	166	162	175	162	162
OpenIn	9073	10462	7541	9475	7075	6987	9447	7094	7264	10721	8514
Read	5	0	5	4	5	9	4	4	4	9	5
CloseIs	5	0	0	0	0	0	0	5	9	0	2
CloseCon	821	808	867	822	854	821	803	826	923	896	844

Tabell 5.2: *Nedladdning av 253 bytes till Siemens SL45i.*

Tabell 5.2 visar hur lång tid olika operationer, vid nedladdning av 253 bytes, tar på en mobiltelefon av typen Siemens SL45i[34]. Alla tider är räknade i millisekunder.

Test nr	1	2	3	4	5	6	7	8	9	10	Medel
OpenCon	213	171	189	152	153	198	185	162	147	171	174
OpenIn	22549	20126	15529	20131	14865	15049	17948	14805	15012	17938	17395
Read	4	9	9	19	9	9	5	9	33	9	12
CloseIs	0	4	0	0	0	5	5	0	10	0	2
CloseCon	798	859	863	734	776	840	844	895	844	886	834

Tabell 5.3: *Nedladdning av 411 bytes till Siemens SL45i.*

Tabell 5.3 visar hur lång tid olika operationer, vid nedladdning av 411 bytes, tar på en mobiltelefon av typen Siemens SL45i[34]. Alla tider är räknade i millisekunder.

Test nr	1	2	3	4	5	6	7	8	9	10	Medel
OpenCon	51	50	50	51	50	51	50	50	50	51	50
OpenIn	8655	7242	11134	8010	8240	8060	8869	8019	7857	12973	8906
Read	4	4	3	4	4	4	4	3	4	4	4
CloseIs	1	0	1	0	1	1	1	1	1	1	1
CloseCon	229	227	228	227	227	227	228	228	229	228	228

Tabell 5.4: *Nedladdning av 87 bytes till Motorola Accompli008.*

Tabell 5.4 visar hur lång tid olika operationer, vid nedladdning av 87 bytes, tar på en mobiltelefon av typen Motorola Accompli008[25]. Alla tider är räknade i millisekunder.

Test nr	1	2	3	4	5	6	7	8	9	10	Medel
OpenCon	50	51	50	50	51	51	50	51	50	50	50
OpenIn	7099	7579	9417	7786	6877	7343	7856	6681	7204	9035	7688
Read	6	5	5	5	5	5	5	5	5	5	5
CloseIs	1	1	1	1	1	1	0	1	1	1	1
CloseCon	229	227	228	227	227	227	229	227	227	228	228

Tabell 5.5: *Nedladdning av 253 bytes till Motorola Accompli008.*

Tabell 5.5 visar hur lång tid olika operationer, vid nedladdning av 253 bytes, tar på en mobiltelefon av typen Motorola Accompli008[25]. Alla tider är räknade i millisekunder.

Test nr	1	2	3	4	5	6	7	8	9	10	Medel
OpenCon	51	51	50	52	78	51	51	51	51	78	56
OpenIn	11722	10840	11473	12426	16276	12272	12484	12893	12546	12071	12500
Read	6	6	117	5	5	5	5	117	6	5	28
CloseIs	1	1	1	1	1	1	1	1	1	1	1
CloseCon	229	227	228	227	227	228	228	227	228	228	228

Tabell 5.6: *Nedladdning av 411 bytes till Motorola Accompli008.*

Tabell 5.6 visar hur lång tid olika operationer, vid nedladdning av 411 bytes, tar på en mobiltelefon av typen Motorola Accompli008[25]. Alla tider är räknade i millisekunder.

Vi ser i tabellerna 5.1 till 5.6 att den tidskrävande operationen är att öppna en *InputStream*. De andra operationerna går relativt snabbt. Den sammanlagda tiden för att ladda hem något är dock stor, och det är tveksamt om användaren är villig att vänta upp emot 10 sekunder för så små datamängder. Om man lägger nedladdningen i en bakgrundstråd kan man sysselsätta användaren med något vilket är att föredra.

5.2 Sessioner

Om man vill ha en varaktig förbindelse med en server kan det vara ett problem eftersom HTTP inte stöder detta. Man kan lösa detta på flera sätt. I detta avsnitt undersöker vi hur man kan införa sessioner i en MIDlet. Skälet till att man vill ha en varaktig förbindelse är för att serverapplikationen ska kunna spara information som är speciell för den applikation som exekveras i mobiltelefonen. M h a en session kan man veta att det är samma applikation som fortfarande kommunicerar med servern. När applikationen ska avslutas kan man skicka ett avsluta-meddelade som talar om för servern att man avslutar sessionen.

Om man med hjälp av en servlet vill utföra en sökning i en databas, bör man spara resultatet, vilket gör att man inte behöver göra sökningen mer än en gång Detta är minneskrävande men eftersom det sker i en server är det inte något problem. Detta kan man göra m h a en s k session som det finns stöd för i J2EE-API:t. Servleten håller då koll på sessionen m h a ett sessionsid. När klientapplikationen anropar servern skapas en session och servern skickar tillbaka sessionsnumret som en textsträng till klienten i en header i ett HTTP-svar. Strängen är en s k cookie. En cookie är en textsträng med information som skickas till klienten. I webbrowsrar sparas informationen på datorn för att kunna användas nästa gång

användaren accessar webbsidan. För varje ny förfrågan skickar klienten med cookien som en del av frågan.

Så här kan serverns HTTP-svar se ut:

HTTP/1.1 200 OK

Content-Type: text/plain

Content-Length: 53

Date: Tue, 18 Dec 2001 17:19:22 GMT

Server: Apache Tomcat/4.0.1 (HTTP/1.1 Connector)

Set-Cookie:

JSESSIONID=35E2621570C3B1D052E86285D1A002D6;Path=/midp

För att få koden mer generaliserad och robust bör man ha med följande i klientapplikationen:

Tillåt möjligheten att servern skickar flera cookies till klienten. I cookien kan det finnas information om vilken serverapplikation som skickar cookien och hur länge ett visst paket får vara på väg innan det blir för gammalt. Man undersöker en header genom att använda metoderna *getHeaderFieldKey* och *getHeaderField* som finns i interfacet *HttpConnection*. Det minsta man behöver spara är sessionsid:t som finns i variabeln JSESSIONID. Detta gör att servern kan veta vilken MIDletapplikation den kommunicerar med.

Om man inte kan använda sig av cookies för att hålla reda på sessionen, t ex om browsern inte tillåter detta, kan man använda sig av en metod som kallas för URL-rewriting. Normalt när man vill accessa en server skriver man så här (utan URL-rewriting):

http://www.somesite.com/servlet/shop/catalog.html

Om man använder sig av URL-rewriting kan det se ut så här:

http://www.somesite.com/servlet/shop/catalog.html;jsessionid=cl98373673At

Här lägger klienten till sitt sessionsid i slutet på URL:en. Om man inte skickar med ett sessionsid vet inte servern/klienten om man fortfarande är på samma session eller om en ny session påbörjats. Om man börjar på en ny session så är oftast inte gammal information tillgänglig. Detta beror dock på hur servern är implementerad.

Källa: [17], [20]

5.2.1 Analys

Cookies stöds inte i alla plattformar varför det kan vara bättre att använda sig av URL-rewriting.

6 Exekvering

I detta kapitel går vi igenom parallella processer i avsnitt 6.1 där vi undersöker hur man skall dela upp sin applikation i olika trådar. I avsnitt 6.2 och 6.3 går vi igenom vad som är viktigt att tänka på vid läsning och skrivning mot olika externa enheter. Hos referenserna som är upplistade i slutet av varje avsnitt finns information om det som tas upp i avsnittet. Hos de referenser som står inuti ett avsnitt finns information som endast tar upp den specifika delen.

6.1 Synkronisering och trådar

Synkronisering används för att ingen annan tråd skall kunna komma åt delade data medan den håller på att uppdateras av en annan tråd. Här undersöker vi hur synkronisering kan effektiviseras. Vi tittar också på när man bör använda trådar.

6.1.1 Rekommendationer

En vanlig regel är att om en operation tar längre tid än en tiondels sekund skall den göras i en tråd så att operationen inte blockerar användargränssnittet. Användargränssnittets svarstider är extremt viktiga på en mobiltelefon. Om man använder trådar är det viktigt att skydda delad data under en uppdatering. Detta kan göras med nyckelordet *synchronized*, men oftast innebär detta att det blir mycket overhead. Klasserna *Vector* och *HashTable* använder sig av synkronisering. Om man kan se till att enbart en tråd i taget kan ändra på datan behöver man inte använda sig av dessa klasser. Försök att minimera trådarnas skapande/förstörande cykler, genom att inte döda en tråd om man kan behöva den senare. Sätt prioritet på trådarna så att de viktigaste alltid får företräde.

Det går normalt snabbare att anropa en synkroniserad metod än ett synkroniserat block på grund av Javas implementation, det blir mer overhead när man synkroniserar block än metoder. Här är ett exempel på ett synkroniserat block:

```
synchronized(this)  //this syftar på den aktuella klassen
{
        System.out.println("Jag är synkroniserad");
}
```

I en MIDlet-applikation skriver man kod som anropas av systemet. Tråden exekveras då i en systemtråd. En sytemtråd är den tråd som applikationen exekveras i när man startar applikationen. När MIDlet:ens *startApp*, *pauseApp* och *destroyApp*-metoder anropas, körs dessa i en systemtråd. *startApp* används för att starta applikationen När man startar applikationen anropas *startApp* av systemet. Metoden anropas också när man ska fortsätta exekvera applikationen efter att metoden *pauseApp* anropats. *pauseApp* anropas när applikationen tillfälligt ska stoppas och *destroyApp* anropas när applikationen avslutas. Den händelsehanterande metoden, *commandAction* från interfacet *CommandListener* körs också inuti en systemtråd. Eftersom man implementerar interfacet *CommandListener* i sin huvudklass kommer metoden *commandAction* att köras i en systemtråd. Man bör därför minimera koden i denna metod så att metoden kan returnera så snabbt som möjligt för att möjliggöra att andra viktiga systemmetoder kan anropas. Systemtrådarna ska kunna fortsätta utföra viktiga systemhändelser som t ex att rita om fönster, utan att fastna i vänteläge medan en förbindelse kopplas upp.

Källor: [13], [14], [15], [21], [29], [32], [33], [37]

6.1.2 Analys

Man bör alltid använda separata trådar för applikationens logik, det grafiska användargränssnittet och serverkommunikationen. Detta eftersom kommunikation med en server leder till fördröjningar och för att det skall vara lättare att ändra i användargränssnittet.

6.1.3 Tester

Vi ville testa om det fanns någon begränsning av antalet trådar som kunde exekvera samtidigt. Vi testade detta på en Motorola Accompli 008 [25]. Vår test visade att det inte fanns någon begränsning mer än att varje tråd tar upp minne. Detta gör att det är minnet som begränsar hur många trådar som kan exekvera parallellt.

6.2 Input / Output

Det är ofta mycket svårt att veta hur lång tid det tar att utföra olika I/O-operationer eftersom det är beroende på vilken plattform applikationen körs. Vissa mobiltelefoner kan ha

relativt kort accesstid, medan andra har långsam accesstid. De råd som finns i detta avsnitt är dock mer generella och syftar till att effektivisera exekveringen.

6.2.1 Rekommendationer

Använd buffrande I/O-klasser, om det finns. Undvik onödiga I/O-operationer till skärm och minne. Filinformation kräver systeminformation. All I/O bör göras i bakgrundstrådar för att inte låsa applikationen för t ex användarinteraktivitet. Detta kan man göra m h a interfacet *Runnable* som används för att skapa en tråd. Släpp tagna resurser, såsom http-förbindelser och filer, så tidigt som möjligt. Detta gör att minne som tagits i anspråk av resursen frigörs. Om det inte går att undvika fördröjningar, se till att användaren ser vad som händer, t ex genom att skriva ut en statusruta på skärmen.

Källor: [2], [4], [13], [21], [22], [29]

6.2.2 Analys

Det tar extra resurser att skapa och hålla igång en extra tråd för den virtuella maskinen vilket gör att det inte alltid är lönsamt att skapa en ny tråd för varje I/O-tillfälle. Om den I/O-operation som sker går snabbt behöver man inte starta en ny tråd för detta.

6.3 Record Management System - RMS

I mobiltelefoner med beständigt minne, d v s minne som behåller informationen även om applikationen har avslutats som t ex en hårddisk, kan man spara information mellan användningar av en viss applikation. Det finns därför ett antal klasser som hjälper till med detta. I det här avsnittet undersöker vi hur man kan använda sig av dessa på ett effektivt sätt.

6.3.1 Rekommendationer

RMS, Record Management System, är en samling klasser för att spara/hämta data från mobiltelefonens minne. Man läser/skriver alltid in hela minnesregister, genom att använda klassen *RecordStore*:s *getRecord-*, *addRecord-* och *setRecord*-metoder. Minimera antalet läsningar/skrivningar till minnet. I de flesta mobiler tar det längre tid att skriva till minnet än att läsa från det, därför är det extra viktigt att undvika många skrivningar.

Även om man minimerar antalet skrivningar finns det en risk att läsningar blir en flaskhals i applikationen. Detta beror på att man skapar många objekt som bara används en kort tid och detta kan leda till att skräpsamlaren inte hinner med.

Två dåliga sätt att gå igenom ett record är:

```
// Öppna om det finns en record store annars skapa
RecordStore rs = openRecordStore("Namn",true);
try
{
   //Hämtar id för nästa record som skall läggas till
   int lastID = rs.getNextRecordID();
   byte[] data;
   for(int i=0; I<lastID; I++)
   {
        try
        {
                  data = rs.getRecord(i); // Hämtar sparad data
        }
         catch(InvalidRecordIDException  e)
         {
                  continue;
         }
   }
}
catch(Exception e){…}

RecordStore rs = openRecordStore("Namn",true);
try
{
  // Returnerar en enumeration för att gå igenom ett antal records
  RecordEnumeration enum = rs.enumerateRecords(null,null,false);

  While(enum.hasNextElement())
  {
    byte[] data =enum.nextRecord();
    …
  }
}
catch(Exception e){…}
```

Problemet med ovanstående kodsnuttar är att de skapar nya arrayobjekt för varje record. Ett bättre sätt är:

```
RecordStore rs = openRecordStore("Namn",true);
try
{
    RecordEnumeration enum = rs.enumerateRecords(null,null,false);
    byte[] data = new byte[100];
    int len = 0;
    while(enunm.hasNextElement() )
    {
        int id = enum.nextRecordID();
        len = rs.getRecordSize(id);
        if(len >data.length)
        {
            data = new byte[len + EXTRAUTRYMME];
        }
        // Här sparas record I dataarrayen
            rs.getRecord(id, data, 0);
            …
        }
}
catch(Exception e){…}
```

I det sista exemplet bestämmer man redan från början storleken på arrayen. Detta gör att man inte behöver skapa en ny array med rätt storlek varje gång, utan kan återanvända samma byte-array för alla *Records*, om den inte är för liten, då byter man ut arrayen mot en som är större.

Källor: [6]

6.3.2 Analys

Försök att undvika läsning av och skrivning till minnet så mycket som möjligt. Läs in det man behöver i början och vänta med att skriva saker till minnet så länge som applikationen kan modifiera datan. Detta ökar exekveringshastigheten. Problemet är att man måste läsa in mer information i RAM-minnet.

Det man kan göra är att testa hur mycket minne som man behöver läsa in och hur ofta man behöver accessa minnet.

6.3.3 Tester

Vi har testat hur lång tid det tar använda RMS. Det vi testade var att skriva och läsa och ta bort 10, 100 och 1000 byte i en *RecordStore*. Vi har också testat hur lång tid det tar att skapa, öppna, stänga samt ta bort en *RecordStore*. Det fanns ingen data i den RecordStore som vi tog bort. Tiden mätte vi med metoden *System.currentTimeMillis*. Vi anropade denna metod före och efter testkoden för att få tidsåtgången. Vi testade på både en Siemens SL45i, [34] och en Motorola Accompli 008, [25].

Test nr	1	2	3	4	5	6	7	8	9	10	Medelvärde
Write 500 bytes	281	323	300	314	314	307	287	328	323	332	310,9
Write 100 bytes	88	88	84	89	83	89	102	88	87	79	87,7
Write 10 byes	78	74	74	70	79	74	74	73	73	74	74,3
Read 500 bytes	190	171	189	171	171	189	188	175	176	184	180,4
Read 100 bytes	69	60	55	60	60	56	60	60	63	56	59,9
Read 10 bytes	42	41	42	42	42	41	41	42	41	42	41,6
Del 500 bytes	263	240	249	272	254	253	268	254	236	240	252,9
Del 100 bytes	65	64	67	65	70	69	69	66	69	69	67,3
Del 10 bytes	41	42	51	37	36	41	37	37	40	42	40,4
Create Record Store	2188	2160	2220	2183	2239	2169	2183	2141	2179	2164	2182,6
Open Record Store	402	406	411	416	411	415	416	410	420	411	411,8
Close Record Store	120	120	120	88	129	102	101	120	129	115	114,4
Delete Record Store	715	706	702	747	711	724	711	748	724	747	723,5

***Tabell 6.1**: Tidsåtgång vid användande av RMS med Siemens SL45i.*

Tabell 6.1 visar hur lång tid olika minnesoperationer tar på en mobiltelefon av typen Siemens SL45i, [34]. Alla tider är räknade i millisekunder.

Vi ser i tabell 6.1 att det tar lång tid att skapa eller radera en *RecordStore*. Att skapa en *RecordStore* är att jämföra med att skapa en mapp på en dator. Det är alltså inte endast att skapa objektet *RecordStore* som tar tid, utan den största tiden går åt till att skapa en *RecordStor*e i det beständiga minnet. Att öppna och stänga en *RecordStore* är betydligt snabbare, men ändå tidskrävande. Därför skall man minimera antalet g g r som man behöver öppna eller stänga en *RecordStore*. Att öppna och stänga en *RecordStore* händer betydligt oftare än att skapa och radera denna, vilket innebär att det är tidsåtgången för att öppna och stänga fördröjningar som man oftast kommer att få.

Ur tidssynpunkt är det alltså bättre att skriva all data i en applikation till en enda RecordStore än att skapa många *RecordStore*s. Att skapa många *RecordStore*s ger dock bättre struktur, t ex bilder i en *RecordStore*, värden i en annan etc. Det kan också bli svårt att hålla reda på datan om allt ligger i samma *RecordStore*.

Desto mer bytes man skriver, läser eller raderar, desto längre tid tar det. Det går dock snabbare att läsa, skriva eller radera 500 bytes, än att radera 100 bytes 5 ggr eller 10 bytes 50 ggr. Det är alltså bra att operera på så stora datamängder som möjligt.

Test nr	1	2	3	4	5	6	7	8	9	10	Medelvärde
Write 1000 bytes	1647	1616	1515	1682	1577	1578	1573	1574	1673	1532	1596,7
Write 100 bytes	1546	1606	1497	1481	1576	1578	1572	1574	1481	1559	1547,1
Write 10 byes	1541	1612	1522	1477	1571	1572	1568	1569	1476	1478	1538,6
Read 1000 bytes	21	21	22	21	23	24	21	22	21	21	21,7
Read 100 bytes	21	21	21	20	20	23	21	20	20	20	20,7
Read 10 bytes	20	20	20	20	20	20	20	21	20	20	20,1
Del 1000 bytes	291	285	181	177	243	243	283	207	176	258	234,4
Del 100 bytes	294	283	174	176	241	108	282	202	175	240	217,5
Del 10 bytes	181	184	163	161	256	261	171	168	164	228	193,7
Create Record Store	3273	3649	4218	3148	3245	3340	3269	3277	3148	3120	3368,7
Open Record Store	1443	1499	1477	1466	1469	1472	1423	1418	1466	1468	1460,1
Close Record Store	4	3	4	4	4	4	4	4	4	4	3,9
Delete Record Store	131	121	93	103	110	131	121	99	94	113	111,6

Tabell 6.2*: Tidsåtgång vid användande av RMS med Motorola Accompli 008.*

Tabell 6.2 visar hur lång tid olika minnesoperationer tar på en mobiltelefon av typen Motorola Accompli 008, [25]. Alla tider är räknade i millisekunder.

Motorola Accompli 008 tar betydligt längre tid på sig att hantera en *RecordStore* i de flesta avseenden. Speciellt att skriva till minnet tar lång tid jämfört med Siemens SL54i. De operationer som Motorolan utför snabbare är läsning, borttagning och stängning av en *RecordStore*.

Vi har också jämfört hur lång tid det tar att skapa en bild genom att hämta bilden från en webserver, genom att läsa in bildens data från mobiltelefonens beständiga minne, d v s från en *RecordStore,* och genom att skicka med bilden i jar-filen. Vi testade också hur lång tid det tog att skriva den nedladdade bilden till en RecordStore. I tiden för att läsa från en RecordStore är inte tiden för att öppna *RecordStore*:en medräknad. Denna tid finns i tabellerna 8.1 och 8.2. Nedan ses resultatet av jämförelsen.

Test nr	1	2	3	4	5	6	7	8	9	10	Medelvärde
Download	16938	16766	19516	18649	17002	15931	16356	16416	16568	16078	17022
Write	139	120	138	129	144	138	124	111	134	138	132
Read	730	720	715	720	725	720	720	738	743	710	724
FromJar	835	821	813	803	799	803	817	790	794	816	809
Download, Write and Read	17807	17606	20369	19498	17871	16769	17200	17264	17445	16826	17878

Tabell 6.3: *Tidsåtgång när data hämtas och bild skapas av datan på Siemens SL45i.*

Tabell 6.3 visar hur lång tid det tar att hämta 411 bytes från olika lagringsutrymmen och skapa en bild av datan på en mobiltelefon av typen Siemens SL45i[34]. Den visar också hur lång tid det tar att spara den nedladdade datan i en RecordStore och den sammanlagda tiden för att ladda ned data, skriva datan till en RecordStore och sen läsa datan från en RecordStore. Alla tider är räknade i millisekunder.

Test nr	1	2	3	4	5	6	7	8	9	10	Medelvärde
Download	12354	12712	11853	11849	11978	12320	12192	12268	11956	12144	12163
Write	2851	2888	2918	2928	3063	3069	4546	3010	3034	4601	3291
Read	273	273	275	275	276	275	276	276	276	276	275
FromJar	312	326	326	327	326	326	326	327	326	326	325
Download, Write and Read	15478	15873	15046	15052	15317	15664	17014	15554	15266	17021	15729

Tabell 6.4: *Tidsåtgång när data hämtas och bild skapas av datan på Siemens SL45i.*

Tabell 6.4 visar hur lång tid det tar att hämta 411 bytes från olika lagringsutrymmen och skapa en bild av datan på en mobiltelefon av typen Motorola Accompli 008[25]. Den visar också hur lång tid det tar att spara den nedladdade datan i en RecordStore och den sammanlagda tiden för att ladda ned skriva till en RecordStore och sen läsa från en RecordStore. Alla tider är räknade i millisekunder.

Test nr	1	2	3	4	5	6	7	8	9	10	Medelvärde
Download	13185	12217	12617	12625	12088	12856	12331	12735	13050	12221	12593
Write	2732	4250	2838	2730	2868	4343	2703	2838	4372	4323	3400
Read	401	402	402	401	401	402	401	401	401	401	401
JarImage	455	429	455	429	429	429	458	429	429	428	437
Download, Write and Read	16318	16869	15857	15756	15357	17601	15435	15974	17823	16945	16394

Tabell 6.5: *Tidsåtgång när data hämtas och bild skapas av datan på Motorola Accompli 008.*

Tabell 6.5 visar hur lång tid det tar att hämta 1192 bytes från olika lagringsutrymmen och skapa en bild av datan på en mobiltelefon av typen Motorola Accompli 008[25]. Den visar också hur lång tid det tar att spara den nedladdade datan i en RecordStore och den sammanlagda tiden för att ladda ned data skriva datan till en RecordStore och sen läsa datan från en RecordStore. Alla tider är räknade i millisekunder.

Det var endast på Motorola Accompli 008 som det gick att läsa 1192 bytes från det beständiga minnet. Därför är det endast på denna telefon som vi mätt tider för att hämta en så stor bild.

Vi ser i tabellerna 6.3 till 6.5 att det snabbaste sättet att skapa en bild är att läsa den från en RecordStore. Det förutsätter dock att RecordStore är öppen. Att öppna en RecordStore tar lite tid, så om endast en bild skall laddas, är det snabbaste sättet att ladda datan från jar-filen. Om man har många bilder i en RecordStore är detta dock ett bra sätt att hantera bilderna. Att ladda hem en bild tar mycket längre tid än att läsa data från en RecordStore eller jar-filen. Det är inte att föredra att ladda ner en bild varje gång den skall användas. Man kan dock ladda hem bilderna en gång och sedan spara dem i en RecordStore. Då får man bara den långa tidsfördröjningen, som nedladdning från en webserver innebär, första gången man behöver bilderna. Denna tid återfinns på den sista raden i varja tabell.

7 Resultat och rekommendationer

Innan vi börjat arbetet med att undersöka MIDP för mobiltelefoner, tänkte vi oss mobiltelefoner främst som telefoner. Att några mer avancerade applikationer skulle fungera på en mobiltelefon verkade osannolikt. Men nu i efterhand vet vi att det var fel. Skrivs koden på ett optimerat sätt finns det många användbara applikationer som kan fungera på en mobiltelefon. Den största begränsningen, som vi ser det, är den långsamma kommunikationen, och att interaktionen mellan användare och mobiltelefon är väldigt tidskrävande och besvärlig. Den lilla skärmen och de små knapparna är ett mycket sämre och långsammare interaktionsmedel än en dators bildskärm och tangentbord. Vi har upplevt att denna skillnad är större än skillnaden i prestanda vilken också är stor. Även prestandan har inneburit problem. T ex har väntetiderna varit långa och programkrasher p g a att minnet tagit slut har varit vanliga. Det har dock funnits flera sätt för oss att lösa dessa problem.

När man använder rekommendationerna i denna rapport skall man vara medveten om att vissa rekommendationer har både positiva och negativa effekter. En vanlig negativ effekt är att koden blir svårare att förstå. Några rekommendationer som gör koden mer komplex är, använd shiftoperatorerna, använd publika klassmedlemmar, använd ej arv, använd arrayer istället för matriser och initiera ej variabler. Det finns också rekommendationer som står i konflikt med andra rekommendationer. Ett exempel på detta kan vara att applikationen blir snabbare men tar mer minne i anspråk, eller att tidsåtgången utökas men att man får en vinst inom något annat område. Några exempel är om man använder *int* istället för *byte* eftersom *int* är snabbare att arbeta med men tar upp mer minne eller om man skapar nya lagringsklasser istället för att rensa och återanvända gammla.

Vissa rekommendationer har positiva effekter på mer än ett område. Det är främst rekommendationer som gäller tidsoptimering och minnesoptimering som har synergieffekter. Några av dessa rekommendationer är, använd arrayer istället för matriser och använd char-arrayer istället för *String/StringBuffer*.

Många rekommendationer som nämnts har bara positiva effekter. Dessa är bra att kunna. Några av dessa är, använd inte wrapperklasser och skapa inte ett objekt om man endast skall anropa statiska metoder.

I detta kapitel är de resultat och rekommendationer som finns beskrivna i denna uppsats. Vi har delat upp dem i fem kategorier. Dessa är design, användbarhet, minnesoptimering, tidsoptimering och storlek på jarfilen. I avsnittet om tidsoptimering finns rekommendationer som kan göra en MIDlet snabbare. Under design finns rekommendationer för hur man skall

dela upp koden i olika klasser. Användbarhet handlar om att en applikation skall vara lätt att förstå och uppfattas som snabb ur användarens synpunkt. Minnesoptimering handlar om sätt att utnyttja så lite minne som möjligt. Tidsoptimering innehåller rekommendationer som får applikationen att exekvera snabbare. Storlek på jar-filen handlar just om sätt att minska storleken på jar-filen. Avsnitten om tid och minne är uppdelad i rekommendationer man alltid kan göra och rekommendationer som kan vara bättre att göra när applikationen är klar, bl a för att koden blir svårare att förstå. Rekommendationerna här är kortfattade. Mer information finns i de avsnitt som anges.

7.1 Design

- Separera GUI-klasser från applikationens logik, se avsnitt 3.16.1
- Använd högnivå-API:et om placering av bilder och text ej är viktig, se avsnitt 4.1.1
- Använd högnivå-API:et om man vill vara säker på att applikationen ska fungerar på alla mobiltelefoner, se avsnitt 4.2.1
- Använd lågnivå-API:et om placering av bilder och text är viktig och om skärmen skall uppdateras ofta, se avsnitt 4.2.1, 4.5

7.2 Användbarhet

- Gör nedladdningar och läs och skrivoperationer till RMS i bakgrundstrådar, se avsnitt 3.10.1, 5.1, 6.2.1
- Vid fördröjningar, meddela användaren vad som händer, se avsnitt 3.10.1, 4.4.2, 6.1
- Återanvänd gränssnitt, se avsnitt 3.16.1
- Liknande operationer skall utföras på liknande sätt, se avsnitt 4.4.1
- Användaren ska kontroll över programmet, inte tvärt om, se avsnitt 4.4.1
- Bygg menyerna i en hierakisk struktur, se avsnitt 4.4.1
- Använd Double Buffering, se avsnitt 4.5
- Om en operation tar mer än en tiondels sekund, utför den i en egen tråd, se avsnitt 6.1.1

7.3 Minnesoptimering

Dessa optimeringar kan man alltid göra:

- Sätt färdiganvända objektreferenser till null, se avsnitt 3.2.1
- Skapa inte ett objekt om man endast skall anropa statiska metoder, se avsnitt 3.2.1
- Använd ej inre klasser, se avsnitt 3.2.1
- Använd inte wrapper-klasser, se avsnitt 3.4.1
- Undvik upprepade metodanrop och upprepade skapande av objekt inuti en loop, se avsnitt 3.8.1
- Släpp tagna resurser, se avsnitt 6.2.1
- Undvik undantag, se avsnitt 3.11.1
- Använd StringBuffer istället för String om innehållet skall ändras/läggas till, se avsnitt 3.14.1

Dessa optimeringar bör göras om de behövs, efter att applikationen är fungerande, eftersom de gör det svårare att läsa och förstå koden:

- Skapa inte nya objekt, återanvänd gamla istället, se avsnitt 3.2.1
- Använd inlining, se avsnitt 3.6.1
- Använd arrayer istället för matriser, se avsnitt 3.13.1
- Använd char[] istället för String/StringBuffer, se avsnitt 3.14.1
- Använd byte istället för short, int, long för lagring av tal runt 0, se avsnitt 6.1.1
- Använd ej *synchronized*, se avsnitt 6.3.1
- Återanvänd arrayer, se avsnitt 6.3.1
- Använd inte rekursion om ej kompilatorn kan optimera rekursion till en loop, se avsnitt 3.13.1

7.4 Tidsoptimering

Dessa optimeringar kan man alltid göra:

- Använd *System.arraycopy* vid kopiering av arrayer, se avsnitt 3.14.1
- Initiera ej variabler som initieras automatiskt, se avsnitt 3.15.1
- Ta bort redundant kod, se avsnitt 3.16.1, 3.18.1
- Skicka sammansatt information över nätverksförbindelser, se avsnitt 5.1
- Använd synkroniserade metoder istället för synkroniserade block, se avsnitt 6.1.1
- Minimera skrivning till RMS, se avsnitt 6.3.1
- Använd int som loopräknare, se avsnitt 3.4.1, 3.8.1, 3.12.1

- Undvik undantag, se avsnitt 3.10.1, 3.18.1

Dessa optimeringar bör göras om de behövs, efter att applikationen är fungerande, eftersom de gör det svårare att läsa och förstå koden:

- Använd arrayer istället för länkade listor, se avsnitt 3.13.1
- Använd arrayer istället för matriser, se avsnitt 3.13.1
- Använd char-arrayer istället för String/StringBuffer, se avsnitt 3.14.1
- Använd lokala variabler istället för klassmedlemmar, se avsnitt 3.15.1
- Använd metoder av typen *System.metod* om möjligt, se avsnitt 3.16.1
- Använd short-circuit vid AND- och OR-villkor, se avsnitt 3.16.1
- Ta bort onödiga finesser, t ex animationer vid nedladdning, se avsnitt 3.16.1
- Utför beräkningsintensiva processer på en server, se avsnitt 5.1
- Återanvänd arrayer, se avsnitt 6.3.1
- Minimera konstruktorer, se avsnitt 3.2.1
- Använd ej djupa arvshierarkier, se avsnitt 3.2.1, 3.5.1
- Skapa objekt vid ”bra” tillfällen, se avsnitt 3.2.1
- Skapa objektspecifika klasser istället för att använda de generella klasser som finns, se avsnitt 3.3.1
- Använd inte wrapper-klasser, se avsnitt 3.4.1
- Undvik att omvandla mellan klasser och interface, se avsnitt 3.5.1
- Använd inlining, se avsnitt 3.6.1
- Förutbestäm generella lagringsklassers storlek, se avsnitt 3.7.1, 3.14.2
- Om ett lagringsobjekt inehåller många element, skapa ett nytt objekt istället för att tömma det gamla, se avsnitt 3.7.1
- Undvik upprepade metodanrop och upprepat skapande av nya objekt i loopar, se avsnitt 3.8.1
- Jämför med 0 i loopar, se avsnitt 3.8.1
- Gör klassmedlemmar publika, om inlining inte utförs av kompilatorn, istället för åtkomst m h a en metod, se avsnitt 3.9.1, 3.15.1
- Deklarera metoder som *final* om de inte skall ärvas, se avsnitt 3.9.1
- Använd switch-satser istället för if-else, se avsnitt 3.11.1
- Använd a += b istället för a = a + b, se avsnitt 3.11
- Använd shiftoperatorerna vid multiplikation av tal med formen 2^n, se avsnitt 3.11.1

- Lagra resultat i en lokal variabel och använd den i multipla beräkningar, se avsnitt 3.12.1
- Använd inte rekursion, se avsnitt 3.13.1

7.5 Storleken på jarfilen

- Lägg instruktioner i en egen applikation som är valfri att ladda ned, se avsnitt 3.16.1
- Obfuscera, se avsnitt 3.17.1
- Ladda ned rätt bildstorlek efter att applikationn startat istället för att lägga många bilder med olika storlekar i jar-filen , se avsnitt 5.1.2

8 Slutsatser

När vi började med arbetet visste vi inte riktigt hur en rapport skulle vara utformad, men nu tycker vi att vi har ett bra grepp om detta. Vår uppgift var sammansatt av olika områden vilket gjorde att vi kunde skriva om varje område direkt efter att vi utforskat det. Det innebar att vi kontinuerligt dokumenterade arbetet vi utförde under hela arbetstiden, istället för att spara dokumentationen till slutet, vilket gav ett normalt arbetstempo under hela arbetstiden. Vi arbetade inte mer under de sista veckorna än någon annan vecka. Vi har arbetat normala arbetsdagar, åtta timmar per dag, plus att vi har haft möten med handledaren under lediga timmar de dagar vi varit på föreläsningar på Karlstad universitet. Tidsåtgången för denna uppgift var vad vi hade förväntat oss vid arbetets start, vi följde tidsplanen.

Från början var det svårt att förutsäga vilka områden som skulle ta längst tid att utforska, p g a att Java-applikationer till mobiltelefoner är relativt nytt. Det visade sig att de flesta optimeringar som uppdagades var inriktade på tids- och minnesoptimering. Det är också inom dessa två områden vi har fått flest nya kunskaper. Många av dessa optimeringar är generella och fungerar således bra även vid andra projekt där en MIDlet-applikation inte är målet. Att interaktionen mellan användare och mobiltelefon är dålig p g a de små knapparna och de små svartvita skärmarna visste vi sedan tidigare då vi har skickat SMS och använt andra tjänster på vanliga mobiltelefoner. Dessa tjänster har dock varit anpassade för respektive mobiltelefon, vilket inte en MIDlet alltid är. Att inte en applikation är anpassad för en viss telefon leder således till en försämring av de redan undermåliga interaktionssätten.

Det som har resulterat i mest frustration är att vissa applikationer som fungerat bra i en emulator, inte fungerar bra och i vissa lägen inte alls, på mobiltelefonerna. Vad som gjort att applikationerna inte fungerat har varit extremt svårt att finna och har tagit lång tid att undersöka.

Att lära sig att utveckla MIDlets har varit lätt, eftersom vi gått en kurs i Java-programmering. I denna kurs använde vi J2SE, men de många likheterna med J2ME gjorde att de kunskaper vi fått under kursen underlättade studierna av MIDlet-programmering.

Vi är i stort nöjda med hur vi utfört uppgiften och skulle inte ändra så mycket om vi skulle göra om den. Vi skulle dock ha lagt ned lite mer arbete på att ta reda på hur en uppsats skall vara uppbygd, vilket skulle resulterat i mindre jobb med att ändra på det vi skrivit. Efter ett

tag skapade vi en applikation där vi kunde testa tids- och minnesåtgång för olika kodexempel. Hade vi gjort det från början hade vi sparat lite tid och hunnit utföra flera tester.

Den viktigaste lärdomen vi fått är att vi fått se hur man arbetar på ett företag. Detta har varit mycket lärorikt och vi förstår nu varför vi fått lära oss många av de saker vi lärt oss på olika kurser. Arbetet har således gett oss en bättre helhetssyn vad gäller ämnet datavetenskap.

Referenser

[1] Optimazation, Jonathan Allin
http://www.symbian.com/books/wjsd/WJSD_Intro/wjsd-extract.html, 02-03-18

[2] "Atomic File Transactions.", Jonathan Amsterdam,
http://www.onjava.com/pub/a/onjava/2001/11/07/atomic.html, 02-02-11

[3] "Various performance tips", Asha Balasubramanyan,
http://www.nandighosha.org/forum/topic.asp, 02-02-11

[4] "Cutting Edge Java Game Programming", Neil Bartlett, The Coriolis Group, 02-02-11

[5] "Make Java fast: Optimize!", Doug Bell,
http://www.javaworld.com/javaworld/jw-04-1997/jw-04-optimize.html, 02-02-04

[6] The performance of games on J2ME, Jason R. Briggs,
http://www.javaworld.com/javaworld/jw-03-2001/jw-0309-games.html, 02-02-18

[7] CodeShield for Java – Product Specification, Codingart,
http://www.codingart.com/codeshield.html, 02-03-18

[8] "Recourses for software developers", Eric Giguere,
http://www.ericgiguere.com, 02-02-04

[9] "How to use InputConnection for raw input streams I & II", Eric Giguere
http://www.kvmworld.com/developer/fss/connections, 02-03-04

[10] "Mobile Information Device Profile for Java 2 Micro Editon", Eric Giguere, C. Enrique Ortiz,
http://wireless.java.sun.com/allchapters/, 02-03-04

[11] "Flicker-free graphics with the Mobile Information Device Profile", Eric Giguere
http://developer.java.sun.com/developer/J2METechTips/2001/tt0725.html, 02-02-18

[12] "Design for performance", Brian Goetz,
http://www.javaworld.com/javaworld/jw-03-2001/jw-0323-performance.html,
02-02-11

[13] "Java Optimizations", Jonathan Hardwick,
http://www.cs.cmu.edu/~jch/java/optimization.html, 02-02-04

[14] "HP-UX Programmer's Guide for Java", Hewlet Packard
http://www.unixsolutions.hp.com/products/java/perf.html, 02-02-11

[15] "Patrick Killelea's Java performance tips", Patrick Killelea,
http://www.patrick.net/jpt/index.html, 02-02-11

[16] "Networking, User Experience and Threads", Jonathan Knudsen,
http://wireless.java.sun.com/midp/articles/threading/, 02-02-25

[17] "Session Handling in MIDP", Jonathan Knudsen,
http://wireless.java.sun.com/midp/articles/sessions/, 02-02-25

[18] "Advanced MIDP networking, Accessing using sockets and RMI from MIDP-enabled Devices", Qusay Mahmoud,
http://wireless.java.sun.com/midp/articles/socketRMI/, 02-02-25

[19] "Invoking JavaServerPages from MIDlets", Qusay Mahmoud,
http://www.onjava.com/pub/a/onjava/2001/12/05/wirelessjava.html, 02-02-25

[20] "MIDP Inter-Communication with CGI and Servlets", Qusay Mahmoud,
http://wireless.java.sun.com/midp/articles/servlets/, 02-02-25

[21] "Programming strategies for Small Devices", Qusay Mahmoud
http://wireless.java.sun.com/midp/articles/ui/, 02-02-18

[22] "JavaTM Performance Tuning and Java Optimization Tips", Glen McCluskey,
http://www.glenmccl.com/jperf/, 02-02-11

[23] "Performance tuning", James McGovern,
http://www.sys-con.com/java/article.cfm, 02-02-11

[24] Motorola LWT, Motorola, 02-02-18

[25] Motorola Accompli 008, Motorola
http://www.motorola.se, 02-03-11

[26] "Method Description for Semla A Software Design Method with a Focus on Semantics", Martin Blom, Eivind J. Nordby, Anna Brunström, 02-03-04

[27] "Calculating Bytes in a MIDP Input Stream", Ed Ort,
http://wireless.java.sun.com/midp/questions/calcbyte/, 02-02-25

[28] "Wireless j2me platform programming", Vartan Piroumian, Alibris I.D., 02-03-18

[29] "Performance tuning report in German.", Sebastian Ritter,
http://www.bastie.de/resource/res/mjp.pdf och
http://www.bastie.de/java/mjperformance/contents.html, 02-02-11

[30] "Optimization", Jack Shirazi,
http://www.onjava.com/pub/a/onjava/2001/05/30/optimization.html, 02-02-04

[31] "Java Performance Tuning", Jack Shirazi,
http://www.oreilly.com/catalog/javapt/chapter/ch04.html, 02-02-04

[32] "Optimizing a Query on a Collection", Jack Shirazi,
http://java.oreilly.com/news/javaperf_0900.html, 02-02-04

[33] "Comparing techniques for performance tuning a query on a Map class ", Jack Shirazi,
http://www.javaworld.com/javaworld/jw-11-2000/jw-1117-optimize.html, 02-02-04

[34] Siemens SL45i API, Siemens, 02-02-18
http://www.siemens.se

[35] Siemens SL45i,Siemens,
http://www.siemens.se, 02-03-11

[36] "Serious Java Programming For The Wireless World", Shiuh-Lin Lee,
http://www.wirelessdevnet.com/channels/java/features/kvm.html, 02-02-18

[37] "An assortment of tips", Curt Smith,
http://www.ajug.org/jsl/javaperformance.html, 02-02-11

[38] "Object management article", Dennis M. Sosnoski,
http://www.javaworld.com/javaworld/jw-11-1999/jw-11-performance.html, 02-02-11

[39] MIDP API, Sun Microsystems, 02-02-18

[40] ”J2SE API”, Sun Microsystems”, http://java.sun.com/j2se/1.4/docs/api/index.html, 02-03-04

[41] J2EE API, Sun Microsystems, 02-03-11

[42] ”Java”, Sun Microsystems, http://java.sun.com, 02-03-04

[43] “SL45i_6688i_Whitepaper_41”, Sun Microsystems, 02-03-11

[44] “SMTK_UI%20Design%20guidelines_42”, Sun Microsystems, 02-03-11

[45] “Java 2 Platform, Micro Edition Datasheet”, Sun Microsystems, 02-03-18

[46] “The K Virtual Machine Datasheet”, Sun Microsystems, http://java.sun.com/products/cldc/ds, 02-03-18

[47] “Obfuscating Java Programming Language Code”, Paul Tyma, Godfrey Nolan, Paul Martino, http://industry.java.sun.com/javaone/99/abstracts_by_track/0,1666,8,00.html, 02-03-18

[48] Bilaga brev

A Specifikation för Examensarbete

Examensarbete – Optimerad MIDlet-design

A.1 Sammanfattning

Detta dokument beskriver examensarbetet Optimerad MIDlet-design vid Telia Mobile AB i Karlstad. Arbetet syftar till att undersöka vilka kriterier som är viktiga att tänka på när en MIDlet-applikation tas fram inom J2ME/MIDP. Arbetet är utredande men ändå av praktisk natur.

A.2 Grundläggande Information

A.2.1 Bakgrund ex-jobb/utredning

Under en relativt snar framtid kommer många av de mobila terminaler som släpps på marknaden att ha inbyggda virtuella Javamaskiner för J2ME. Detta gör det möjligt för användarna att hämta hem applikationer från Internet som exekveras lokalt i terminalen. Detta är en mycket intressant teknik med många möjligheter. Applikationerna kommer att kunna exekveras lokalt i telefonen och i viss mån utnyttja telefonifunktioner såsom att ringa samtal, skicka SMS, komma åt adressboken etc. Applikationerna kan också med hjälp av de mobila näten (GSM/GPRS/UMTS) ha Internetkoppling, d.v.s. de kan skicka och hämta data från Internet, dock med de begränsningar som de mobila näten och J2ME har.

Att mobila terminaler öppnas för tredjepartsapplikationer på detta sätt är nytt. Traditionella applikationsutvecklare är inte vana att göra program för ”klena” (liten processor, liten display, lite minne, små eller inga knappar…) terminaler som dessutom är anslutna till Internet via en ”dyr” och osäker förbindelse.

A.3 Mål

A.3.1 Ex-jobbets mål

Målet är att ta fram en guide/hjälpmedel, **”MIDP designguide for developers”**, som ska ta fram applikationer inom J2ME/MIDP (Java för små mobiltelefoner).

Guiden ska delas upp i olika kriterier där man ska beakta vissa saker. Kriterierna ska tas fram och prioriteras så att en utvecklare kan använda guiden för att dels se om det överhuvudtaget verkar troligt att det går att implementera en tänkt applikation, och dels se vad man ska tänka på för att få den så bra som möjligt.

Kriterierna:

- Källkod
  - Prestanda - olika sätt att optimera kod
  - Storlek – minsta möjliga källkod
  - Programstruktur – lokala/globala variabler, static etc.
  - Obfuscating
  - Kontrakt (pre post)
- GUI
  - Display
  - Användarinteraktion
  - Ljud
  - Generella applikationer som anpassar sig efter terminalens UI (mer kod) eller speciella applikationer för varje terminal (flera versioner av koden)?
- Kommunikation
  - Hur gör man applikationer som ”pratar” över Internet utan att det blir för dyrt etc. Ex: chat och mail-klinter
- Exekvering
  - Läs/skriv-operationer i minnet. Kapacitet?
  - Parallella processer. Ex: GUI & kommunikation
- HW

    - Olika terminaler har olika utseende, knappar, pekskärm, hjul osv. Hur hanterar man detta bäst (se punkt 4 under GUI)?
    - Prestanda hos befintliga och kommande terminaler.
- Applikation kontra nätprestanda (radio)
    - Roundtrip time. Hur snabbt är systemet?
    - Sessionsproblematik (servern vet inte om klienten lever)

Aktiviteter:

- Teoretiska studier av plattformen samt terminalspecifika utökningar.
- Implementera test-applikationer.
- Sammanfatta resultat i form av guidelines och tabeller som en del av slutrapporten.

A.3.2 Omfattning/avgränsning

Prioritering enligt tidsplanen.

A.3.3 Strategi

Vi arbetar med ett kriterie i taget. Sen får vi se hur många vi hinner med. Vi har som mål att hinna alla.

A.4 Rapportering

Handledaren hos TMAB kommer att finnas på plats i Karlstad och avrapportering kommer att ske löpande med ett avstämningsmöte en gång i veckan utöver spontana diskussioner. De dagar handledaren inte finns på plats finns annan personal tillgänglig för frågor.

Dessutom bör möten hållas regelbundet med alla inblandade, d.v.s. examensarbetarna, handledaren på TMAB samt universitets handledare.

A.5 Projektets avslutande

Rapport

Föredrag

- skola
- TMAB

Demo

Efter projektets avslutande och slutrapportens godkännande tar handledaren på TMAB hand om arkivering av genererad programkod och dokument för TMAB:s räkning.

B Brev

B.1 Fråga om GUI

My name is Per Davidsson and I am studying on the University of Karlstad (Sweden) to develope a userguide for MIDlet-programming. I´m wondering if you feel that the best way to develop an application is to do a diffrent for each platform (Siemens SL45i, Motorola A008 etc), or to develop an application that is addapting to the platform it is beeing downloaded to. I'm also wondering if I may publish your response in my essay?
/Per Davidsson

B.2 Svar från Midletsoft

Good questions. I would say that in the beginning, for stand-alone client-side MIDP products, Midletsoft developed for a general device audience, void of specific devices in mind. However, our products still had to be tested, and most of our initial products were tested on whatever was available (mainly Motorola's Accompli 008). However, as more J2ME enabled WIDs hit the market, the demand for specific applications for each device will grow; therefore, Midletsoft is now designing the applications with device type in mind, too.

Since WIDs are different than PCs, it's going to prove almost impossible to code once for all devices -- however, the beauty of Java is that it allows us to quickly change code to make our applications device specific. Solve that problem (come up with a flexible IDE), and you'll be a millionaire. (you have to share it with us, thought. ;)

Best to you and your school work! (you can use whatever you want from our site or from this e-mail for your school work)

David J. Stennett
Director of Operations

Midletsoft, LLC

B.3 Svar från Uppli

Thank you for contacting us.

Everytime it comes to multi-platform development, you have to face a trade-off that involves a few considerations:

1) what is the market size of a particual platform? Is it worth investing resources?
2) what are the enhanced features offered by proprietary APIs? Does my application need them?
3) does standard J2ME set of capabilities (wich represents only the “minimum common denominator”) satisfy the application’s requirements?

Proprietary APIs are likely to be used in games development in order to implement teatures like feedback (by using VibraCall), sounds, music and enhanced graphics. But Java Community Process is working in association with handset manufacturers to define an extended Game or Next Generation MIDP API in order to address these issues. On the other hand, most of business or PIM applications will continue using standard MIDP libraries.

I think that what will define the guidelines for future development will be the number of available Java handsets produced by manufacturer; if expectations about MIDP devices’ sales are met, we will see different versions of the same application for different platforms to deliver a better experience to the consumer and differentiate a MIDlet from the competition. Of course I grant you the permission to cite what sentences you like best in your essay.

Best regards
Tito Costa
CEO
Uppli, San Diego

B.4 Svar från Reqwireless

My name is Roger Skubowius, I'm the founder of Reqwireless.

Your question per specializing a MIDlet to a particular platform is an interesting problem we've faced, and I thought it may be useful to share with you our thoughts on this matter.

The basis of J2ME/MIDP is to provide a portable development/runtime platform for limited devices, such as cell phones and PDAs. As this genre of devices has their own unique idiosyncrasies, it becomes tempting to take advantage of these per-device capabilities rather than program for the generalized J2ME/MIDP interface. I would suggest not pursuing this temptation for a number of reasons:

- programmers tend to believe that they can isolate these per-devices enhancements through a portability layer. In this case, the application calls API points in the portability layer, and per-device issues are dealt with within that layer. In theory, this is a good way to approach this problem. Given the MIDlet size constraints and runtime speed available to applications, this is generally not possible for J2ME/MIDP. In this environment, applications tend to have to be "bare-bones", where there is simply no room for a MIDlet to have extra code, such as a portability layer.

- introducing per-device enhancements defeats the theme of J2ME/MIDP. This environment is supposed to address and abstract out many of the per-device oddities. Having an application call per-device functionality is against this theme and results in, not only different look&feel issues across disparate devices, but also now requires users to download specific MIDlets for their specific device(s). Again, this goes against the theme of Java and J2ME/MIDP.

- longer development cycles: Any time a product must deal with per-device issues, the development speed and, usually code quality, suffers.

I do believe that a MIDlet can and should adapt to its runtime environment, but only in a non-device manner. For example, an application could determine how much memory is currently available and attempt to allocate 10% of that memory for an internal cache. Compare this to an absolute value, the distinction is clear and illustrates how an application could adapt to different devices and their capabilities.

I hope I've been of some help, please let me know if there's anything else I can assist you with.
Roger Skubowius, Reqwireless Inc.

B.5 Svar från CoreJ2ME

The idea behind MIDP is to provide a platform/API that will run on a variety of devices. However, to access device specific features would require an API designed/developed specifically for that device. Although such an API may provide valuable functionality, it limits the portability of the application among devices.

In my opinion, it comes down to what device(s) the developer is targeting. For development of applications within a company, it may be reasonable to use a device specific API. That is, if company "A" purchases all devices from one manufacturer, using a device specific API may not be a problem.

On the other hand, to develop applications for distribution to the consumer market, it would be important to support as many devices as possible.

I hope this helps.

John
CoreJ2ME

C MyConnector-klass

Nedan följer koden till en klass som utför nätverkshandlingar.

```
import java.io.*;
import javax.microedition.io.*;

public class MyConnector extends Thread
{
   private MyMIDlet mMIDlet;
   private String mURL;
   private HttpConnection mHttpConnection;
   private boolean mCancel, mTrucking;
   public MyConnector(MyMIDlet midlet, String url)
   {
      mMIDlet = midlet;
      mURL = url;
      mCancel = false;
      mTrucking = false;
   }

   public synchronized void run()
   {
      while (mTrucking)              // Används för att veta om
      {                              // en ny förbindelse skall
                                     // skapas
         try
         {
            wait();                  // Sätter tråden att vänta
         }
         catch (InterruptedException ie)
         {}
         if (mTrucking)
            connect();
```

```
    }
}
public synchronized void go()     // Används istället för start
{                                 // för att starta tråden igen
    notify();
}

public void cancel()              // Används för att avbryta
{                                 // uppkopplingen
    try
    {
        mCancel = true;
        if (mHttpConnection != null)
            mHttpConnection.close();
    }
    catch (IOException ignored)
    {}
}

private void connect()
{
    InputStream in = null;
    try
    {
        mHttpConnection = (HttpConnection)Connector.open(mURL);
                                  // kopplar upp förbindelse
        mHttpConnection.setRequestProperty("Connection","close");
                                  // Tvingar servern att
                                  // stänga socketen när den
                                  // skrivit klart
        in = mHttpConnection.openInputStream();

        int contentLength = (int)mHttpConnection.getLength();
        if (contentLength == -1)
        contentLength = 255;      // Undersöker serverns svar
        byte[] raw = new byte[contentLength];
        int length = in.read(raw);  // Läser in byteström
```

```
            in.close();                 // stänger förbindelse
            mHttpConnection.close();
            String s = new String(raw, 0, length);
                                        // skriver svaret till
            mMIDlet.networkResponse(s); // MIDlet:en
         }
         catch (IOException ioe)
         {
            if (mCancel == false)       // Om något fel har skett
            {                           // undersöks att inte cancel
               try                      // har anropats
               {
                  if (in != null)
                     in.close();
                  if (mHttpConnection != null)
                     mHttpConnection.close();
               }
               catch (IOException ignored)
               {}
            mMIDlet.networkException(ioe);
                                        // Ett undantag returneras
         }                              // om cancel inte valts
         mCancel = false;
      }
   }
}
```

D Metoder från klassen HttpConnection

En nätverksförbindelse kan vara i tre tillstånd:

- Setup, förbindelsen med servern har ännu inte skett. Dessa metoder kan då anropas:

setRequestMethod	-	Sätter ”mode” till antingen POST, GET eller HEAD.
setRequestProperty	-	Sätter ett värde till ett nyckelord, som t ex *accept.*

- Connected, förbindelsen har satts upp, begäran är skickad och svar väntas. Dessa metoder överför “mode” till Connected tillstånd, Connected innebär att man kan skicka begäran till servern. Man behöver inte ha skickat något meddelande för att vara i tillståndet Connected:

openInputStream	-	Skapar och öppnar en inström för bytes.
openOutputStream	-	Skapar och öppnar en utström för bytes.
openDataInputStream	-	Skapar och öppnar en ström för att läsa in primitiva datatyper på ett portabelt sätt.
openDataOutputStream	-	Skapar och öppnar en ström för att skriva primitiva datatyper på ett portabelt sätt.
getLength	-	för HTTP returneras värdet på *content –length* fältet i headern.
getType	-	för HTTP returneras värdet på *content–type* fältet i headern.
getEncoding	-	för HTTP returneras värdet på *content–encoding* fältet i headern.
getHeaderField	-	returnerar värdet på den header vars namn man skickar med som parameter.
getResponseCode	-	Om man t ex får ett HTTP-svar som ser ut så här: HTTP:/ 1.0 200 OK, returnerar metoden statusvärdet 200.
getResponseMessage	-	Om man t ex får ett HTTP-svar som ser ut så här: HTTP:/ 1.0 200 OK, returnerar metoden statusmeddelandet OK.
getHeaderFieldInt	-	Returnerar värdet på det header-fält som man specificerar som parameter, finns det inte skickas ett defaultvärde tillbaks som man specificerat m h a en medskickad parameter.
getHeaderFieldDate	-	Returnerar värdet på det headerfält vars namn man skickat med som parameter parserat som ett datum.

getExpiration	-	Returnerar värdet på headerfältet *expires.*
getDate	-	Returnerar värdet på *date*-fältet i headern.
getLastModified	-	Returnerar värdet på *last-modified*-fältet i headern.
getHeaderField	-	Returnerar värdet på det fält vars namn man skickar med som parameter.
getHeaderFieldKey	-	Returnera nyckeln till det x:te fältet som man skickar med som parameter.

- Closed, förbindelsen är stängd, ett IOException kastas om man försöker anropa klassen *Connection* och dess underklassers metoder.

Vissa metoder kan man använda när förbindelsen är öppen. Dessa är:

close	-	*close* stänger strömmar och förbindelser.
getRequestMethod	-	returnerar det nuvarande "mode":t, som kan vara GET, POST eller HEAD.
getRequestProperty	-	Returnerar värdet på det medskickade property-namnet.
getUrl	-	Returnerar strängvärdet på på Url:en för den förbindelsen.
getHost	-	Returnerar värd-information m h a Url:en.
getProtocol	-	Returnerar namnet på protokollet, t ex http eller https.
getFile	-	Returnerar fil-delen på Url:en.
getPort	-	Returnerar portnumret på Url:en.
getQuery	-	Returnerar frågedelen av Url:en.
getRef	-	Returnerar refdelen av Url:en.

När man skapar en förbindelse m h a Connectorklassens open-metod kan man göra det i tre varianter. Dessa är:

- open(String name)
  name är namnet på målet, d v s serverns address.
- *open(String name, int mode)*
  name är namnet på målet, d v s serverns adress.
  mode talar om vilken access-mode man vill ha. Som exempel finns READ och WRITE
- *open(String name, int mode, boolean timeouts)*
  name är namnet på målet, d v s serverns adress.
  mode talar om vilken access-mode man vill ha.
  timeouts talar om att man vill ha timeout-undantag vilket betyr att man inte vill vänta för evigt.

E Http-connection

```
String url = "http://myServer.com/servlet/MyServlet ";
...
void invokeServlet(String url) throws IOException
 {
   HttpConnection c = null;
   InputStream is = null;
   OutputStream os = null;
   StringBuffer b = new StringBuffer();
   TextBox t = null;
   try
   {
      // skapa förbindelse
      c = (HttpConnection)Connector.open(url);
      // sätter request-metoden för http- förbindelsen till POST
      c.setRequestMethod(HttpConnection.POST);
      // öppnar en utström
      os = c.openOutputStream();
      // skriver meddelandet i en byteström
      os.write(("…”).getBytes());
      // tvingar en eventuellt buffrad bytestöm att skrivas.
      os.flush();



      //öppnar en inström
      is = c.openDataInputStream();
      int ch;
      // läser inströmmen
      while ((ch = is.read()) != -1)
      {
                 b.append((char) ch);
         System.out.print((char)ch);
      }
      // skriver ut byteströmmen som en sträng i en TextBox
      t = new TextBox("Confirmation", b.toString(), 1024, 0);
```

```
            t.setCommandListener(this);
      }
      finally                    // finally anropas alltid efter try-
      {                          // satsen
         // Stäng strömmarna och http-förbindelsen
         …
      }
      display.setCurrent(t);
   }
…
```

F Servlet

```
public class EmailServlet extends HttpServlet
  {
     // request innehåller httpbegäran från klienten till servern
     // response innehåller serverns svar
     public void doPost(HttpServletRequest request,
     HttpServletResponse response)
     throws IOException, ServletException
     {
        // Formatterar svarstexten
        response.setContentType("text/plain");
        // Hämta skrivare
         PrintWriter out = response.getWriter();
        //Kontakta tredjeparts-applikation, t ex via RMI
        …
        // Skickar meddelande till klienten
        out.println("meddelande till midlet");
     }
}
```

G Ordlista

MIDP	Mobile Information Device Profile – Den profil av Java som används för att skapa applikationer till mobiltelefoner
CLDC	Connected Limited Device Configuration – Den konfiguration som ligger under MIDP. Konfigurationen specificerar krav på den exekverinsmiljö som den exekverar i.
MIDlet	En Javaapplikation till en mobiltelefon utvecklad m h a MIDP.
J2SE	Java 2 Standard Edition – Den Javateknologi som används för att skapa applikationer till persondatorer.
J2EE	Java 2 Enterprise Edition - Den Javateknologi som används för att skapa serverapplikationer applikationer till servrar.
AMS	Application Managment System – Del av mobiltelefonens operativsystem där Javas virtuella maskin exekverar.
Virtuell maskin	En av Sun Microsystem specifierad applikationsmotor som används för att exekvera Javaapplikationer i.
Obfuscering	Ett sätt att minska storleken på källkoden genom att döpa om namn på funktioner och variabler.
API	Application Programming Interface – Klasser och funktioner som finns definierade i ett programmeringsspråk.
RMS	Record Managment System – Ett antal klasser som används för att kunna spara information i mobiltelefonen när applikationen avslutats.

Siemens SL45i	Mobiltelefon som har kapacitet att ladda ner och exekvera MIDlets.
Motorola Accompli008	Mobiltelefon som har kapacitet att ladda ner och exekvera MIDlets.
PNG	Portable Network Graphics – Ett bildformat som används i mobiltelefoner.
RMI	Remote Method Invocation – En Javateknologi för att kunna anropa funktioner i en serverapplikation från en klientapplikation.
J2ME	Java 2 Micro Edition - Den Javateknologi som används för att skapa applikationer till apparater med begränsade resurser.
GUI	Graphical User Interface – Ett grafiskt användargränssnitt.
CDC	Connected Device Configuration – En konfiguration som specificerar krav på den exekverinsmiljö som konfigurationen exekverar i.
JVM	Den virtuella maskin som används i persondatorer.
KVM	Den virtuella maskin som används i mobiltelefoner.
Skräpsammlare	En skräpsammlare är en funktion som söker igenom minnet efter allokerade objekt som inte används längre och avallokerar dessa.
Java	Programmeringsspråk utvecklat av företaget Sun Microsystems.
JAR	Java Archive – Ett filformat som innehåller en eller flera klassfiler.