Datavetenskap

Carl-Magnus Andersson & Claes Röjder

Ett mobiltelefonspel blir till

Examensarbete, C-nivå

2004:20

Ett mobiltelefonspel blir till

Carl-Magnus Andersson & Claes Röjder

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

Carl-Magnus Andersson & Claes Röjder

Godkänd, 2004-06-03

Handledare: Per Strömgren

Examinator: Stefan Alfredsson

Sammanfattning

Under våren 2004 fick vi i uppdrag av Carbello AB att färdigutveckla ett spelkoncept för mobiltelefoner. Denna rapport handlar om hur detta uppdrag genomfördes. Under projektets gång har vi lärt oss åtskilliga saker och vi delar med oss av vår lärdom om sånt som är viktigt att tänka på då man utvecklar applikationer för mobiltelefoner. Några av dessa saker är vikten av att hålla ner storleken på applikationen med hjälp av diverse knep, dela upp utvecklandet i smådelar och självklart att ha en ständig kommunikation sinsemellan samt med uppdragsgivaren. I rapporten beskrivs våra med- och motgångar och vad vi känner att vi skulle göra annorlunda om vi skulle få chansen att göra ett liknande projekt. Resultatet av detta examensarbete är att vi fått förståelse för hur applikationer för mobiltelefoner tillverkas och vi lyckades att skapa ett spel som vi är nöjda med och som fungerar på det sätt som planerats.

A mobile phone game is developed

Abstract

We were assigned by Carbello AB during the spring of 2004 to develop a game for a mobile phone. This report is about how that assignment was done. We have learned several things during the project and you'll find recommendations of what to do and not to do when you produce applications for a mobile phone. Some of the things you should think of are to create small applications by using some tricks, for example divide the project into smaller parts and of course always keep a good communication with each other and the supervisor. This report describes our ups and downs during the project and tells what we would do different if we ever did a similar project. The result of this project is that we have got an increased understanding about how applications for mobile phones are made and we were able to produce a game that we are very pleased with and that works in the way we intended.

Innehållsförteckning

Figurförteckning

Tabellförteckning

1 Inledning

Inledningen behandlar bakgrunden till examensarbetet där även vår uppdragsgivare, Carbello, beskrivs samt den arbetsuppgift som vi erhöll från dem. Inledningen beskriver även syfte och mål med examensarbetet. För de som inte är insatta i applikationsutveckling för mobiltelefoner finns det fördjupande avsnitt om J2ME och MIDP.

1.1 Bakgrund

Mobiltelefonerna har utvecklats mycket snabbt under de senaste åren och denna utveckling kommer sannolikt att fortsätta även i framtiden. Detta har lett till att marknaden för applikationer, bland vilka även spel ingår, för mobiltelefoner har ökat. Spel baserade på programmeringsspråket Java kan med lätthet läggas in i nyare mobiltelefoner av användarna själva.

1.1.1 Carbello AB

Carbello är ett företag baserat i Karlstad, som inriktar sig på speltillverkning för mobiltelefoner och har legat bakom en rad populära spel. De utvecklar spel på projektbasis där utvecklingen sker i deras egna lokaler.

1.1.2 Inledande kontakter

Vi kontaktade Carbello för att höra om de kunde erbjuda oss ett projekt att utveckla under vårt examensarbete. Vi fick en positiv respons från dem då de hade en spelidé liggande som de själva inte hade tid att utveckla. Resultatet av detta blev att vi skulle producera ett tetris-liknande spel åt dem.

1.2 Syfte

Vårt syfte med detta examensarbete var att fördjupa oss i utvecklandet av spelapplikationer för mobiltelefoner genom att använda oss av J2ME (Java™ 2 Platform, Micro Edition).

Ingen av oss hade några tidigare erfarenheter av att skriva program för mobiltelefoner men vi antog att det skulle vara både spännande och intressant. Vi tror också att det är nyttigt att ha någon slags erfarenhet av att jobba med mobiltelefoner när vi i framtiden söker jobb.

Under utvecklingen av spelet fick vi en väldigt bra inblick i hur speltillverkning går till och det svåra i att sätta sig in i ett delvis nytt programmeringsspråk.

1.3 Mål

1.3.1 Delmål i projektet

Vi satte tidigt upp fyra delmål:

Mål 1 - Spelidén och reglerna har sammanfattats i ett dokument.

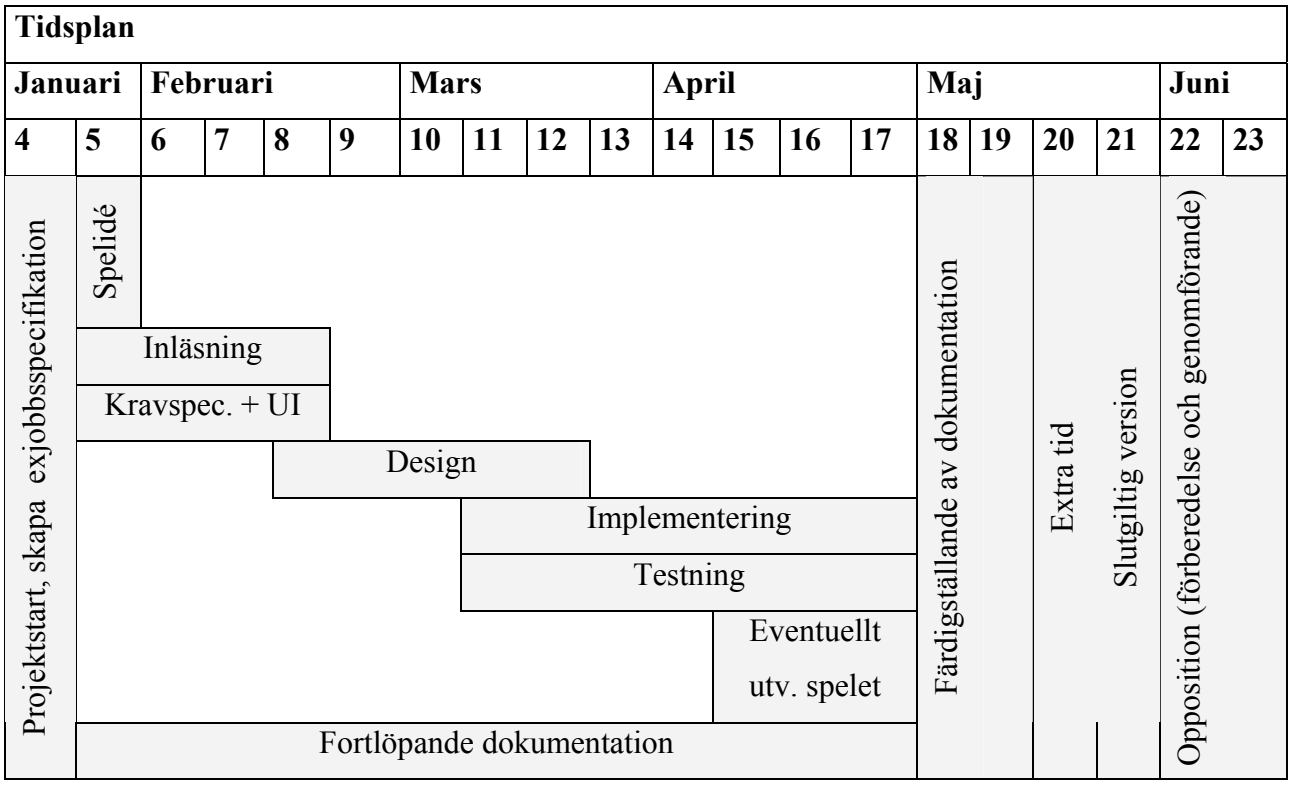
Mål 2 - Vi har lärt oss att använda programvaran och kan skapa enklare applikationer.

Mål 3 - Designen är färdig.

Mål 4 - Spelet är färdigställt och fungerar korrekt på en Nokia-telefon.

1.3.2 Tidsplan

För att lättare nå våra mål satte vi tidigt upp en tidsplan.



Tabell 1 Tidsplan

Deadline Mål 1 slutet på vecka 5

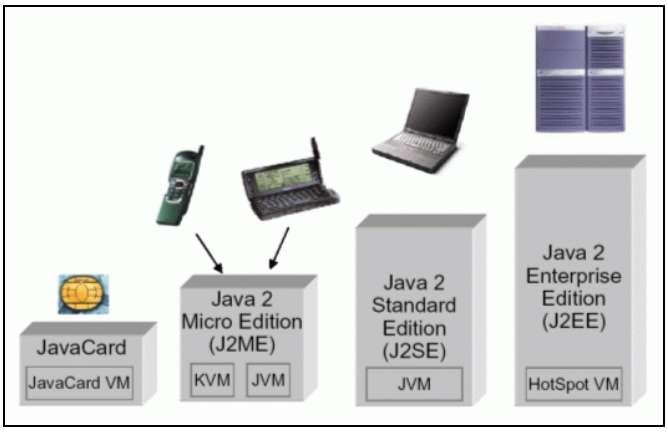
Deadline Mål 2 slutet på vecka 8

Deadline Mål 3 slutet på vecka 12

Deadline Mål 4 slutet på vecka 17

1.4 J2ME

J2ME står för Java™ 2 Platform, Micro Edition [1] och är en nedbantad version av Suns stora plattform, Java 2. Java 2 består, förutom av J2ME, även av J2SE (Java 2 Platform, Standard Edition) som i första hand är skapad för att utveckla applikationer för arbetsstationer och J2EE (Java 2 Platform, Enterprise Edition) som är designat för att utveckla omfattande affärssystem.



Figur 1 Javaplattformen

Förr i tiden var det viktigt att utveckla program som tog upp så lite av datorns minne som möjligt men i takt med att datorernas prestanda förbättrades med mer minne, snabbare processorer och större hårddiskar så behövde utvecklarna inte längre tänka lika ofta på hur de program de utvecklade skulle klara sig i förhållande till prestandan i datorn. Det har dock på senare tid återigen blivit viktigt att utveckla applikationer som använder ett minimum av prestanda. Detta beror på att dagens teknik gjort det möjligt att skapa relativt avancerade applikationer för handburna enheter såsom mobiltelefoner och handdatorer. Tack vare J2ME kan Java-applikationer utvecklas för mobiltelefoner och handdatorer. Genom att plocka bort de paket som tog upp mycket minne och endast behålla de mest primära klasserna (ärvda från paketen java.lang, java.io, och java.util) lyckades utvecklarna minska storleken på J2ME. Konfigurationen som används kallas CLDC (Connected Limited Device Configuration) och består av en virtuell maskin samt ett bibliotek med de klasser som blev kvar då de mest krävande klasserna plockades bort. När CLDC kopplas samman med MIDP (Mobile Information Device Profile) resulterar detta i en plattform där utvecklare kan skapa

applikationer som kan köras på apparater med lite minne, liten processorkraft och minimalt med grafik.

1.5 MIDP

MIDP tillhandahåller en standardmiljö för att köra J2ME applikationer på mobiltelefoner och enkla typer av handdatorer. Med hjälp av MIDPs högnivå-API (paket som innehåller en samling standardklasser samt specifikationer för klassernas metoder och attribut) kan standardkomponenter skapas och de applikationer som använder sig av dessa kan köras på många olika apparater från många olika tillverkare. En applikation som enbart använder sig av dessa API fungerar på samtliga apparater som stöder J2ME. Fördelen med högnivå-API är alltså portabilitet men det leder också till minskad flexibilitet t.ex. när det gäller applikationens utseende. För att få större kontroll över applikationer kan lågnivå-API användas och ännu större kontroll fås genom att använda API som tillhandahålls av tillverkarna av mobiltelefonerna. Nackdelen med dessa är dock att portabiliteten försvinner eftersom applikationerna inte fungerar på alla apparater även om de stöder J2ME.

1.5.1 MIDP 1.0

De första mobiltelefonerna med MIDP 1.0 inbyggt började dyka upp runt 2001. MIDP 1.0 innehåller grundläggande funktionalitet för användargränssnitt (menyer, textfält etc.), en RecordStore för att kunna spara data samt grundläggande nätverkskompabilitet [4] men trots detta så saknas några funktioner som kan tyckas nödvändiga. MIDP 1.0 saknar bland annat stöd för att använda hela ytan av en mobiltelefons display, använda sig av bilder med genomskinlighet samt möjligheten att känna av knapptryckningar som involverar fler än en knapp åt gången. Det går inte heller att spela upp vilket ljud som helst med MIDP 1.0 men genom att utnyttja en mobiltelefons inbyggda ljud som används då en Alert visas kan ett fåtal ljud trots allt spelas upp.

1.5.2 MIDP 2.0

MIDP 2.0 är bakåtkompatibel med MIDP 1.0 och när den kom var i princip varje klass i MIDPs användargränssnitt uppdaterad. MIDP 2.0 erbjuder större grafiska möjligheter som till exempel att skapa egna grafiska objekt och att rita trianglar samt en bättre layoutkontroll. MIDP 2.0 har även ett nytt paket som heter Game som är inkluderat i lcdui-paketet (lcdui-paketet innehåller klasser och metoder för det grafiska användargränssnittet i Java). Gamepaketet gör att utvecklare får bättre kontroll över spelknapparnas tillstånd och det finns

även en automatisk skärmuppdatering. ”Layer” är också en ny klass vilken representerar visuella objekt och med hjälp av denna klass kan ett elements position, storlek samt information om huruvida det syns eller ej erhållas. ”Sprite” är en klass som ärver av ”Layer” och som består av en samling grafiska block som kan roteras och flippas och har metoder som känner av när de kolliderar med varandra (även kallat collision detection). Multipla lager tillåts i MIDP 2.0 och detta sköts av klassen ”LayerManager”. MIDP 2.0 erbjuder också möjligheten till ljuduppspelningar, t.ex. kan ljud spelas upp från en webbserver. Ytterligare en förbättring jämfört med MIDP 1.0 är att MIDP 2.0 kan använda sig av flyttal.

1.5.3 Säkerhet

I MIDP 1.0 sköts säkerheten med hjälp av en ”sandlådemodell” där varje applikation på mobiltelefonen endast får köras i sin egna säkra miljö (sin egen sandlåda). Detta gör visserligen att applikationer inte riskerar att förstöra något i en annan applikation men det leder också till att vissa nätverksresurser och nätverkskommunikationer inte finns tillgängliga. MIDP 2.0 har ett sätt att signera och certifiera applikationer vilket gör att en apparat kan ge tillåtelse för en applikation, som den vet är säker, att använda sig av vissa känsliga nätverksresurser.

1.5.4 Valet av MIDP till vårt spel

Endast en del av allt det nya i MIDP 2.0 har beskrivits ovan och dess potential är oerhört stor. I dagens läge är det dock bara några få nyare mobiltelefoner som stöder MIDP 2.0 vilket betyder att det är bättre att utveckla spel med hjälp av MIDP 1.0 eftersom dessa går att spela på alla mobiltelefoner som stöder J2ME. Vårt spel utvecklas till Nokias Serie 40 som använder sig av MIDP 1.0 och därför har vi inte kunnat ta del av MIDP 2.0 som verkar vara ett bättre alternativ då spel ska utvecklas.

2 Arbetet

Vi fick i uppdrag av Carbello att skapa ett spel för en mobiltelefon. Carbello hade en grundidé för spelet men den var varken dokumenterad eller speciellt genomarbetad. För att få struktur på arbetet skapade vi en exjobbsspecifikation där de viktigaste punkterna noterades. Efter att exjobbsspecifikationen färdigställts påbörjades dokumentationen av själva spelidén. Under början av arbetet satte vi oss, var och en för sig, in i J2ME och det utvecklingsverktyg, JBuilder X, som vi valt. Detta var dock inte särskilt lätt och det gick många timmar till denna

inläsning. I samband med att dokumenteringen av spelidén (se bilaga B) slutfördes skapade vi ett flödesschema för de olika menyerna i spelet, samt ett klassdiagram (se bilaga C) tillsammans med de metoder som vi ansåg oss behöva för att kunna göra spelet.

2.1 Arbetsuppgifter

Arbetsuppgifterna för att skapa spelet bestod av att:

1. ta fram en exjobbsspecifikation
2. skapa en kravspecifikation (innehållande den dokumenterade spelidén)
3. rita ett flödesdiagram över menyerna i spelet
4. ta fram ett klassdiagram med tillhörande metoder
5. implementera spelet

I slutet av implementeringen utfördes tester för att säkerställa att spelet motsvarade Carbellos krav.

2.2 Förutsättningar

Då spel till mobiltelefoner skapas finns det vissa saker att tänka på:

- Spel för mobiltelefoner ska ha korta speltider. Detta beror på att spelaren inte vill att batteriet ska ta slut och förr eller senare vill denne ta emot eller ringa samtal. Användaren vill ha något som denne kan fördriva tiden med en kort stund.
- Spel ska vara så små som möjligt på grund av att mobiltelefoner har små minnen och små processorer. Fördelen med att spelen är små är att utvecklaren ägnar mer tid åt att få fram en bra spelidé istället för att lägga ner mycket tid på grafiska effekter. Detta gör att dessa spel känns mer genomtänkta och spelas en längre tid innan de glöms bort.
- Spel ska skapas på ett sådant sätt att det är enkelt att anpassa för mobiltelefoner av olika märken och modeller.
- Spelet ska vara designat på ett sådant sätt att en eventuell översättning av språket i spelet går snabbt och enkelt att genomföra. Engelska fungerar inte överallt.
- Spelet ska skapas med MIDP. Vilken version av MIDP som används är direkt knutet till den mobiltelefon som spelet utvecklas för då alla mobiltelefoner i nuläget inte kan utnyttja den senaste versionen (MIDP2).

2.3 Krav och spelidé

Vi pratade med vår handledare på Carbello för att få reda på vad för slags spel som de hade tänkt sig att vi skulle skapa och vilka krav som fanns på detta. Carbello hade gjort en enkel prototyp i Shockwave som vi studerade för att få idéer och inspiration när vi dokumenterade spelidén. Prototypen var till stor hjälp då vi redan i inledningen av arbetet kunde se hur det var tänkt att spelobjekten skulle interagera med varandra.

Det första vi gjorde i examensarbetet var att skapa en exjobbsspecifikation där vi i stora drag gick igenom vad arbetet gick ut på samt satte upp en tidtabell med olika delmål. Tidtabellen baserades på vår erfarenhet av tidigare projekt samt de framtagna förslagen som fanns på universitetets hemsida för examensarbeten. Bland de saker som ingick i exjobbsspecifikationen var en rimlighetsbedömning där vi fick bedöma huruvida vi trodde oss kunna lyckas med det åtagna arbetet eller ej. Anledningen till att vi började med exjobbsspecifikationen var att examensarbetet inte fick något definitivt klartecken förrän den hade skickats in och godkänts av universitetet.

När vårt förslag till examensarbete blivit godkänt började vi med att skriva ner den grundläggande spelidén för att sedan kontrollera med vår handledare att den spelidé vi tagit fram överensstämde med deras vision av spelet. Förutom handledarens kommentarer hämtade vi inspiration från spelet Candy Crisis vilket var det spel som Carbello tittade på när de kom på idén till det spel vi fick i uppdrag att utveckla. Den övriga spelidén jobbade vi fram genom att bolla olika idéer och aspekter av spelet mellan varandra. Vi beslutade oss för att använda Carbellos egna namnförslag, Blobs, som arbetsnamn på spelet och spelkomponenterna fick det passande namnet blobbar. Den viktigaste delen av spelidén är spelreglerna, som vi ägnade mycket tid åt att finslipa, och en av de svåraste uppgifterna var att komma fram till en bra poängberäkning. Detta löste vi genom att skapa en preliminär poängberäkning som vi planerade att justera senare om vi upptäckte att den inte fungerade på ett bra sätt. Då vi visade upp den spelidé vi hade kommit fram till föreslog vår handledare att vi skulle lägga till ett nytt spelobjekt, vilket även förekom i Candy Crisis, för att spelet inte skulle bli för enkelt att spela. Detta resulterade i att glaskulor lades till i spelet, vilka slumpmässigt skulle falla ner på spelplanen, för att försvåra borttagandet av blobbar. Vi beslutade även att det skulle finnas en high score-lista där de tre bästa resultaten skulle sparas.

Alla tekniska krav på spelet sammanfattades i ett enskilt dokument, främst för att veta vad vi var tvungna att tänka på under implementeringen. De viktigaste kraven var att spelets JAR-fil inte fick överstiga 64 kB samt att spelet som ett minimum skulle fungera på Nokia Serie 40-telefoner. För de övriga tekniska aspekterna hänvisar vi till bilaga B. De tekniska kraven bildade kravspecifikationen tillsammans med spelidén.

Efter att kravspecifikationen slutförts skapades ett flödesdiagram som åskådliggjorde hur de olika spelmenyerna såg ut och hur de interagerade med varandra.

När alla dokument som beskrivits ovan färdigställts var det dags att börja designa spelet.

2.4 Design

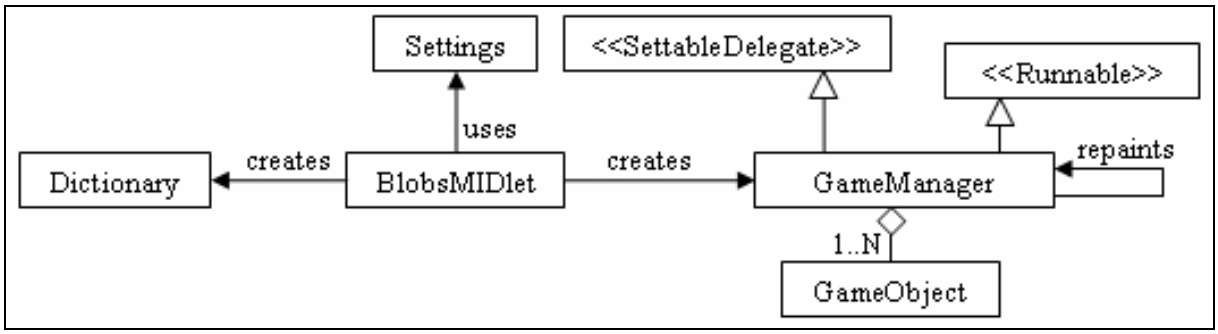
Designen bestod av att skapa ett diagram över de klasser som behövdes för att skapa spelet samt att dokumentera de metoder som behövdes.

Vår första idé när det gällde spelet var att spelplanen skulle delas in i en matris i vilken spelobjekten skulle placeras. Idén byggde på att de spelobjekt som introducerades på spelplanen skulle falla ner samtidigt som kontroller gjordes mot matrisen för att se om de kolliderade med spelobjekt som redan befann sig på spelplanen. Om så var fallet skulle de fallande spelobjekten läggas in i matrisen. Därefter skulle kontroller och beräkningar göras för att avgöra om spelobjekt skulle tas bort från spelplanen eller ej. När detta var gjort skulle nya spelobjekt introduceras på banan. Detta skulle sedan fortlöpa tills matrisen fyllts ända till toppen varvid spelet var över. Idén om hur spelet skulle byggas upp visade sig hålla och följde med genom hela projektet.

Först skapades ett preliminärt klassdiagram med 17 klasser samt en väldigt enkel beskrivning av vad de olika klasserna innehöll. Detta diagram skickades till handledaren på Carbello för att höra vad han hade för synpunkter på det. Handledaren svarade att då J2ME-spel skapas ska, av prestandaskäl, alltid ett minimalt antal klasser användas och att han brukade använda sig av fem eller sex då han skapade spel. Anledningen till att så få klasser som möjligt ska användas är dels att det skapas en del data i varje klass header i de kompilerade klassfilerna och dels för att många små filer blir större än en stor i JAR-filen på

grund av att filerna inte packas ihop med varandra utan var och en för sig. Få klasser resulterar alltså i bättre prestanda i och med att storleken på spelet blir mindre.

Efter att ha fått synpunkterna på det preliminära diagrammet skapade vi ett nytt diagram bestående av fem klasser, alltså betydligt färre än de 17 som vi från början tänkt använda, se figur 2. Vi beskrev också vilka metoder som planerades att användas för att bygga upp spelet.



Figur 2 Klassdiagram

Under implementationen uppdaterade vi kontinuerligt designdokumentet innehållande diagrammet och metoderna. I slutet av implementeringen lades en extra klass till vars främsta uppgift var att ta hand om utskrifterna av huvudmenyn och de olika undermenyerna. Anledningen till att en ny klass kunde läggas till var att spelets totala storlek, inklusive alla bilder, var mycket mindre än de 64 kB som spelet maximalt fick uppgå till.

För att undvika redundans, som skulle kunna uppstå på grund av att vi skrev kommentarer både i koden och i designdokumentet, valde vi i slutskedet av implementationsfasen att ta bort metodbeskrivningarna ur designdokumentet för att med hjälp av ett speciellt dokumentations-format som heter Javadoc skapa beskrivningar av klasser, metoder och medlemsattribut. Ett exempel på hur vi har dokumenterat kan ses i bilaga D. Dessa beskrivningar kan skrivas ut för att komplettera klassdiagrammet som designdokument.

2.5 Implementering

Implementeringsfasen var den fas som gav mest problem men även den som var roligast att utföra. Det var här vi fick se när det vi hade planerat i de andra faserna började växa till liv.

2.5.1 Implementeringen börjar

Vi började implementeringen med att skapa alla de klasser som hade tagits fram under den inledande designdelen av projektet varefter vi byggde skelett av de olika metoder som tagits fram. Alla metoder som slutligen ingick i spelet fanns inte att tillgå från början av implementeringen eftersom designdelen och implementering överlappade varandra i tidsplaneringen.

Efter att grunden för spelet byggts skapades menyn som skulle visas då MIDleten, applikationen på mobiltelefonen, startades. I början nöjde vi oss med att endast ett kommando på menyn fungerade, nämligen ”New Game”. Det som hände då vi valde att spela ett nytt spel var att grafiken för spelplanen samt den omkringliggande grafiken ritades upp. Nästa steg var att rita ut en ensam blob som vi, efter inläsning och testande, lyckades få att falla ned på skärmen. Efter ytterligare tester lyckades vi få fram två blobbar varav den ena av dem roterade runt den andra. Då detta var avklarat fortsatte vi att skapa den grundläggande funktionaliteten vilket vi gjorde genom att få spelobjekten att falla på rätt sätt. Under tiden vi experimenterade med detta implementerade vi den Nokia-specifika metoden fullCanvas vilken gör att hela skärmen utnyttjas då spelet ritas upp. Utan fullCanvas ritas en kant med titel ut överst på displayen och en kant med de alternativ som kan utföras längst ner.

För att hålla reda på spelobjekten som kommer in på spelplanen skapades en matris. Matrisen användes också för att hålla reda på var de aktiva (dvs. de spelobjekt som faller) spelobjekten var på spelplanen samt om de krockade med några av de inaktiva objekten. Dessutom användes matrisen till att hålla reda på kanterna av spelplanen.

För att slumpa fram om ett spelobjekt skulle vara en blob eller en glaskula användes den inbyggda metoden random() som hittas i math-biblioteket. Vi använde denna tillsammans med moduluräkning för att slumpa fram heltal. Denna slumpningsmetod användes även till andra slumpningar som till exempel vilken färg en blob skulle ha.

Då spelaren trycker på nedåtknappen ska spelobjektet (förutsatt att de båda blobbarna är aktiva) falla snabbare. Först tänkte vi flytta de aktiva blobbarna flera pixlar i y-led vid varje uppdatering då de skulle falla snabbare men när vi använde oss av denna metod fick det effekten att vi ofrivilligt hamnade utanför spelplanen. Istället bestämde vi oss för att ändra uppdateringsintervallet (den tid som avgör hur långt det är mellan varje ny uppritning av

skärmen) och därigenom skapa illusionen att blobbarna faller snabbare. Detta gjorde att vi slapp ovan nämnda bieffekt.

2.5.2 Färgsättning och sammanfogning av blobbarna

En blob kan anta 16 olika tillstånd (t.ex. sammankopplad uppåt och nedåt eller sammankopplad åt vänster) och varje tillstånd representeras av en bild. Vi erhöll ett antal bilder från Carbello som symboliserade de olika tillstånden för en blob. Dessa originalbilder som finns med i JAR-filen är gula och en av våra stora utmaningar var att när spelet startar ska det ladda in dessa bilder med olika färger. Till vår hjälp fick vi lite kod från Carbello för att kunna ändra bildernas färgpaletter men trots att vi fått koden lyckades vi till en början inte att skapa bilderna i andra färger. Efter att ha slitit med detta ett antal timmar hade vi ett möte med handledaren på Carbello där han förklarade koden för oss. Det visade sig att vi hade gjort ett så simpelt fel som att glömma att indexera bilderna vilket ledde till att deras färgpalett inte kunde ändras (detta beror på att bilder som inte är indexerade inte har någon färgpalett).

När färgproblemet för de olika blobbarna lösts började vi fundera på hur vi skulle göra för att få dem att flyta ihop med varandra då blobbar av samma färg placeras intill varandra. Detta visade sig dock inte vara något större problem utan löstes relativt snabbt med hjälp av en metod som gick igenom matrisen då en blob landade. Då minst fyra blobbar av samma färg befann sig bredvid varandra togs dessa bort med hjälp av en rekursiv funktion som höll reda på hur många blobbar som skulle tas bort samt deras position i matrisen. Ett tillägg till detta blev senare att även angränsande glaskulor skulle raderas.

2.5.3 Utvecklandet fortsätter

Allt eftersom implementeringen fortskred ägnades då och då några timmar till att snygga till och förbättra koden. Anledningen till detta var att vi i slutändan ville ha så lite kod som möjligt och att den skulle vara överskådlig och enkel att läsa.

Poängen i spelet illustreras av egenhändigt skapade bilder som ritas upp beroende på den under spelet givna poängställningen. Varje siffra (0 – 9) är en bild som sätts in på rätt position, ental, tiotal osv.

Avancerar en spelare till en ny nivå i spelet ökas farten med vilken blobbarna faller för att på så sätt öka svårighetsgraden från nivå till nivå. Detta gjordes på samma sätt som då de aktiva blobbarnas fart ökade när en spelare tryckte på nedåt-knappen och var därför enkelt att implementera.

När implementeringen fortsatte uppstod diskussion om huruvida de aktiva blobbarna skulle falla snabbt så långt det går när en användare tryckte nedåt eller om de bara skulle falla snabbt under den tid som användaren höll in knappen för att sedan återgå till nivåns fallhastighet. Efter en ihärdig diskussion sinsemellan samt med handledaren på Carbello beslutades att det bästa förhållningssättet var att låta blobbarna falla ned snabbt så långt det går utan att användaren kan påverka dem. Anledningen till detta var att man som användare inte tjänar något på att blobbarna faller snabbt en liten bit för att sedan kunna styras igen. I liknande spel som tetris tjänar spelaren på detta eftersom det där kan uppstå hål i kolumner där han eller hon kan placera spelobjekt men detta kan ej uppstå i vårt spel.

Vi mätte hur lång tid olika delar av programmet tog på sig för att beräkna olika händelser. En av mätningarna utfördes då två aktiva blobbar blev inaktiva. Detta visade sig vara en ganska krävande operation i jämförelse med beräkningarna som utfördes då blobbarna föll. Den beräkning som tog längst tid var dock då båda blobbarna stannat och kontrollen av vilka blobbar som skulle tas bort tillsammans med poänggenereringen utfördes. Detta tog cirka 10 ms vilket betydde att tiden mellan varje uppritning av displayen måste vara längre än 10 ms.

Ytterligare en förfining av spelet var att vi la till ögon på blobbarna. Om flera blobbar sitter ihop erhåller endast en av dem ögon. När en blob landar kontrolleras om den sitter ihop med andra blobbar och samtidigt kontrolleras hur många ögon som finns på blobbarna som sitter ihop och är det fler än ett par ögon tas de överflödiga ögonen bort.

Bilden av spelplanen delades upp i flera mindre delar för att minska tiden det tog att rita upp den varje gång spelskärmen uppdaterades. Detta gjorde att endast den del av spelplanen som för tillfället behövde uppdateras ritades om.

2.5.4 Skiftning

Vid denna tidpunkt ansåg vi att de mest grundläggande delarna av spelet var färdiga men det visade sig att så inte var fallet. Efter ett möte med handledaren på Carbello fick vi klart för oss

att det var bättre att simulera förflyttningar som var mindre än hela pixlar, istället för att ändra tiden mellan uppdateringarna, då hastigheten med vilken blobbarna föll skulle förändras. Anledningen till att man måste simulera detta är att MIDP 1.0 inte stöder flyttal vilket betyder att t.ex. en blob inte kan flyttas 1,2 pixlar nedåt på skärmen. För att lösa detta måste blobben ha ett tal som representerar var på skärmen den ska skrivas ut. Talet måste även vara tillräckligt stort för att en förflyttning på t.ex. 0,1 pixlar ska kunna simuleras genom att talet ökas eller minskas med ett heltal. Simuleringen går att lösa på två olika sätt, antingen med multiplikation/division eller med skiftning. Vår handledare sa att vi skulle använda oss av skiftning eftersom detta är cirka 30 till 50 procent snabbare [5] än multiplikation. Detta gör att spelet i sig flyter bättre och att uppdateringshastigheten inte behöver förändras. Det svåra för oss var att detta gjorde att alla våra kontroller av hur spelobjekten skulle bete sig på spelplanen inte längre fungerade. Samtliga metoder som gjorde dessa kontroller var tvungna att göras om vilket tog tid. Det allra första problemet vi stötte på med skiftningen var att kontrollen om de aktiva blobbarna hamnade bredvid blobbar av samma färg inte längre fungerade. Detta problem, och flera andra, löstes dock efter ett antal timmars kodande medan ett annat problem visade sig vara mycket svårare att avhjälpa. Då de aktiva blobbarna roterade bredvid ett par andra blobbar i ett visst ögonblick eller om de gick in från sidan i ett par blobbar resulterade detta i att de aktiva blobbarna förflyttades in i de andra blobbarna. Efter ett par dagars arbete lyckades vi dock att lösa felet. Det visade sig att vi hade skrivit fel i en if-sats och när felet lokaliserats åtgärdades detta mycket snabbt.

2.5.5 Slutgiltiga finjusteringar

Efter ett test av spelet på en mobiltelefon ändrades färgen på den lila blobben till orange eftersom det var svårt att se skillnaden mellan den blå och den lila blobben på mobiltelefonens dåligt upplysta display.

Använder en applikation mycket minne (t.ex. då stora bilder har lästs in) utan att lämna tillbaka det då det inte längre behövs kan felet ”unable to run application” uppstå. Detta beror på att minnesläckage inträffar och vårt spel drabbades av detta då huvudmenyn och dess undermenyer implementerades. Vi hade valt att använda bilder istället för Nokias standardmenyer och efter att en meny ritats upp glömde vi att ta bort de bilder som inte användes ur minnet. Detta ledde till ett allt för stort minnesutnyttjande med systemkrasch som följd. Detta löstes genom att instanserna av bilderna nollställdes varefter systemets skräpsamlare (garbage collector) anropades.

Efter ännu ett möte med handledaren på Carbello fick vi tips om hur storleken på spelet drastiskt kunde minskas. Spelets storlek reducerades med drygt 10 kB och det är alltid bra att ett spel för en mobiltelefon storleksmässigt är så litet som möjligt.

Det ena tipset var att istället för att använda 16 olika bilder på hur en blob ser ut vid olika sammansättningar, lägga samtliga bilder i en enda bild. Detta gjorde att storleken minskade, eftersom varje bild har en header vilken bland annat innehåller en palett med de färger som bilden använder sig av, och att inläsningen av blobbarna gick mycket snabbare än vad den gjort innan. Bilderna av blobbarnas olika tillstånd lades i en lång rad och då en av blobbarna skulle ritas upp användes den i Java inbyggda metoden setClip som endast ritar upp en viss del av bilden.

Det andra tipset var att med hjälp av JBuilder ersätta alla metod- och variabelnamn med minimala namn vid kompileringen. Detta leder till att den kompilerade filen som läggs in blir mycket mindre än vad den var innan.

Med anledning av tipset om setClip placerade vi alla bilder som hörde samman i en enda bild. I alla menyer använde vi oss av den i Java inbyggda metoden drawString och ritade ut texten flera gånger för att på detta sätt skapa en kantlinje runt texterna. Detta flöt relativt bra på emulatorn men i mobiltelefonen gick det desto segare. Vi lärde oss senare att just drawString tar väldigt mycket processorkraft vilket ledde till att vi skrev om koden så att kantlinjerna endast ritades ut en gång istället för vid varje uppdatering av skärmen. Vid inmatningen av namn till topplistan, se figur 6, ändrade vi så att endast de två bokstäverna som förändrades ritades upp på nytt vilket gjorde att den menyn fungerade mycket bättre på mobiltelefonen.

Under spelets gång, då spelplanen började bli full av blobbar, blev spelet lite segt. Detta berodde på att alla blobbar på skärmen ritades upp vid varje uppdatering av skärmen. Vi fick gå in i koden och ändra på detta så att endast den del av skärmen som skulle förändas vid uppdateringen ritades om, vilket gjorde att spelet genast flöt mycket bättre.

Efter detta var spelet i princip klart, det enda som hände var att lite överflödiga bilder som inte längre användes togs bort från spelet samt att vi gick igenom koden och rensade den från

bortkommenterad kod samt skrev dit extra kommenterar där vi upplevde att koden behövde förklaras lite bättre.

2.6 Problem

Ett av våra problem var att endast en av oss båda hade sysslat med programmering i Java tidigare. Vår första förhoppning var att J2ME skulle var lika enkelt att lära sig som Java. Anledningen till att Java var enkelt att lära sig berodde mycket på Suns egna hemsida med det medföljande API:t, men för de klasser som behandlar J2ME lyckades vi inte hitta något API i början vilket försvårade lärandet avsevärt. Som tur var fann vi till slut dokument på Nokias Forum [2] som vi kunde använda oss av.

Ett annat problem vi råkade ut för var att vi inte lyckades få den version av JBuilder, som vi tänkte använda oss av från början, att fungera på rätt sätt. Detta tog upp ganska mycket tid men till slut så prövade vi att installera en nyare version och genast blev det enklare att få saker och ting att fungera som de skulle.

Till en början då vi implementerade och testade spelet i en emulator så gjorde vi detta med hjälp av de klassfiler som skapades istället för att skapa en JAR/JAD-fil och exekvera via den. När vi testade JAR-filen för första gången uppstod ett problem eftersom spelet inte kunde exekveras. Detta åtgärdades ganska lätt efter att vi fått reda på att klasserna fullCanvas och commandListener inte fungerar tillsammans. Ett annat fel vi hade var att vi hade använt oss av fel emulator från början. När vi kom till ett möte med Carbello för att testa applikationen för första gången blev vi tillsagda att använda en 7210-emulator istället för den Serie 40-emulator som vi använde eftersom 7210-emulatorn bättre återspeglar hur en riktig mobiltelefon fungerar. Som tur var behövde vi endast ändra på en rad i koden för att få applikationen att fungera med 7210-emulatorn men det tog ett litet tag att komma underfund med vad felet var.

Figur 3 7210-emulatorn

Ytterligare en sak som försvårade projektet var att vi satt på olika platser (Karlstad och Arvika) under inledningen av projektet. Det fungerade rätt bra när vi studerade hur programvaran och J2ME fungerade men det tog ganska lång tid för oss då vi med hjälp av ICQ och telefon skulle försöka sammanställa våra idéer i olika dokument. Vi märkte speciellt då det nya klassdiagrammet skapades hur mycket smidigare det var att arbeta tillsammans på samma ställe.

Tiden verkar kanske inte ha varit ett särskilt stort problem eftersom vi har legat bra till enligt tidsplanen men det har varit ganska jobbigt för oss att få detta att fungera. Att det varit jobbigt beror bland annat på att Claes varit tvungen att lägga ner mycket tid på att hjälpa sin flickvän som startat eget företag och på att Carl-Magnus läser en annan utbildning i Arvika, vilken upptar 40 timmar i veckan. Förutom detta så tränar vi båda flera gånger i veckan och detta har lett till att det mesta av exjobbet har utförts under kvällar och helger, vilket har varit ganska påfrestande för oss.

3 Resultat

Vårt examensarbete var mycket lärorikt att utföra då vi hela tiden erhöll nya kunskaper. Det gav oss en inblick i de olika stegen vid tillverkningen av en mobilapplikation. Då vi oftast inte befann oss på samma ställe var vi tvungna att utveckla ett system för samarbete på distans. Detta gav oss ökad förståelse för vikten av planering vilket vi tycker att vi har lyckats bra med och med hjälp av tekniska hjälpmedel så som ICQ och telefon har vi hela tiden lyckats styra

projektet åt rätt håll. En hel del långa heldagar då vi har träffats, antingen i Arvika eller i Karlstad, har också förekommit.

3.1 Spelidé

Det första som behandlades i projektet var spelidén där vi fick en chans att utveckla spelets karaktär. Spelidén är grunden till hela spelet och dess logik. För en komplett beskrivning av spelidén hänvisar vi till kravspecifikationen i bilaga B.

3.2 Kravspecifikation

Den skapade kravspecifikationen innehåller spelidén samt kraven som sattes av Carbello. Kraven från Carbello innehöll bland annat implemenationskrav t.ex. att spelets JAR-fil inte fick uppgå till mer än 64 kB. För kompletterande information om kravspecifikationen hänvisar vi till bilaga B.

3.3 Designdokumenten

När kravspecifikationen var klar kunde vi sätta oss ner och färdigställa ett klassdiagram. I vårt ursprungliga designdokument ingick dels klassdiagrammet men också de metoder, samt dess inparametrar, som skulle ingå. Detta ledde till att designdokumentet utvecklades under nästan hela implementationsfasen medan klassdiagrammet i stort sett behöll sitt ursprungliga utseende. De enda ändringarna vi gjorde på klassdiagrammet var i början av designfasen efter ett påpekande av Carbello att vid mobilprogrammeringen ska så få klasser som möjligt användas och att vi i slutet av implementationsfasen lade till en klass som ritade upp och tog hand om huvudmenyn och alla undermenyer. I slutet av implementationsfasen beslutade vi även att det var bättre att ha alla beskrivningar av klasser, metoder och parametrar i koden för att på så sätt undvika redundans. Designdokumenten består av bilaga C och D.

3.4 Spelet

Den slutliga koden för spelet bestod av drygt 3700 rader kod (inklusive beskrivningar och kommentarer). Vi lyckades nå de mål som vi tillsammans med Carbello hade dokumenterat i kravspecifikationen. Tyvärr hade vi inte tid till att ytterligare utveckla spelet utöver den grundläggande spelidén men vi är ändå nöjda med det spel vi skapade. Det kändes bättre att

lägga ner extratid på att finslipa grundidén än att börja planera för nya funktioner som troligtvis ändå inte skulle ha hunnit bli färdigställda i tid.

Utseendet på spelet (se figur 4, 5, 6 och 7), rent grafiskt sett, motsvarade väldigt väl de konceptskisser vi hade skapat och som återfinns i kravspecifikationen.

Figur 4 Spelvy

Figur 5 High score

Figur 6 Inmatning av namn

Figur 7 Inställning av startnivå

4 Analys och rekommendationer

Detta kapitel består av rekommendationer för projekt av samma typ som vårt. Samtliga sektioner innehåller också en analysering av hur vi förhållit oss till dessa rekommendationer då vi skapade vårt spel.

4.1 Arbetssätt

Finns det möjlighet så är det alltid bättre att samtliga personer involverade i ett projekt sitter på samma ställe. Detta underlättar kommunikationen och gör att det alltid finns personer att diskutera med då problem eller frågor uppstår.

Vid ett projekt där medlemmarna måste eller väljer att arbeta åtskilda är det oerhört viktigt att tidigt göra upp en planering för vem som ska göra vad och när detta ska göras. Sker implementeringen på olika ställen är det en fördel att använda sig av ett versionshanterings-system för att undvika att flera personer arbetar med samma fil samtidigt vilket kan leda till att kod går till spillo av misstag.

Vi har suttit på olika ställen och träffats endast vid enstaka tillfällen för att programmera samt diskutera våra olika framsteg. När vi kodade försökte vi sätta upp delmål som skulle lösas, detta för att enklare få en överblick över vad som för tillfället skulle göras. Tillvägagångssättet med delmål föll ut mycket väl för det gav en positiv feedback då projektet hela tiden gav känslan av att röra sig framåt. Eftersom vi enbart var två personer som utförde implementeringen ansåg vi att vi inte behövde använda oss av något versionshanteringssystem men så här i efterhand känns det som att det hade varit smidigare att utnyttjat ett sådant då vi hade sluppit att skicka filer fram och tillbaka mellan varandra.

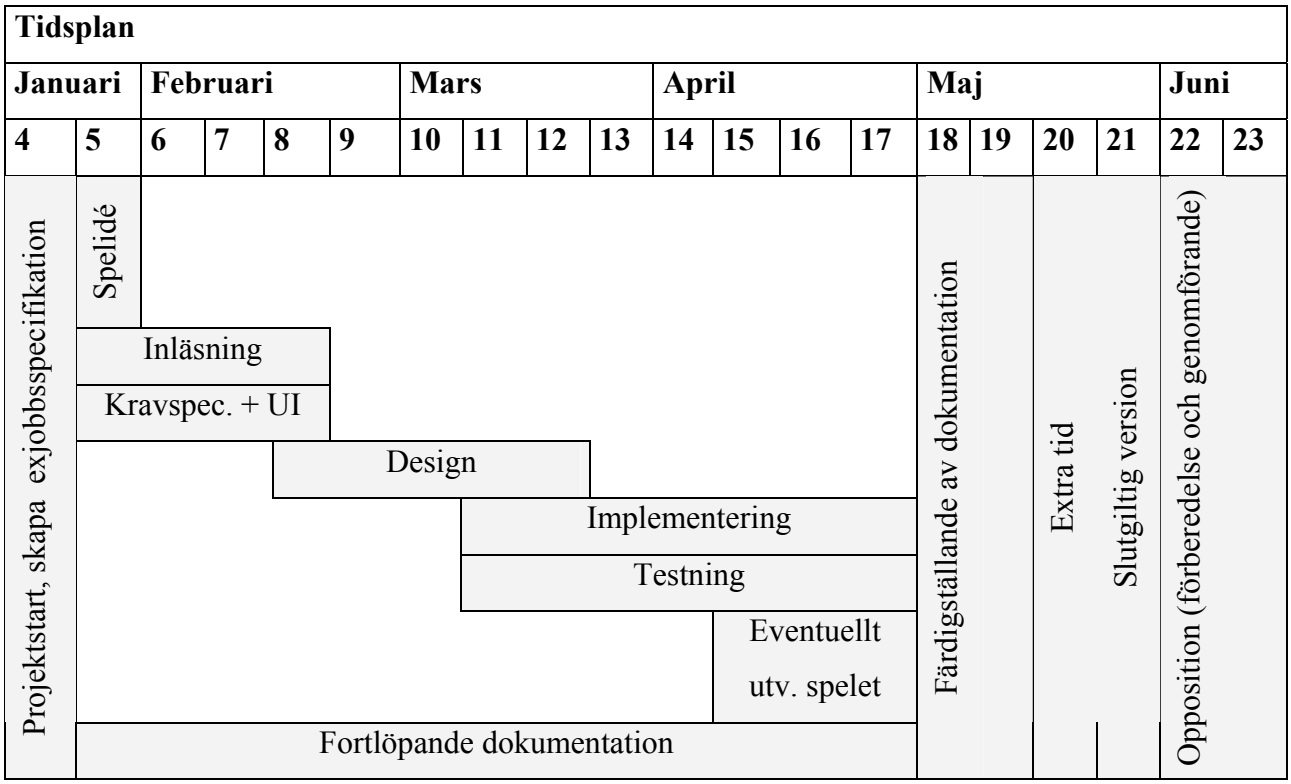
4.2 Dokumentation

I ett stort projekt som detta skapas en mängd olika dokument. Vi kommer i denna sektion att gå igenom vilka dokument vårt projekt består av och vad som är viktigt att tänka på då de skapas.

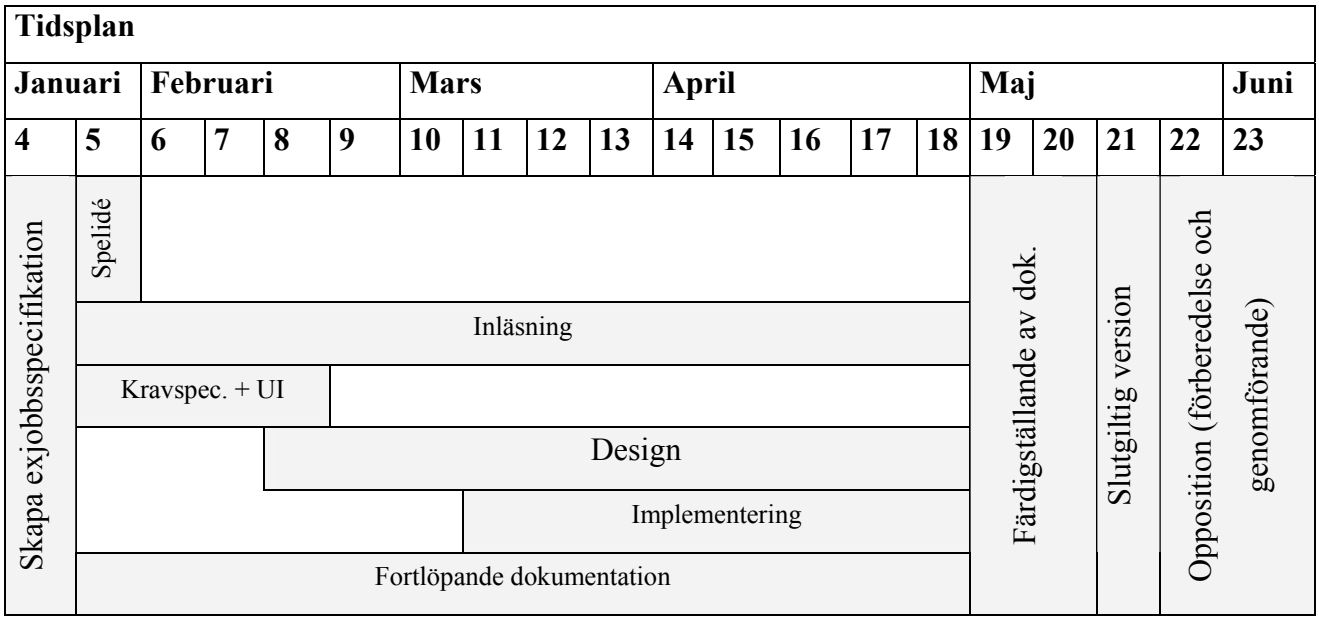
4.2.1 Tidsplan

En tidsplan är en planering som talar om vad som ska göras och i vilken ordning det ska utföras. Tidsplanen skapas med hänsyn till vilka resurser som finns och vad som krävs för att ett projekt ska kunna färdigställas. Det är viktigt att göra en bra tidsplanering eftersom den klart och tydligt visar vilka som ska jobba med vad och när de ska göra det. Skulle ett projekt bli försenat kan detta tidigt upptäckas med hjälp av en tidsplanering och lämpliga åtgärder kan vidtas.

Allt eftersom arbetet med examensjobbet fortskred upptäckte vi att vissa delar av den ursprungliga tidsplanen inte motsvarade hur det egentliga arbetet utfördes.



Tabell 2 Den ursprungliga tidsplanen



Tabell 3 Tidsplan baserat på hur projektet verkligen utfördes

De skillnader som uppstod var att inläsningen pågick under hela produktionen och inte enbart under fyra veckor som vi först hade planerat. Allt eftersom implementeringen

framskred uppstod nya problem och frågor som krävde att vi fick söka och studera information för att kunna ta oss vidare i produktionen.

Designen upptog även den mer tid än vi hade räknat med och fortsatte under hela implementeringen. I det stora hela så bibehölls dock klassdiagrammet i sin ursprungliga form även om ytterligare en klass lades till i slutet av implementationsfasen. Den stora skillnaden var egentligen att metoderna som användes hela tiden justerades, samtidigt som nya metoder lades till medan andra togs bort.

Testningen som vi hade planerat in i tidsplanen blev inte av. Vi gjorde i och för sig egna tester under tiden som vi implementerade spelet samt att vår handledare på Carbello prövade spelet några gånger då vi träffade honom för att få hjälp och feedback men detta var inte den sorts testning som vi från början planerat vilken gick ut på att en utomstående person skulle undersöka om spelet fungerade på rätt sätt enligt kravspecifikationen.

Den tid för eventuell utveckling av spelet som angetts i den ursprungliga tidsplanen försvann helt och hållet eftersom vi inte fick någon tid till att utveckla spelet utöver den kravspecifikation som vi hade.

Slutförandet av implementeringen och överlämnandet av koden till Carbello skedde en vecka senare än planerat. Detta berodde helt och hållet på att vi räknade fel på veckorna och trodde att vi befann oss en vecka längre bak i planeringen än vad vi i själva verket var.

I det stora hela tycker vi att vår tidsplan har fungerat bra och att den efterföljts relativt väl trots att den förändrats en del. Vi har dock kommit fram till att design och eventuell inläsning pågår under hela produktionen från det att de väl påbörjats.

4.2.2 Spelidé

Finns en väl genomarbetad och detaljerad dokumentation av spelidén underlättas själva implementationen oerhört mycket. Programmerare får en helhetsbild av spelet och slipper t.ex. att fundera på vad som ska hända vid olika situationer i spelet och kan istället koncentrera sig på hur det ska lösas. Utan en bra dokumentation av spelidén kan problem uppstå om en ny person ska sätta sig in i spelet för att kunna arbeta med det eller om någon som arbetar med spelet, och kanske har stora delar av spelidén i huvudet, slutar. Det lönar sig

alltså att lägga ner mycket tid på att dokumentera spelidén eftersom det kommer att leda till mindre missförstånd och komplikationer då spelet väl implementeras.

I dokumentationen av spelidén ska konceptskisser av t.ex. spelobjekt, kartor och spelvyer finnas med för att på så sätt öka förståelsen för själva spelidén och utformningen av spelet.

Vi anser att vår dokumentation av spelidén blev mycket bra och att det är relativt enkelt för utomstående att sätta sig in i hur spelet fungerar. Anledningen till detta beror till stor del på att Carbello hade gjort en enkel prototyp i Shockwave som de visade för oss och att vi kunde studera redan utgivna spel som fungerade på samma sätt som det spel Carbello ville att vi skulle göra.

4.2.3 Flödesschema

Ett flödesschema visar på ett enkelt och tydligt sätt hur olika delar av en applikation hänger ihop, t.ex. vad som ska hända då ett alternativ väljs i en meny. Detta bidrar till att utomstående enkelt får en förståelse för hur och varför systemet är uppbyggt på ett visst sätt.

Det flödesschema som vi skapade är uppbyggt på samma sätt som de Carbello använder, vilka i sin tur är baserade på riktlinjer [3] som är framtagna av företaget Nokia.

4.2.4 Designdokument

Designdokumentet visar hur systemet som spelet ska byggas på ska implementeras, främst genom ett klassdiagram. Vi började med att försöka specificera de olika klasserna i spelet samt vilka metoder de skulle kunna tänkas använda sig av. Detta är ett bra förfaringssätt som underlättar den kommande implementeringen eftersom programmerarna inte behöver lägga ner så mycket tid på att tänka ut klassers relationer med varandra samt vilka metoder som behöver skapas.

4.3 Implementation

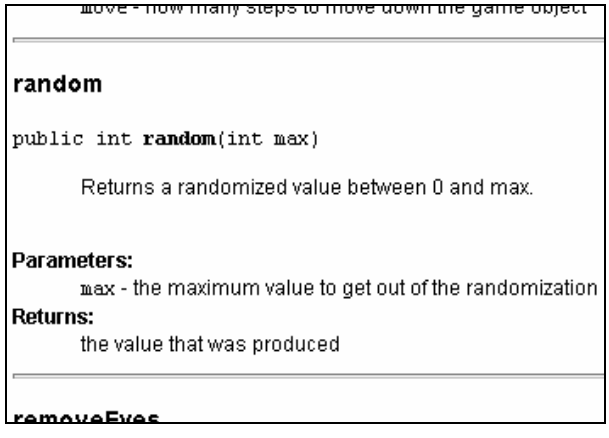
I denna sektion behandlas detaljer som är viktiga att tänka på då applikationer för mobiltelefoner skapas. Först tar vi upp hur koden bör dokumenteras och sedan följer tips och detaljer om vad och hur utvecklare bör göra då de skapar mobiltelefonspel.

4.3.1 Kommentarer i koden

När en utvecklare läser någon annans kod är det mycket viktigt att denna kod är väl dokumenterad så att han eller hon snabbt kan förstå vad koden gör. Vi har beskrivit alla klasser, metoder och medlemsattribut med Javadoc eftersom vi tycker att det är ett mycket bra sätt att dokumentera kod. Förutom beskrivningarna som syns i filerna tillsammans med koden genererar Javadoc ett dokument som är lättöverskådligt och relativt enkelt att använda. I JBuilder kan dokument som skapats med Javadoc öppnas genom att en metod markeras och tangenten F1 på tangentbordet trycks in vilket gör att Javadoc passar utmärkt för dokumentation i denna utvecklingsmiljö.

```
/**
 * Returns a randomized value between 0 and max.
 * @param max the maximum value to get out of the randomization
 * @return the value that was produced
 */
public int random(int max){
  return Math.abs(GameManager.randomObj.nextInt() % (max + 1));
}
```

Figur 8 Exempel på hur Javadoc används i koden



Figur 9 Dokument genererat med Javadoc

4.3.2 Antalet klasser

Vid skapandet av mobilapplikationer ska så få klasser som möjligt användas (helst inte flera än fem eller sex stycken). Anledningen till detta är dels att det skapas en del data i varje klassheader i de kompilerade klassfilerna och dels för att många små filer blir större än en stor i JAR-filen på grund av att filerna inte packas ihop med varandra utan var och en för sig. Vid

skapandet av applikationer för mobiltelefoner är det av prestandaskäl viktigt att de storleksmässigt görs så små som möjligt och ju färre klasser det finns desto mer kod får plats.

Först skapade vi ett klassdiagram som bestod av alldeles för många klasser men efter att ha pratat med vår handledare på Carbello reducerade vi antalet klasser till fem. Det slutliga spelet bestod dock av sex klasser istället för de fem som vi trodde oss behöva. Anledningen var att vårt spel tog liten plats och att vi därför kunde kosta på oss att lägga till en klass som tog hand om alla menyer. Hade vi inte haft så gott om plats kunde vi ha lagt in funktionaliteten för menyerna i samma klass som hade hand om spelvyn och spellogiken men det skulle ha inneburit att koden i den klassen blivit väldigt komplex och svårbegriplig. Vi tycker att vi gjorde helt rätt när vi lade till den nya klassen.

4.3.3 Olika färgpaletter för en bild

Om bilder används upptar dessa relativt stor plats jämfört med koden i sig. Det gäller att tänka på att inte använda fler bilder än absolut nödvändigt. Behövs två eller flera bilder som ser likadana ut fast med olika färger så räcker det med endast en bild. Den aktuella bilden måste först indexeras vilket enkelt kan utföras i ett bildbehandlingsprogram t.ex. Adobe Photoshop eller Paint Shop Pro. När detta är klart måste en metod skapas som läser in en bild och ersätter dess färgpalett med en annan palett.

Figur 10 Den högra blobben har skapats av bilden på den vänstra blobben

Carbello hade skapat och använde sig av ett par metoder för att läsa in bilder med olika färgpaletter och de delade med sig till oss av den kod som behövdes för detta. I början förstod vi inte riktigt hur koden skulle användas men efter att ha fått hjälp av vår handledare fungerade allt mycket smidigt. De alternativa färgpaletterna som vi använde skapades med hjälp av hexadecimala tal som vart och ett motsvarar en färg t.ex. motsvarar talet FF0000 färgen röd.

4.3.4 Flera bilder i en bildfil

Genom att placera flera bilder bredvid varandra i en fil behövs mindre plats för att lagra bilderna. Varje bildfil består nämligen av en header med information om filen och placeras

flera bilder i samma fil så behövs endast en header istället för flera vilket upptar mindre plats. För att sedan rita ut en del av en bild måste den i Java inbyggda metoden setClip användas. Med denna metod anges vilken del av displayen som får ritas om och sedan måste bilden ritas ut på en position som gör att rätt del av bilden hamnar på den del av displayen som ritas om.

Figur 11 Bilderna på alla olika tillstånd en blob kan anta har lagts i samma fil

Vi placerade samtliga bilder på blobbarnas olika tillstånd och samtliga siffror i en fil vardera vilket minskade den totala storleken med ett par kB. Ytterligare en fördel med detta var att det gick snabbare att läsa in bilderna på blobbarnas tillstånd med olika färgpaletter eftersom endast en fil per färg lästes in till skillnad mot 17 filer per färg som det var innan.

4.3.5 PNGCrush

PNGCrush är ett program som används för att komprimera bildfiler av typen png utan att bildernas kvalitet försämras. Vi har använt detta program på alla bilder som inkluderats i spelet och storleken på alla bilderna har förminskats med minst 20 procent.

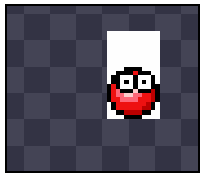
4.3.6 Linjer och fyrkanter

Linjer och fyrkanter kan skapas och ritas ut på displayen med hjälp av Javas inbyggda metoder vilka i vissa fall kan användas istället för bilder. Anledningen till att använda t.ex. en linje istället för en bild beror helt på att linjen är en inbyggd metod som inte tar någon plats till skillnad från bilden som måste skickas med och sparas på mobiltelefonen.

I vår meny där spelaren kan ange viken nivå spelet ska starta på använde vi från början ett antal bilder. Detta gjorde att uppdateringarna av menyn gick relativt långsamt men efter att ha tagit bort bilderna och istället använda de inbyggda metoderna för att rita linjer och fyllda rektanglar skedde uppdateringarna mycket snabbare.

4.3.7 Uppdatering av displayen

Det gäller att inte uppdatera mer än nödvändigt av spelplanen eftersom det kan leda till att spelet blir segt och besvärligt att kontrollera. För att undvika detta kan metoden setClip, som beskrivs i sektion 4.3.4 ”Flera bilder i en bildfil”, användas för att endast uppdatera en viss del av spelskärmen. Detta kan vara ett relativt enkelt sätt att få spel att flyta bättre även då mycket grafik visas på spelplanen på en och samma gång.



Figur 12 Endast det som motsvarar den vita ytan uppdateras då blobben faller

Då vi för första gången prövade att spela vårt spel på en mobiltelefon upptäckte vi att det gick väldigt segt då många blobbar befann sig på spelplanen på en och samma gång. Efter att ha blivit tipsade av vår handledare på Carbello om att använda setClip justerade vi metoderna som ritade ut spelobjekten på spelplanen vilket resulterade i att endast de delar där grafiken behövde uppdateras ritades om. Detta ledde till slut till att spelplanen kunde fyllas med maximalt antal spelobjekt utan att spelet blev segt.

4.3.8 Game actions

J2ME har ett inbyggt stöd för förflyttning i sid- och höjdled vilket tillåter tillverkare att ange specifika knappar till vissa game actions (spelhandlingar som ofta utförs i spel) t.ex. kan knappen 2 sättas till "upp", knappen 8 till "ner" osv. Genom att använda den inbyggda metoden getGameAction kan de koder för olika knappar som skickas till metoderna keyPressed, keyReleased och keyRepeated göras om till game actions vilket underlättar för utvecklare av spel.

Vi använde oss av game actions i vårt spel och det fungerade mycket bra. Det game action som motsvarar uppåt användes för att rotera blobbarna.

4.3.9 Språk

Det är viktigt att tänka på att skapa applikationer så det är enkelt att lägga till och ändra språket som används. I dag kan applikationer snabbt och lätt spridas till ett flertal länder och trots att engelska förstås på de flesta platser så finns det dock platser där det inte förstås. Ett smidigt sätt att lösa detta är genom att använda sig av en klass där alla texter som används i applikationen läggs in på olika språk. Sedan anropar men helt enkelt klassen för att få tag i en text på rätt språk.

Vårt spel använder sig enbart av engelska men vi har skapat en klass enligt ovan, vilket gör att det är enkelt att lägga till andra språk om det behovet uppstår.

4.3.10 Testa kontinuerligt

Det är viktigt att inte bara skriva en massa kod utan att tänka sig för eller utan att testa kontinuerligt, eftersom detta kan leda till att en mängd följdfel lätt uppstår, vilket i sin tur kan leda till onödigt många timmars jobb för att rätta till felen. När det gäller mobiltelefoner är det viktigt att inte enbart testa på datorn med hjälp av en emulator, utan också att testa på en riktig mobiltelefon. Detta beror på att emulatorn inte alltid simulerar verkligheten exakt som den är.

Vi fick tyvärr inte tillgång till en telefon förrän i slutet av projektet med resultatet att vi råkade ut för en del fel som inte uppstod då vi använde oss av Nokias emulator. Som tur var så var dessa fel relativt lättlösta och vi slapp ägna en massa tid åt att leta efter dem.

4.3.11 Text

Den inbyggda metoden drawString i Java bör användas återhållsamt eftersom den kräver mycket processorkraft i förhållande till vad den gör. Ibland kan det faktiskt vara värt att fundera på om det inte är bättre att använda bilder ifall drawString måste utföras ett flertal gånger.

Vi använde drawString till att skapa kantlinjer på text vilket gjorde att varje text skrevs ut fyra gånger extra. På vår emulator flöt det fint men när vi testade på mobiltelefonen upplevde vi en markant skillnad. Ett lysande exempel på detta är skärmen för inmatningen av namn där det gick oerhört segt då vi använde oss av drawString. Vi löste dock detta genom att skriva om metoden som ritade ut de olika bokstäverna så att endast två bokstäver utan kantlinjer ritades ut varje gång skärmen uppdaterades.

4.3.12 Multiplikation

Multiplikation tar mycket av mobiltelefonernas processorkraft eftersom processorn egentligen adderar tal flera gånger för att få fram rätt resultat. Ett snabbare sätt att åstadkomma samma sak som med multiplikation är att skifta (<<) vilket är cirka 30 till 50 procent snabbare då en konstant skiftas och upp till 500 procent snabbare då en variabel skiftas. Den enda begränsningen med skiftning är egentligen att den bara kan användas istället för multiplikationen då talet som ska multipliceras befinner sig i talmängden 2^n (där n är ett heltal) dvs. 2, 4, 8, 16 osv. Skiftning används alltså enligt följande:

$$tal << n <=> tal * 2^n$$

Skiftning kan också användas för division (>>) och sker då enligt följande:

$$tal >> n <=> tal / 2^n.$$

Från början använde vi oss endast av multiplikation och division men efter att ha pratat med vår handledare på Carbello ändrade vi till skiftning på de ställen där det gick. Vi upplevde dock ingen direkt skillnad eftersom vi inte hade några multiplikationer eller divisioner med stora tal.

5 Slutkommentar

I ett projekt som detta är det viktigt att vara lyhörd och att ha ett bra samarbete. Vi har, trots svårigheter att få ihop ett gemensamt arbetsschema, lyckats mycket bra och fått fram en produkt som fått bra respons. Från början trodde vi att det skulle bli väldigt svårt att lära sig ett nytt programspråk men det visade sig att vi med lite hårt arbete relativt lätt kunde klara av det. Vi är mycket nöjda med vårt resultat som gett oss ökad förståelse för utveckling av applikationer för mobiltelefoner och vi skulle inte ha något emot att i framtiden jobba i denna bransch.

Referenser

[1] Java 2 Platform, Micro Edition (J2ME) http://java.sun.com/j2me/ (2004-03-03)

[2] Nokia Forum http://forum.nokia.com (2004-03-05)

[3] Creating MIDlets with Borland JBuilder X
http://www.forum.nokia.com/ndsCookieBuilder?fileParamID=4203 (2004-04-29)

[4] MIDP and Game UI
http://www.forum.nokia.com/ndsCookieBuilder?fileParamID=2867 (2004-04-29)

[5] Java Tutorials: Part 6: Shifts and relations
http://www.hubcanada.com/story_5399_25 (2004-04-29)

Bilaga A Ordlista

- **Alert:** En varning som erhålls på mobilen då systemet vill påkalla något viktigt som skett.
- **API:** Application Programming Interface. Ett paket som innehåller en samling standardklasser, vilka kan användas i alla javaprogram, samt specifikationen för hur metoderna och attributen i de olika klasserna och objekten ska kommas åt.
- **CLDC:** Connected Limited Device Configuration. En konfiguration som är anpassad för hårdvara med långsamma processorer och begränsat minne. Används för att programmera applikationer till trådlösa apparater.
- **Core:** Se ”kärna”.
- **Emulator:** En applikation som simulerar en hårdvara. I vårt fall simulerade emulatorerna diverse mobiltelefoner av märket Nokia.
- **J2ME:** Java™ 2 Platform, Micro Edition. Med denna teknik kan Java-applikationer till mobiltelefoner och handdatorer utvecklas. J2ME är en bantad version av företaget SUN’s plattform Java2.
- **JAD:** En JAD-fil innehåller en beskrivning av innehållet i JAR-filen, länken vart det kan laddas ner.
- **JAR:** En JAR-fil innehåller alla komprimerade filer som behövs för att köra en applikation.
- **Javadoc:** Ett sätt att kommentera kod för javaprogram enligt en given standard.
- **JBuilder:** Ett utvecklingsverktyg för Java skapat av Borland.
- **JDK:** Java Development Kit. En JDK är en samling verktyg för utveckling av applikationer och funktioner i Java.
- **Kärna:** Javas grundläggande funktionalitet finns i kärnan.
- **Metod:** Javas motsvarighet till funktion i t.ex. C och C++.
- **Micro Edition:** Se J2ME
- **MIDlet:** En applikation skriven med J2ME och som använder sig av MIDP, skrivs ibland även som midlet.
- **MIDP:** The Mobile Information Device Profile. Högnivå-APIs som används till mobiltelefoner och enkla typer av handdatorer. Används för applikationens livscykel, användargränssnitt, nätverkskontakt och åtkomst för egenskaper hos den applikation som utvecklas.

- **Paket:** Ett paket i Java innehåller klasser som har ett samröre med varandra. Genom att inkludera ett paket ges tillgång till alla klasser i det paketet.
- **PNG:** Portable Network Graphics. Ett format för att lagra grafik. Stödjer bilder med indexerad färg, gråskalor, true colour och en alphakanal för transparens. Formatet gör bilderna väldigt komprimerade och därmed passar det väldigt bra för t.ex. mobiltelefoner.
- **RecordStore:** Minnesutrymme som används för att spara information från en applikation. Minnesutrymmet är ett beständigt minne vilket innebär att informationen finns kvar även då mobiltelefonen stängs av.
- **Tetris:** Ett spel som går ut på att skapa fyllda rader med hjälp av olikformade objekt varvid raden försvinner.
- **Versionshanteringssystem:** Distribuerat system som gör att flera användare kan jobba med samma filer utan att data riskerar att gå förlorad t.ex. genom att filer sparas över.
- **Virtuell maskin:** Den virtuella maskinen tar hand om all javakod vilket gör att koden blir plattformsoberoende. Det spelar alltså ingen roll om koden körs på en PC, Macintosh eller mobiltelefon.
- **Wireless:** Trådlös, bärbar enhet.

Bilaga B Kravspecifikation för Blobs

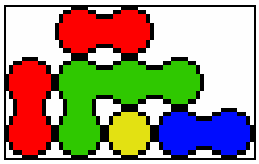
Detta dokument omfattar spelets grundidé och de tekniska aspekterna som gäller för projektet. Alla bilder i dokumentet visar grafik som ska användas i spelet, med undantag för bilder med vit bakgrund.

Innehåll

Bilaga:
Flödesschema

1. Beskrivning

Spelet går ut på att samla så mycket poäng som möjligt genom att få minst fyra blobbar av samma färg att ligga i direkt anslutning till varandra (se bild nedan). Om fyra eller fler blobbar ligger i direkt anslutning med varandra så tas dessa bort. I takt med att poängen ökar så ökar också spelets hastighet.



De fyra gröna fälten ligger i direkt anslutning till varandra

Samtidigt som man samlar poäng så måste man försöka undvika att någon blob hamnar ovanför den övre kanten av spelplanen. När en blob nått den övre kanten och ytterligare en blob hamnar i samma kolumn så är spelet slut.

Om spelaren samlat tillräckligt med poäng under speltillfället så hamnar han/hon på en high score-lista. Detta gör att spelet får en längre livslängd eftersom spelaren alltid har möjligheten att ta sig in på high score-listan.

2. Spelkomponenter

De komponenter som ingår i spelet förklaras under denna rubrik.

2.1 Meny

Här visas spelets meny med de olika alternativ man kan välja.

Spelets meny

* Detta alternativ visas endast om en spelomgång inte har avslutats. Väljer man "Continue" så fortsätter man på denna spelomgång. Om man istället väljer alternativet "New Game" så försvinner den oavslutade spelomgången och en ny omgång börjar.

2.2 New Game

Detta är den skärm (spelvy) som visas då spelet spelas i singel player mode.

Spelvy

2. 2. 1 Nästa-rutan

Här visas vilken färg de två kommande aktiva blobbar kommer att ha.

2. 2. 2 Nivårutan

Här visas vilken nivå spelaren befinner sig på.

2. 2. 3 Poängrutan

Här visas den aktuella poängställningen då spelet spelas.

När vissa poäng uppnåtts så kommer man till nya nivåer vilket leder till att hastigheten, med vilken de aktiva blobbarna faller, ökar.

2. 2. 4 Spelplan

Det är på spelplanen som blobbarna faller ner och det är alltså här som man ska placera dem så att de tas bort och man får poäng.

Spelplanen rymmer sex blobbar på bredden och tio på höjden vilket leder till att spelplanen kan delas in i sju kolumner och tio rader.

2. 2. 4. 1 Mark

Kanten längst ner på spelplanen kallas för marken.

2. 2. 4. 2 Vägg

De båda kanterna på vänster och höger sida av spelplanen kallas för väggar.

2. 2. 5 Blob

Blob är en bolliknande figur som finns i fem olika färger (blå, grön, gul, röd och lila). Varje färg är värd lika många poäng.

Färgen på en blob bestäms med hjälp av en slumpgenerator där sannolikheten för varje färg är lika stor.

2. 2. 5. 1 Ögon

Varje blob har ett par ögon men om två eller flera blobbar ligger i direkt anslutning till varandra så reduceras antalet ögon till ett par för hela anslutningen. Dessa ögon kommer att vara placerade på den blob som varit på spelplanen längst.

De båda aktiva blobbarna har alltid varsitt par ögon. Om de båda aktiva blobbarna har samma färg och de hamnar bredvid varandra, utan att ansluta till någon blob av den färgen som redan finns på spelplanen, så placeras ögonen på den av de aktiva blobbarna som var CB.

När man flyttar de aktiva blobbarna i sidled så tittar deras ögon åt det håll blobbarna rör sig.

Samtliga ögon på spelplanen blinkar vid oregelbundna tillfällen baserat på en slumpgenerator. Placeringen av ögonen på de blobbar som finns på spelplanen och som inte är aktiva blobbar kommer inte att ändras.

2. 2. 5. 2 Center blob (CB)

Den av de båda aktiva blobbarna som var placerad till vänster då de kom in på spelplanen.

2. 2. 5. 3 Rotation blob (RB)

Den av de båda aktiva blobbarna som var placerad till höger då de kom in på spelplanen.

2. 2. 5. 4 Aktiva blobbar

De två blobbar som faller ner på spelplanen ovanifrån kallas för aktiva blobbar och kommer in på spelplanen i kolumn tre och fyra. Blobbarna är placerade bredvid varandra och kan flyttas i sidled och/eller roteras för att anta olika positioner. Förflyttning i sidled, en kolumn i taget, kan utföras om det inte finns någon blob eller vägg intill de aktiva blobbarna i deras färdriktning. Rotationer sker 90° medurs runt CB. För att en rotation ska kunna genomföras får ingen blob eller vägg finnas på den position dit RB ska förflyttas.

När någon av de båda blobbarna har placerats på marken, på en glaskula eller ovanpå någon annan blob så är de inte längre aktiva. Om nu någon av de båda blobbarna inte är placerade på marken, på en glaskula eller på någon blob så kommer den att kopplas loss från den andra blobben och falla rakt ner tills den träffar marken, en glaskula eller en blob.

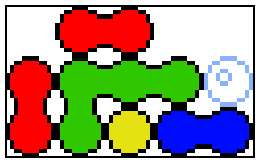
Spelaren kan påverka hastigheten med vilken de aktiva blobbarna faller genom att trycka på eller hålla in nedåt-knappen. Detta leder till att hastigheten sätts till samma hastighet som används vid nivå 9. Genom att släppa knappen så återställs hastigheten till vad den var innan hastighetsökningen.

2.2.6 Glaskula

Varje gång två aktiva blobbar ska tas in på spelplanen så körs först en slumpgenerator för att avgöra om en glaskula ska falla ner på spelplanen eller ej. Sannolikheten för att en glaskula ska uppenbara sig på spelplanen är nivå / 100 och då en glaskula ska falla ner slumpas vilken kolumn den ska falla i. Sannolikheten för varje kolumn är 1 / antalet kolumner. En glaskula kan dock inte hamna i en kolumn där en blob eller glaskula befinner sig på översta raden.

En glaskula

En glaskula faller rakt ner, utan att kunna påverkas, tills den hamnar på en blob, en glaskula eller på marken och det är först då som de aktiva blobbarna kommer in på spelplanen. Till skillnad från blobbar så försvinner inte glaskulor även om de är fyra eller fler i direkt anslutning. Det enda sättet att få en glaskula att försvinna är om en direkt anslutning av blobbar uppstår alldeles intill den.



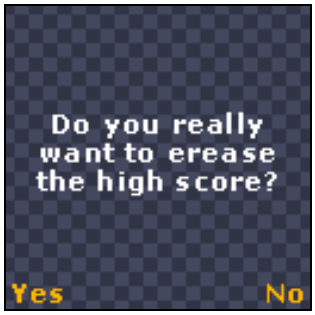
Glaskulan intill den direkta anslutningen försvinner tillsammans med de gröna blobbarna

2.3 High Score

I high score visas en lista med de tre högsta poängen som erhållits i spelet. Tillsammans med varje poäng visas tre bokstäver som har matats in av de spelare som finns med i high score.

High score

High score listan kan raderas om alternativet "Erase" väljs. Har detta alternativ valts så dyker ett meddelande upp där raderingen måste bekräftas.



Bekräftningsruta

Oavsett vilket alternativ man väljer så kommer man tillbaka till high score.

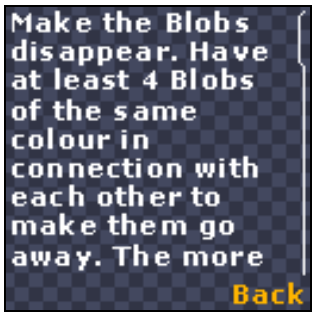
2.4 Settings

Spelaren kan här ställa in vilken nivå spelet ska börja på. Väljer man till exempel nivå 5 så kommer hastigheten, med vilka de aktiva blobbarna faller, redan från spelets start att motsvara den hastighet som gäller för nivå 5. Denna hastighet bibehålls hela tiden tills spelaren når nivå 6 varefter hastigheten påverkas som vanligt.

Här ställer man in vilken nivå man vill starta på

2.5 Instructions

Här förklaras vad spelet går ut på.



Instruktionsskärmen

Texten som ska visas på instruktionsskärmen:
"Make the Blobs disappear. Have at least 4 Blobs of the same colour in connection with each other to make them go away. The more Blobs disappearing at the same time, the more points you'll get. During the game, crystal balls may appear to make it harder for the Blobs to come together. A crystal ball will go away only if a Blob disappears next to it."

2.6 Controls

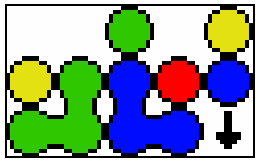
Här visas vilka knappar som används i spelet.

Kontrollskärmen

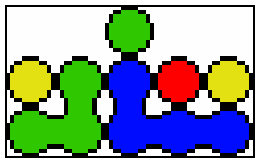
Texten som ska visas på kontrollkärmen:
"Rotate - 2/up
Left - 4/left
Right - 6/right
Drop - 8/down
Pause - right soft key"

3. Spelets grundregler

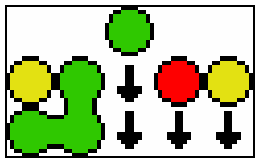
Två aktiva blobbar faller ner på spelplanen. Efter att de båda aktiva blobbarna slutligen har placerats på marken, på en glaskula eller ovanpå någon annan blob så kopplas alla blobbar som angränsar till varandra och som har samma färg ihop. Om fyra eller fler blobbar av samma färg är placerade i direkt anslutning till varandra så tas dessa bort och poäng genereras. När dessa blobbar försvinner så kan det uppstå tomrum under en eller flera blobbar. Finns det blobbar ovanför ett tomrum så faller dessa ner tills de hamnar på marken, på en glaskula eller på en blob. Uppstår en ny direkt anslutning (kallas för kedjereaktion) med fyra eller fler blobbar så tas även dessa bort och poäng genereras. Då ingen ny kedjereaktion uppstår faller två nya aktiva blobbar ner på spelplanen.



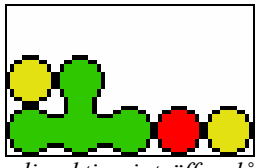
De båda aktiva blobbarna till höger faller ner mot marken



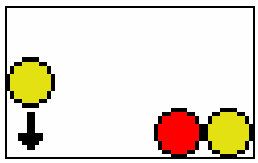
De fyra blåa blobbarna bildar en direkt anslutning



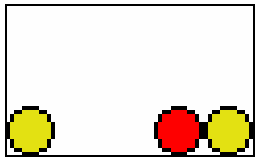
Ett tomrum har uppstått och blobbarna faller ner



En kedjerektion inträffar då fyra gröna blobbar bildar en direkt anslutning



Ett tomrum har uppstått och den gula blobben faller ner



Ingen kedjerektion inträffar vilket betyder att två nya aktiva blobbar kan släppas ner på spelplanen

3.1 Nivåer

Nivå	Min. poäng		Max. poäng
1	0	–	799
2	800	–	2399
3	2400	–	4799
4	4800	–	6999
5	7000	–	10999
6	11000	–	15799
7	15800	–	21399
8	21400	–	27799
9	28800	–	…

För att komma vidare till en ny nivå behöver spelaren samla poäng motsvarande t.ex. 40 direkta anslutningar med fyra blobbar i varje och inga kedjereaktioner. Det finns inte fler än nio nivåer.

Då spelaren kommer till en ny nivå så ökas farten med vilken de aktiva blobbarna faller.

3.2 Poängberäkning

Antal blobbar (minst fyra) i direkt anslutning * 5 poäng * nivå * kedjereaktionsnivå

Då någon av de aktiva blobbarna bildar en direkt anslutning (med minst 4 blobbar) är kedjereaktionsnivån lika med 1. Poäng utdelas. Om en kedjerektion uppstår, på grund av tomrum, ökas kedjerektionsnivån till 2 varefter poäng utdelas. Uppstår en ny kedjereaktion så ökas kedjereaktionsnivån med 1 varefter poäng utdelas. Detta fortsätter tills inga fler kedjereaktioner uppstår.

Varje glaskula som försvinner generar 80 poäng * nivå.

OBS! Poängberäkningen kommer eventuellt att justeras om det i senare skede av projektet visar sig att detta behövs.

3.3 Spelkontroller

Under spelets gång kan man använda sig av fyra olika knappar.

En knapp (2 eller uppåt) används för att rotera de aktiva blobbarna.
Två knappar används för att flytta de aktiva blobbarna i sidled (höger (6 eller höger) resp. vänster (4 eller vänster)).
En knapp (8 eller nedåt) använd för att öka hastigheten med vilken de aktiva blobbarna faller.

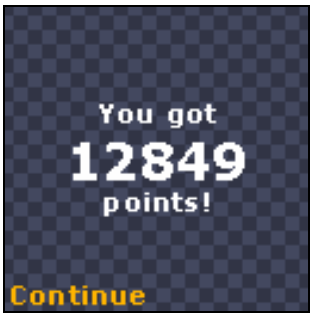
3.4 Paus

Det går att pausa spelet men om spelet befinner sig i pausläge så kan man inte se spelplanen. Anledningen till detta är att spelaren inte ska kunna dra fördel av att se spelplanen utan att de aktiva blobbarna faller ner och på så sätt få obegränsad tid till att undersöka vart det är bäst att placera de aktiva blobbarna.

Om ett spel har pausats så kommer alternativet "Continue" att synas i spelets meny.

3.5 Hur spelet tar slut

När en blob placeras ovanför spelplanens översta rad så är spelet slut. Om spelaren har fått mer poäng än den lägsta noteringen i high score-listan så får han/hon ange tre bokstäver (till exempel spelarens initialer) varefter dessa läggs in i listan.



Denna skärm visas om poängen inte räcker för att ta sig in på high score

Spelaren har fått tillräckligt med poäng för att få vara med på high score och får mata in sitt namn

Om inget namn har skrivits in då alternativ "Continue" valts så kommer bara poängen visas i high score (inget namn visas framför poängen).

4. Angående texter och tecken i spelet

Alla texter och tecken i spelet (med eventuellt undantag av High score och de olika alternativen i menyn) kommer att skapas med hjälp av det inbyggda teckensnittet. Färgen kommer att vara vit alternativt orange/guld.

5. Tekniska aspekter

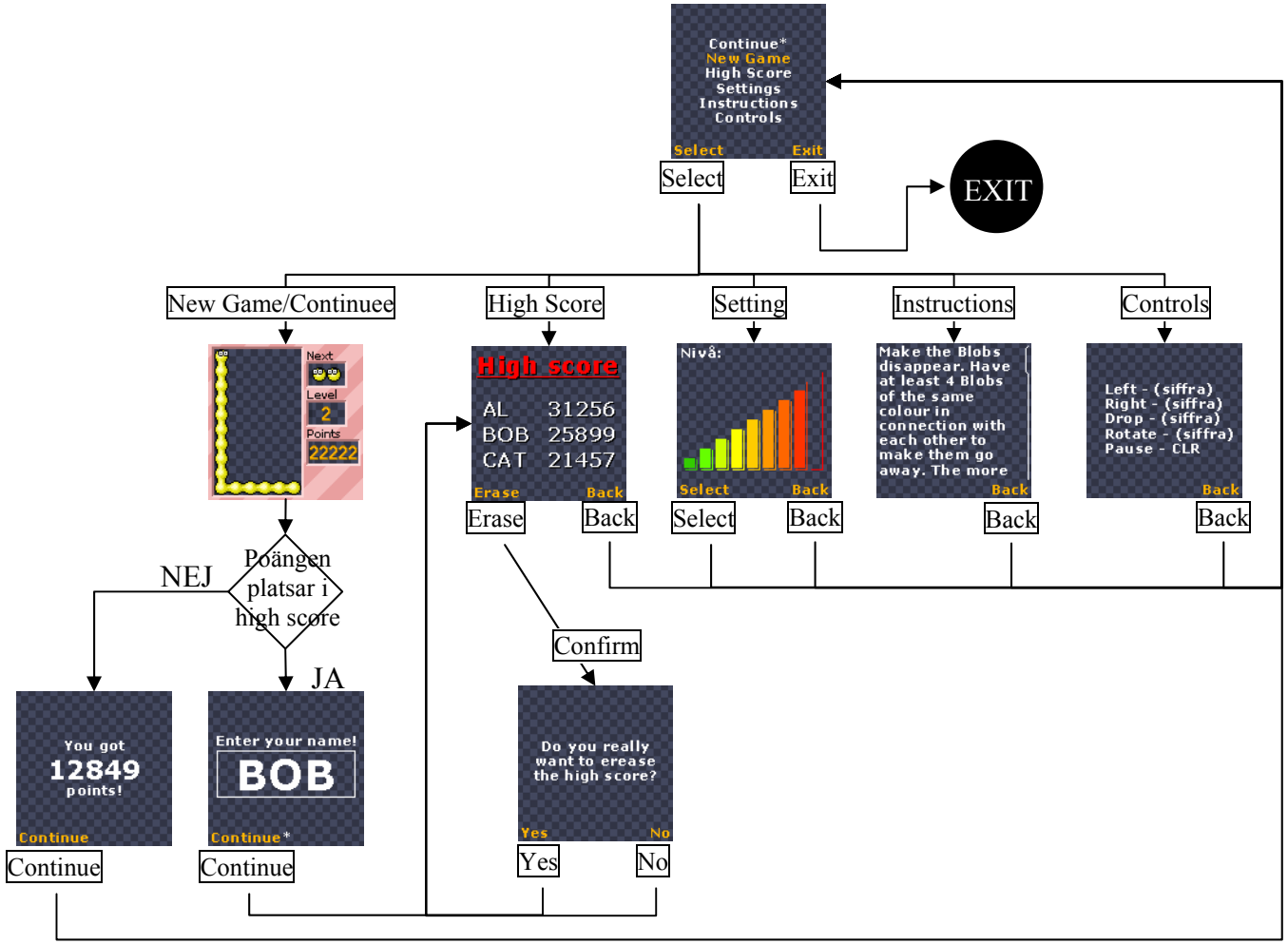
- Spelet ska som ett minimum fungera på Nokias Series 40 - telefoner.
- JBuilder X med Nokia S40 MIDP Concept SDK ska användas. Inga Nokia-klasser får användas, med undantag av metoden/klassen "fullcanvas". Om ljud ska inkluderas i spelet så måste Nokia-klasser användas.
- MIDP1 ska användas.
- Spelets JAR-fil får ej uppgå till mer än 64 kB.

- Koden skall vara skapad på ett sådant sätt att den är lätt att anpassa för andra modeller av mobiltelefoner (inte bara Nokia). Koden ska vara kommenterad på engelska.

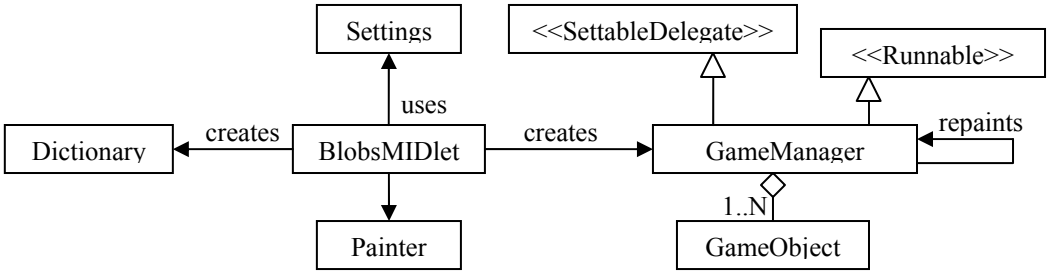
- Språket i spelet ska vara engelska och det ska vara enkelt att ändra.

- Om ljud ska vara med i spelet så kommer Nokia-specifika klasser att användas.

Flödesschema



Bilaga C Klassdiagram



BlobsMIDlet
Basklass som samordnar de övriga klasserna.

Dictionary
Innehåller all text i spelet. Skapad så att nya språk lätt kan läggas till.

GameManager
Sköter själva spelandet och innehåller all logik.

GameObject
Representerar ett spelobjekt (blob eller glaskula).

Painter
Ritar ut och sköter huvudmenyn och samtliga undermenyer.

Settings
Används för att spara undan data på mobiltelefonen.

All beskrivning av metoder etc. finns dokumenterad i koden.

Bilaga D Exempel på Javadoc

blobs

Class GameObject

public class **GameObject**

Title: The game object class.

Description: Symbolises a game object.

Copyright: Claes Röjder & Carl-Magnus Andersson Copyright (c) 2004

Company: Carbello

Version:
1.0
Author:
Claes Röjder & Carl-Magnus Andersson

Field Detail

BLOB

```
public static final int BLOB
```
Constant representing a blob.

CRYSTAL_BALL

```
public static final int CRYSTAL_BALL
```
Constant representing a crystal ball.

START_X

```
private static final int START_X
```
Constant representing the x value where the game field starts.

START_Y

```
private static final int START_Y
```
Constant representing the y value where the game field starts.

IMAGE_SIZE

```
private static final int IMAGE_SIZE
```
Constant representing the width and height of a game object.

---

isBlob

```
private boolean isBlob
```
Variable telling if the game object is a blob or not.

---

hasEyes

```
private boolean hasEyes
```
Variable telling if the game object has eyes or not.

---

active

```
public boolean active
```
Variable telling if the game object is active or not.

---

color

```
private int color
```
Variable used to hold the color of the game object.

---

image

```
private int image
```
Variable representing the image used by the game object.

---

xPos

```
private int xPos
```
Variable that has the x position of the game object.

---

yPos

```
private int yPos
```
Variable that has the y position of the game object.

---

blinkDelay

```
private int blinkDelay
```
Variable that tells how long time the eyes of a gameobject should be closed when it blinks.

---

eye_offset_x

```
public int eye_offset_x
```
Variable that tells how much the eyes x position is different from the body's x position.

---

eye_offset_y

```
public int eye_offset_y
```
Variable that tells how much the eyes y position is different from the body's y position.

Constructor Detail

GameObject

```
public GameObject(int type,
                  int picture)
```
The constructor for a game object. Constructs either a blob or a crystal ball.

Parameters:

`type` - The type of the game object to be created

`picture` - The image used to represent the object

Method Detail

createEyes

```
public void createEyes()
```
Creates eyes on the game object.

---

getColor

`public int` **`getColor`**`()`

Get what color the game object has.

Returns:

the game objects color

---

getEyesStatus

`public boolean` **`getEyesStatus`**`()`

Says if the game object has eyes or not.

Returns:

the status of the eyes

---

getImage

`public int` **`getImage`**`()`

Get the image that represents the game object.

Returns:

the image representing the game object

---

getStartY

`public int` **`getStartY`**`()`

Get the start value in the game field for the game object.

Returns:

the start value

---

setXPosition

`public void` **`setXPosition`**`(int x)`

Sets the game objects x position.

Parameters:

`x` - the x position to set the game object to

---

getXPosition

`public int` **`getXPosition`**`()`

Gets the x position of the game object.

Returns:

the x postion of the game object

---

getYPosition

`public int` **`getYPosition`**`()`

Gets the y postion of the game object.

Returns:

the y position of the game object

---

setYPosition

`public void` **`setYPosition`**`(int y)`

Sets the y position for the game object.

Parameters:

`y` - the y position to set the game object to

---

getRealYPosition

`public int` **`getRealYPosition`**`()`

Gets the y position without the value being shifted.

Returns:

the unshifted y value of the game object

---

getType

`public int` **`getType`**`()`

Gets the type of the game object.

Returns:

the type of the game object.

---

moveLeft

`public void` **`moveLeft`**`()`

Moves the game object one step to the left in the game field.

---

moveRight

`public void` **`moveRight`**`()`

Moves the game object one step to the right in the game field.

---

moveDown

`public void` **`moveDown`**`(int move)`

Moves down the game object.

Parameters:

`move` - how many steps to move down the game object

---

random

`public int` **`random`**`(int max)`

Returns a randomized value between 0 and max.

Parameters:

`max` - the maximum value to get out of the randomization

Returns:

the value that was produced

---

removeEyes

`public void` **`removeEyes`**`()`

Removes the eyes on the game object.

---

setColor

`public void` **`setColor`**`()`

Sets the color of the game object by using random.

---

setColor

`public void` **`setColor`**`(int c)`

Sets the color of the game object by using the input value.

Parameters:

`c` - the color to have on the game object

---

setType

```
public void setType(int type)
```

Set the type of the game object, either a Blob or a crystal ball.

Parameters:

`type` - the type that the game object should be. Use the constans defined in this class

---

setImage

```
public void setImage(int picture)
```

Set the image used to represent the game object.

Parameters:

`picture` - the image to represent the game object

---

randomEyes

```
public void randomEyes()
```

Randomizes where the eyes should be placed on the game objects body.

---

paintEyes

```
public void paintEyes(Graphics g,
                      boolean update_eyes)
```

Paints the eyes on the game objects body.

Parameters:

`g` - provides simple 2D geometric rendering capability

`update_eyes` - if the eyes should be repainted

---

paint

```
public void paint(Graphics g)
```

Paints the body of the game object.

Parameters:

`g` - provides simple 2D geometric rendering capability
