CHALMERS

Testdriven utveckling för mobila applikationer/Android

Test-driven development for mobile applications/Android

BURE RÅMAN VINNÅ

NICLAS MALMBERG

Examensarbete

Högskoleingenjörsprogrammet för datateknik

CHALMERS TEKNISKA HÖGSKOLA

Institutionen för data- och informationsteknik

Göteborg 2011

Abstract

Android is one of the leading software platforms for the new generation of smartphones. The market for applications is growing rapidly. So to be able to compete with such large competition the applications need a solid and stable development methodology. Therefore have we created an application based on a test-driven development, to investigate if the methodology was applicable on the Android platform. The conclusion is that Android's suitability for test-driven development is dependent on the application that is being developed and if difficulties concerning the testing framework can be avoided. We find it suitable for applications that are not being dependent on components of external applications. Although, the concerns regarding the test-framework is unavoidable without the use of suitable third party libraries.

Sammanfattning

Android-plattformen är tämligen ny, är föränderlig och antalet applikationer växer explosionsartat, vilket i sin tur ställer krav på val av utvecklingsmetodik för att kunna konkurrera framgångsrikt. Tanken med detta projekt är att utreda och applicera testdriven utveckling som metodik, på Android-plattformen, samt att utveckla en träningsapplikation, främst inriktad mot löpträning. Applikationen, är tänkt som ett komplement till en existerade websida och att marknadsföra densamma. Huruvida testdriven utveckling som metod passar bra för utveckling mot Android beror på applikationen som ska utvecklas samt om man kan kringgå problematik kring dess testramverk. Den passar bra för applikationer som i låg grad använder sig av komponenter från externa applikationer. Dock är lämpliga tredjepartsbibliotek nödvändiga för att kringgå ramverks-problematiken.

Innehåll

1. Inledning ... 6
1.1 Problembakgrund ... 6
1.2 Problem ... 6
1.3 Syfte ... 6
1.4 Avgränsningar ... 6
2. Metod ... 7
2.1 Kunskapsinhämtning ... 7
2.2 Utvärderings-kriterium ... 7
3. Teknisk bakgrund ... 7
3.1 Agila metoder ... 7
3.2 Testdriven utveckling ... 8
3.2.1 Enhetstest ... 9
3.2.2 Black box-test ... 10
3.2.3 Refaktorering ... 10
3.2.4 Mocknings-mönstret (Self-Shunt) ... 10
3.2.5 JUnit ... 11
3.3 Android ... 11
3.3.1 Historik ... 11
3.3.2 Arkitektur ... 11
3.3.3 Systemtjänster ... 12
3.3.4 Software Development Kit och plattformsstöd ... 12
3.3.5 Verktyg ... 12
3.3.6 Applikationen ... 13
3.3.6.1 Aktiviteter ... 13
3.3.6.2 Tjänst ... 15
3.3.7 Androids testramverk ... 15
4. Projekt JoggKompis/IronLog Runner ... 17
4.1 CodeKick ... 17
4.2 Projektmål ... 17
4.2.3 Projekt-metodik ... 18
4.3 Utvecklingsverktyg ... 18
4.4 Tredjeparts-bibliotek ... 20
4.5 Utvecklingen och uppbyggnad av JoggKompis/IronLog Runner ... 23

4.5.1 Tidsplan 23
4.5.2 Krav för hårdvaran 23
4.5.3 Behörighetskrav 23
4.5.4 Spårning 23
4.5.5 Statistik 24
4.5.6 Kommunikation 25
4.5.7 Övrigt 27
4.5.8 Design 28
4.5.9 Test av applikationen 29
5. Resultat 30
6. Diskussion och slutsats 30
6.1 Diskussion av resultat 30
6.2 Övrig diskussion 31
6.3 Slutsats 31
6.4 Kritisk diskussion 31
7. Referenser 33

1. Inledning

1.1 Problembakgrund

Marknaden för mobila applikationer, som på senare tid växt till en miljard-marknad, är ny och relativt outforskad. Då mobila enheter, med dess begränsade resurser, skiljer sig en del från dagens högpresterande stationära enheter ställs det höga krav på dess applikationer. Val av utvecklingsmetodik spelar en central roll inom olika typer av mjukvaruutveckling och det är inte självklart att en given utvecklingsmetodik passar sig för en viss typ av plattform.

Samtidigt har, på senare tid, en mängd mjukvaruutvecklingsmetoder, under samlingsnamnet agila metoder, fått ökad uppmärksamhet. Vi söker med hjälp av rapporten undersöka hur en del av dem, främst testdriven utveckling, lämpar sig för Android.

I rapporten kommer vi beskriva Android som plattform, ta upp begreppet testdriven utveckling dess fördelar och applikation på plattformen.

Teorin applicerades på en idé om att skapa en applikation som skall fungera som ett komplement till hemsidan Dumbbell.se. Detta för att marknadsföra tjänster som Dumbbell.se erbjuder.

1.2 Problem

Tillämpa testdriven utveckling på Android-plattformen för att avgöra dess tillämpbarhet på plattformen.

1.3 Syfte

Projektet har haft som syfte att undersöka och inhämta kunskap om testdriven utveckling som metod och tillhörande tekniker, verktyg och hjälpmedel, som en del i en agil utvecklingsmetodik. Dessutom, syftar projektet till att inhämta tillräckliga kunskaper om Android, för att kunna utveckla en applikation, med ovanstående metodik, på plattformen.

Efter avslutat arbete hoppas vi ha inhämtat tillräckligt med kunskap om både applikations-utveckling för Android, samt testdriven utveckling för att kunna arbeta vidare med det, på ett tryggt sätt.

1.4 Avgränsningar

Rapporten ämnar inte vara en uttömmande beskrivning av varken Android eller testdriven utveckling. Den syftar till att beskriva den kunskap vi skaffat oss, genom erfarenhet och text, under projektets gång tillräckliga för att bygga vår applikation. Kriterierna som använts är våra egna och ämnar endast beskriva vad vi anser vara nödvändigt för att bedriva testdriven utveckling på Android-plattformen.

2. Metod

För att vi ska kunna bedöma huruvida testdriven utveckling, som metod, passar för Android-plattformen behöver kunskap inhämtas, samt kriterier sättas upp.

2.1 Kunskapsinhämtning

Kunskap om Android har inhämtats genom litteratur och officiell dokumentation. Kunskap om testdriven utveckling samt agila metoder har inhämtats genom litteratur och kunskap från uppdragsgivare. Kunskap om ramverk, relaterade till både Android och testdriven utveckling har inhämtats genom dokumentation.

2.2 Utvärderings-kriterium

För att kunna genomföra vår utvärdering har följande kriterier använts. De har valts så enkla som möjligt för att utvärderingen med lätthet ska kunna falsifieras.

För att kunna bedriva testdriven utveckling på Android-plattformen anser vi att följande minimi-krav bör ställas:

- *Det ska vara möjligt att testa delar av en applikation i isolation från övriga delar, i form av enhetstest. Enhetstesten måste vara snabba att köra*
- *Det ska vara möjligt att testa en applikation i sin helhet. Testen ska vara automatiserade, lättlästa samt enkla att implementera.*

Utifrån dessa kriterier kommer ett resultat att motiveras.

3. Teknisk bakgrund

Under ovanstående rubrik ges en beskrivning av agila metoder, testdriven utveckling samt tillhörande tekniker och koncept. Dessutom ges en översiktlig beskrivning av Android, dess historik, arkitektur och komponenter, vilka varit centrala för vårt arbete.

3.1 Agila metoder

För att förstå logiken bakom testdriven utveckling framförs en kort sammanfattning av Agila metoder, i vilken testdriven utveckling är en viktig komponent(Martin & Martin, 2007, s4).

På senare år har en samling mjukvaruutvecklings-metodiker fått ökad uppmärksamhet. De namnges gemensamt som agil mjukvaruutveckling. Metodikerna säger sig vara reaktioner mot stora tungviktsprocesser, som anses vara för tungrodda och ej anpassningsbara i en föränderlig värld. Exempel på sådana är Extrem Programmering och Scrum. Under 2001 samlades ett antal förespråkare för dylika metoder under namnet The Agile Alliance för att diskutera och identifiera principer som de kan enas kring. Resultatet blev det Agila Manifestet, i vilket följande värden identifierades(Martin & Martin, 2007, s4):

- *Individer och interaktion framför processer och verktyg*
  Hävdar att människan och god kommunikation är den viktigaste beståndsdelen i ett utvecklingsarbete. Fokus bör därmed sättas på att skapa ett välfungerande arbetslag. Dessutom anses det vara angeläget att hålla sig till enkla kommunikationsverktyg och

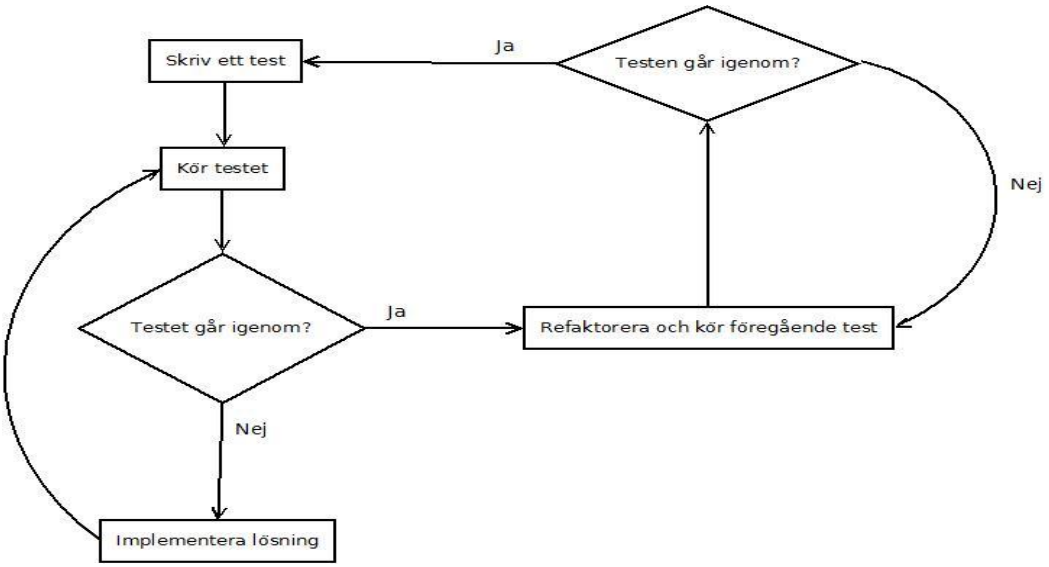
utvecklingsverktyg, om inte situationen kräver annorlunda, då mer avancerade verktyg tenderar att vara svårmanövrerade(Martin & Martin, 2007, s5).

- *Fungerande mjukvara framför omfattande dokumentation*
  Att producera dokumentation tar tid, både att skapa och att underhålla. Tanken är att låta koden självt vara sin egen dokumentation. När systemet och dess kod modifieras görs så även dokumentationen och en möjlig overhead elimineras. Detta ställer dock krav på god kod-kvalitet, vilket medför att extra arbete bör läggas ner på att hålla koden ren. Därmed inte sagt att all dokumentation är av ondo. Ibland kan tillägg behövas, men tanken är att den ska vara kort och endast beskriva strukturer i stora drag. Övrigt informationsutbyte anses idealt ske via interaktion mellan inblandade programmerare och kod. Martin ger följande riktlinje"Produce no document unless its need is immediate and significant."(Martin & Martin, 2007, s6).
- *Kundsamarbete framför kontraktsförhandling*
  Ovanstående värdering bygger på att mjukvara, till sin natur, anses vara föränderlig och leder till svårigheter om ett schema eller kontrakt följs slaviskt. Kontinuerlig feedback, nära samarbete med kund, samt inkrementella leveranser identifieras som nyckelkomponenter, då de ger större manövreringsutrymme för att kunna hantera förändringar i projektet(Martin & Martin, 2007, s6-7).
- *Anpassa sig efter förändring framför att hålla en fast plan*
  Ett projekt anses inte kunna planeras för långt in i framtiden, varför planeringens noggrannhet bör variera över tid. En grov rekommendation som ges är att lägga upp en detaljerad plan en vecka framåt, en grövre plan för de närmsta månaderna och så vidare(Martin & Martin, 2007, s7-8).

Ur dessa punkter identifierades tolv stycken principer, vilka är relevanta för alla agila metoder(Martin & Martin, 2007, s8-10).

3.2 Testdriven utveckling

Testdriven utveckling är en central del i agila metoder, men kan möjligtvis ses som en metodologi i sig självt. Den är starkt centrerad kring och refereras oftast som test-first development, vilket betyder att man börjar med att skriva test innan man börjar implementera produktions-kod. Proceduren kan beskrivas enligt följande: Identifiera ny funktionalitet, skriv och kör test. Om testet ej går igenom, implementera funktionalitet och få testet att gå igenom. Gör därefter en refaktorering, kör föregående test, om sådana existerar, samt försäkra sig om att de går igenom. Identifiera därefter ny funktionalitet och börja om. Förfarandet kan upprepas många gånger för den specifika del som testats och tanken är att det ska konvergera mot ett stabilt tillstånd, då man inte kan identifiera fler sätt att skriva ett felande test, i den del av det objekt som befinner under test. (Sommerville 2010, s222).



Figur 3.1 Flödesschema över test-proceduren

Martin och Martin(2007,s32)identifierar tre centrala punkter för testdriven utveckling

1. *Implementera inte en lösning innan ett felande test har skrivits.*
2. *Överarbeta inte testet. Skriv inte mer än nödvändigt.*
3. *Implementera inte mer av lösningen än nödvändigt för att få testet att gå igenom.*

Ovanstående punkter ska leda till att arbetet bedrivs i korta cykler samt att mängden produktions-kod och därmed mängden redundant kod begränsas. Om hela eller stora delar av det som utvecklas är associerade med ett eller flera test fås ytterligare fördelar i att tillägg och modifieringar i övriga delar av systemet kan göras med stor säkerhet om att inget nytt fel introduceras. Görs modifieringar i en del av programmet som har ett motsvarande test samtidigt som ett nytt fel genereras, blir programmeraren snabbt notifierad, med god noggrannhet, om vart felet uppstått. Följer man dessutom praktiken beskriven ovan så är risken liten att defekt kod släpps till kund. Testen kan även ses som en dokumentation över det som testas, exempelvis genom att visa vad som är ett förväntat resultat av ett funktionsanrop eller hur testobjektet ska användas och ett bra test eliminerar därmed stora delar av dokumentationsbördan. Dessutom uppmuntras deltagarna att upprätthålla god kod-kvalitet genom konstanta refaktoreringar.

3.2.1 Enhetstest

Följande definierar enhetstestning:

Definition: "*A unit test is a piece of a code (usually a method) that invokes another piece of code and checks the correctness of some assumptions afterward. If the assumptions turn out to be wrong, the unit test has failed. A "unit" is a method or function" (Osherove, 2010, p4).*

Enligt definitionen ovan kan många praktiker kvalificera sig som ett enhetstest och säger ingenting om hur den kan relateras till testdriven utveckling. Utöver definitionen bör det nämnas att ett enhetstest enbart ämnar testa en liten del av systemet i isolation.

Enligt Osherove(2010, s6) innehar ett bra enhetstest följande egenskaper:

- Automatiserat och återupprepningsbart

- Lätt att implementera
- Bör finnas kvar för framtida användning
- Vem som helst ska kunna köra testet
- Bör kunna köras med en knapptryckning
- Snabb exekverings-tid

Relationen till testdriven utveckling blir nu tydligare, då de listade egenskaperna adresserar punkter såsom återupprepningsbarhet, enkelhet och snabbhet.

3.2.2 Black box-test

Eftersom ett enhetstest bara testar en avgränsad del av en applikation behövs ett komplement för att säkerställa att applikationen fungerar som förväntat. Ett black box-test, eller acceptans-test, är ett sådant komplement och ämnar testa applikationen som en helhet, eller mindre delar av en helhet. Liknande egenskaper som det som kännetecknar ett bra enhetstest är önskvärda när det används som en del inom testdriven utveckling. Ett black box-test bör därför skrivas med en test-first approach, vara en form av dokumentation samt vara repeterbar. Idealt är ett black box-test skrivet av kund och härstammar från användarfalls-beskrivningar(Martin & Martin, 2007, s36-37).

3.2.3 Refaktorering

För att snabbt kunna anpassa sig till förändring krävs god kod-kvalitet. Refaktorering är ett välkänt begrepp inom objektorienterad design och är centralt inom agila metoder, då de ställer höga krav på kodens läsbarhet. Refaktorering går ut på att man kontinuerligt gör små förändringar i koden för att göra den mer hanterbar och läsbar, utan att programmet i övrigt påverkas. Det kan innebära allt från att bryta ut metoder och klasser, för att avgränsa ansvarsområden, till att ändra namngivning på metoder och variabler. Förfarandet blir kraftfullt i samband med testdriven utveckling, då man, genom verifikation av existerande test, i hög grad kan försäkra sig om att modifieringar i kod inte introducerar nya buggar, vilket i sin tur uppmuntrar programmeraren att upprätthålla en god kod-kvalitet(Martin & Martin, 2007, s41-54).

3.2.4 Mocknings-mönstret (Self-Shunt)

För att framgångsrikt kunna bedriva testdriven utveckling behöver man, i vissa fall, koppla bort objektet under test från systemet i övrigt. Till exempel, man har ett objekt A som kommunicerar med en databas, webtjänst, eller motsvarande och vill testa om A interagerar med det yttre objektet på ett önskvärt sätt, eller A:s beteende i övrigt. Det går naturligtvis att åstadkomma genom att, på ett antal sätt, observera det yttre objektet. Problem uppstår om man inte vill modifiera det, exempelvis en databas, med testvärden. Möjligtvis kanske den inte ens existerar. Dessutom borde inte A vara beroende av implementationen av det yttre objektet. En möjlighet blir då att låta den utomstående enheten representeras av ett gränssnitt som har två implementationer. En som enbart är till för testning samt en som används i produktionskoden. Med hjälp av polymorfism avgörs vilken av implementationerna som A använder. Exempelvis kan sättande av beroenden då göras genom att skicka med en instans av det gemensamma gränssnittet till A:s konstruktor eller via set-metoder. Förutom att simulera enklare tjänster kan även test-implementationen implementeras på ett sådant sätt att testning i större utsträckning bedrivas mot mindre vanliga händelser. Exempelvis, givet en viss input kasta ett visst undantag, för att testa att objektet under test hanterar undantaget på ett korrekt sätt(Martin & Martin, 2007, s443-444).

3.2.5 JUnit

JUnit är ett ramverk för automatiserad enhetstestning, ämnad för Java. Test skrivs som en mängd testfall, vilka är organiserade som metoder, annoterade med @Test, i en testklass. I versioner äldre än JUnit4 måste testklasserna ärva från bas-klassen TestCase. Testning av ett objekt utförs genom att man instansierar objektet i en sådan metod, anropar lämplig metod och undersöker därefter dess tillstånd med hjälp att assert-metoder, vilka jämför förväntat med faktiskt resultat och avgör om testfallet går igenom eller ej. En stor fördel är att man kan köra enstaka eller alla metoder i en testklass, samt att det går väldigt snabbt. Möjlighet till statistik finns såsom hur många test som gått igenom eller felat ges eller vart och varför testet felat. Dessutom finns möjlighet att sätta upp förutsättningar innan test samt att rensa tillstånd efter test för att på så sätt enkelt och tydligt kunna definiera vad som testas(”JUnit”).

3.3 Android

I takt med att mobiltelefoner har utvecklats har så också dess operativsystem och plattform, på vilken mjukvara skrivs. Ett av de, på senare tid, mest populära är Android. Vi kommer här att beskriva Android-plattformen i sig, en Android-applikations fundament samt ett par viktiga komponenter som ingår i en Android-applikation. Dessutom kommer vi att kort redogöra för det stöd Android erbjuder för testning.

3.3.1 Historik

Android är en mjukvaru-plattform, operativsystem inkluderad, licensierad under Apache Software Licence (kärnan exkluderad vilken är GPL), och är ämnad att användas på mobila enheter. Utvecklingsarbetet av Android-plattformen påbörjades 2005 av Google, efter uppköp av Android Inc. Under 2007 bildade en grupp företag the Open Handset Alliance, vars mål är att vidareutveckla Android. Därmed är plattformen inte knuten till ett visst telefonmärke eller leverantör, utan återfinns på flertalet enheter, med varierande hårdvara och prestanda(Ableson, Sén, & King, 2011, s4, s9)(Hashimi, Komatineni, & Maclean, 2010, p3).

3.3.2 Arkitektur

Arkitekturen hos Android delas in i ett antal lager. Det lägsta inkluderar hårdvaran, såsom GPS och kamera och tillhörande kärna. Kärnan som används är en Linux-kärna och utgör en abstraktion gentemot hårdvaran, samt erbjuder nödvändiga OS-tjänster.

På nästa nivå återfinns ett antal C/C++-bibliotek, exempelvis OpenGl för 2d och 3d-grafik, SQLite för databas-funktionalitet och media-bibliotek med stöd för många av de vanligaste video, ljud och bild-formaten. Bibliotekens funktionalitet går att nås via högre lager. På samma lager återfinns dessutom Android runtime, vilket inkluderar delar av Javas standard-API samt Dalvik VM(Virtual Machine). Dalvik VM är en variant på Java VM, anpassad för att lösa problemet med att mobila enheter ofta har ett begränsat minnesutrymme. I korta drag kringgås delar av problemet genom att Dalvik exkluderar redundant information i genererade klassfiler och kombinerar dem i ett eget filformat(.dex).

Nästa lager är applikations-ramverket. Här erbjuds utvecklaren att ta del av de system-tjänster nedanstående lager erbjuder, genom Android runtime. Därmed kan Android erbjuda både plattform-specifika tjänster, samt de flesta från Javas standardbibliotek. Dock, existerar skillnader dem emellan. Exempelvis inkluderas ej delar av Javas grafikbibliotek(Swing) i Android-plattformen, då Androids UI-ramverk skiljer sig från Javas.

Överst återfinns applikations-lagret. De standard-applikationer som medföljer en telefon är alla utvecklade genom Androids Java-API(Hashimi et al., 2010, s10-12)(”What is Android” 2011).

3.3.3 Systemtjänster

Applikations-utvecklaren har möjlighet att ta till vara på alla de systemtjänster Android erbjuder. För vårt arbete viktiga sådana har varit plats-tjänsten och möjlighet till datalagring i en SQLite-databas.

Information om användarens plats kan fås via ett antal leverantörer. Exempelvis kan den tas emot via WiFi och GPS, där GPS ger mest tillförlitlig data men är minst pålitlig. Informationen erbjuds i form av en klass namngiven Location, vilken erbjuder metoder för att få fram data angående riktning, fart, position, träffsäkerhet och höjd, samt om leverantören har haft möjlighet att bifoga specifik data(”Location” 2011)(”Obtaining User Location” 2011).

Varje applikation har möjlighet att skapa en SQLite-databas. SQLite är en relations-databas, är mindre än vanliga relationsdatabaser samt har stöd för större delen av SQL. Den skapade databasen blir associerad med det paket applikationen tillhör och därmed, som standard, privat för applikationen. Android tillhandahåller klasser för att skapa, öppna och uppgradera databasen. Vanligt stöd för att arbeta mot databasen erbjuds i form av möjlighet att utföra SQLite-queries och hantering av Cursors.(Ableson et al., 2011, s144-149)(”About SQLite” n.d.).

3.3.4 Software Development Kit och plattformsstöd

Android som plattform uppdateras ofta, vilket medför att det finns många olika versioner av tillhörande Software Development Kit(SDK) för att utveckla applikationer. För att garantera bakåt-kompabilitet med lämpliga plattformsversioner finns möjlighet att ange den minsta version av SDK applikationen använder. Utöver versioner som stödjer de olika versionerna av plattformen finns också motsvarande API från Google för att utnyttja Google Maps-relaterade tjänster, som inte är plattformsstandard(Ableson et al. 2011, s54) (”Backward Compatibility for Applications” 2011).

3.3.5 Verktyg

För att underlätta utvecklingsarbetet medföljer nedladdad SDK ett antal verktyg. Exempel på sådana är Android Emulator och Dalvik Debug Monitor Server.

Emulatorns syfte är att möjliggöra körning och testning av program, i en miljö likt den på en faktisk enhet, på utvecklarens dator. För att uppnå det, krävs att man skapar och konfigurerar en eller flera Android Virtual Device(AVD). En AVD är en emulator-konfiguration använd till att möjliggöra jämförelser av hur applikationen ter sig på olika typer av enheter. Exempel på egenskaper en AVD är kustomiserbar på är storlek på RAM-minne, skärmstorlek, upplösning samt plattformsversion, vilket ger en bra möjlighet att testa hur applikationen beter sig på olika enheter(”Managing Virtual Devices” 2011).

Dalvik Debug Monitor Server(DDMS) är ett verktyg för debuggning och innehåller kapabiliteter för att, bland annat, logga händelser och tillstånd i emulatorn samt skicka punkter, relaterade till plats, till densamma. Punktens position kan ställas in och skickas manuellt, eller så kan en mängd punkter, med valfri data, läsas från fil, och skickas i ordning för att möjliggöra verklighetstrogen simulering av en användare i rörelse(”Using DDMS” 2011).

3.3.6 Applikationen

Ett färdigbyggt program levereras som en .apk, vilken består av en eller flera .dex-filer(jämför med .jar och .class-filer), samt applikations-resurser, exempelvis bilder. En applikation körs i en egen Linux-process med en egen instans av Dalvik VM och eget unikt ID. ID:et bestäms av det Java paket en applikation tillhör och måste definieras. Detta möjliggör att applikationer kan köras helt i isolation från varandra, endast med behörighet till resurser medföljande dess .apk.

För att ställa högre krav på säkerhet har Android som princip att en applikation inte får tillgång till vissa systemtjänster om den inte har behörighet. Behörighet godkänns av användare och förfrågan görs vid installation. Ovanstående information är nödvändig för att applikationen ska kunna köras. Motsvarande information och mer därtill anges i en specifik fil kallad AndroidManifest.xml som därmed kan ses som en samling nödvändig information om applikationen (”Application Fundamentals” 2011).

En Android-applikation innehåller dessutom ett antal komponenter, varav ett par av dem kommer att redogöras för nedan.

3.3.6.1 Aktiviteter

En aktivitet är något användaren interagerar med och utgör därmed oftast utgångspunkten för en applikation. Ett fönster skapas automatiskt vid start av en aktivitet, i vilket det finns möjlighet att definiera ett grafiskt användargränssnitt, och kan ta upp hela eller delar av skärmen. Det grafiska gränssnittet kan definieras i xml-filer, som en del av en applikations resurser och läses in när applikationen startar, eller skapas dynamiskt, på liknande sätt som görs i standard-Java.

En aktivitet är olikt ett vanligt Java-objekt, på så sätt att man inte instansierar det på konventionellt sätt utan startas asynkront från andra aktiviteter med hjälp av ett så kallat Intent(jämför med swing-komponenter i Javas standardbibliotek) (”Application Fundamentals” 2011).

Ett Intent är en beskrivning av något som skall utföras och används bland annat för att starta aktiviteter. En aktivitet startas antingen genom att uttryckligen referera till aktivitetens klass eller genom att beskriva något man vill få utfört, i det Intent som används. En aktivitet kan i tillhörande applikations manifest-fil berätta vad den kan utföra, för att på så sätt möjliggöra kommunikation applikationer emellan och användning av andra applikationers aktiviteter. Beskrivningen av vad som ska utföras kan ske på ett antal olika sätt och kan exempelvis vara att visa en viss typ av information, men vi ämnar inte gå in djupare på dem (”Intent” 2011).

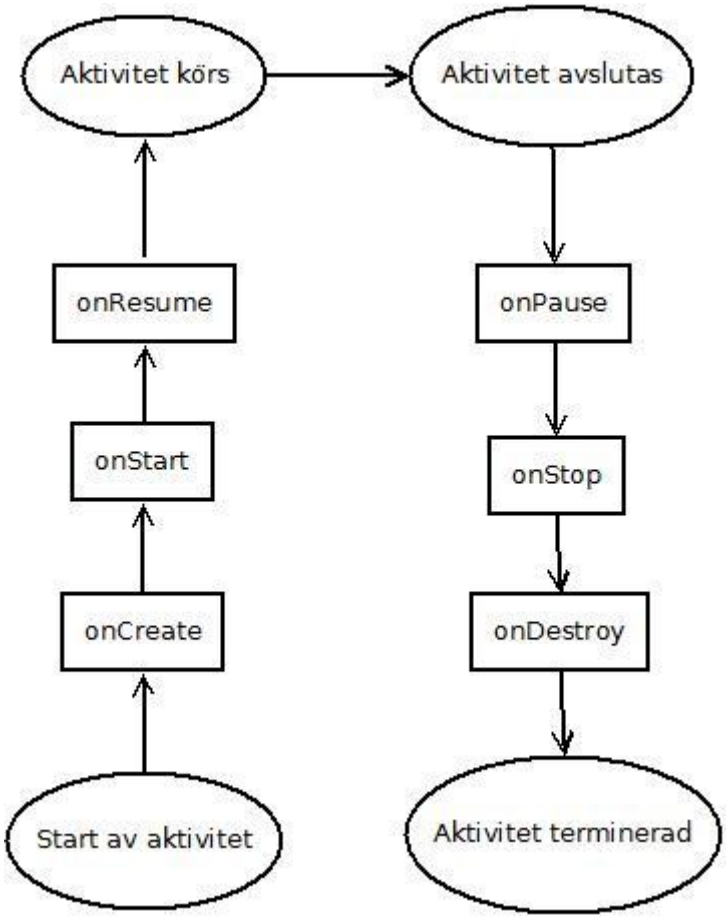
En aktivitet kan befinna sig i följande tillstånd:

- *Aktiv*
  Aktiviteten är synlig och har fokus. Möjligt för användaren att använda UI-komponenter.
- *Pauserad*
  Aktiviteten är synlig men innehar ej fokus. Användaren kan ej använda UI-komponenter.
- *Stoppad*
  Aktiviteten är ej synlig.

För att optimera minnesanvändandet förbehåller Android sig rätten att terminera, utan notifikation, hela eller delar av, en applikation enligt följande generella regler: Synliga, både aktiva och icke-aktiva

aktiviteter har högst prioritet. Därefter följer aktiva tjänster samt processer som har inaktiva aktiviteter(Ableson et al., 2011, p25).

Då hela eller delar av en applikation kan komma att terminernas av systemet eller mista fokus på grund av val användaren gör, kan komplikationer uppstå. Därför är en aktivitet utrustad med ett antal livscykelmetoder relaterade till tillstånden ovan. Livscykelmetoderna är till för att göra de ändringar man anser nödvändiga att göra när aktiviteten byter tillstånd. Exempelvis kan man vilja göra modifieringar i användargränssnittet när aktiviteten återvänder till aktiv, från ett pauserat eller stoppat tillstånd, eller spara data i en databas, när den går från ett aktivt till stoppat tillstånd. Därmed kan det vara viktigt att testa hur applikationen beter sig när livscykelmetoderna anropas.



Figur 3.2:Beskrivning av hur livscykelmetoderna anropas i sekvens från start till slut

Dock, kan en aktivitet komma att termineras när som helst, utan några garantier på att kunna exekvera mer kod, om den befinner sig i ett stoppat tillstånd eller lägre. (”Activity” 2011).

Då en applikation kan innehålla multipla aktiviteter administreras dessa med hjälp av en aktivitetsstack, där den aktivitet som ligger överst är aktiv. Samlingen aktiviteter kallas för Task och kommer till förgrunden när användaren trycker på programikonen. Aktivitetsstacken manipuleras direkt av BACK-knappen och aktivitetssamlingen med HOME. Ett antal mer eller mindre avancerade startinställningar kan ställas in för en aktivitet, bland annat hur den ska startas i förhållande till redan existerande aktiviteter av samma typ, eller till redan existerande aktivitetssamlingar(Tasks and Back Stack 2011).

3.3.6.2 Tjänst

Som beskrivits ovan, kan aktiviteter befinna sig i lägen där de är starka kandidater att bli terminerade, samt har som främsta funktion att förmedla ett gränssnitt mot användaren. Därför erbjuder Android en komponent kallad Tjänst(Service). En tjänst är fri från grafiska användargränssnitt, är tänkt att användas för arbeten som ska pågå under lång tid, samt tillhandahålla tjänster till den egna applikationen eller utomstående applikationer. En tjänst körs, om inte annat anges, i applikationens huvud-tråd, och är därmed inte en egen process. Den är snarare en samlingsplats för saker som behöver göra långvarigt bakgrundsarbete, så tidskrävande arbeten för en tjänst som körs i samma process som applikationen bör göras i separata trådar, för att inte blockera applikationen, skapade och startade i en tjänst. På grund av dess funktion har en tjänst inte likadana livscykel-metoder som en aktivitet, endast två gemensamma, onCreate och onDestroy, de andra är relaterade till hur tjänsten startas.

En tjänst kan startas från aktiviteter, eller en annan tjänst, med hjälp av ett Intent, samt anrop till metoderna startService eller bindService. En tjänst startad med startService körs så länge inte stopService anropas från andra komponenter, eller att den avslutar sig självt. Anrop till startService ger inget sätt för andra komponenter att kommunicera med tjänsten och producerar inget returvärde när tjänsten är klar med sitt arbete och stängts ner. Andra komponenter kan därför binda till tjänst genom metoden bindService för att skapa en klient-server-liknande kommunikations-struktur. Flera komponenter kan vara bundna samtidigt till en tjänst och tjänsten överlever så länge den har bundna klienter. En bindning möjliggör kommunikation både inom och mellan processer. Både startService och bindService kan kombineras för att skapa en självständigt, långvarigtjänst med möjlighet att kommunicera med multipla klienter (”Services” 2011).

Det finns ett par olika sätt för kommunikationen att äga rum, ett av dem är med Handler och Messenger-objekt. En Handler är ett objekt som är associerad med en meddelande-kö, varje tråd har en meddelande-kö så en Handler är alltid associerad med den tråd som skapat den, och används för att leverera körbara-objekt och meddelanden till sin kö. En Messenger är en referens till en Handler och används därmed för att möjliggöra kommunikation mellan klienter och en tjänst, och kan skapas då en bindning mellan tjänst och klient har skett (”Messenger” 2011) (”Handler” 2011).

3.3.7 Androids testramverk

Android erbjuder ett integrerat testramverk, avsedd att möjliggöra testning på olika delar av en Android-applikation. Testen organiseras som ett fristående projekt med egen paketstruktur, manifest-fil och filer med definierade test-fall. Test-projektets manifestfil liknar den för en vanlig applikation, med avseende på att man anger vilka komponenter som ska testas, paketnamn och SDK-version. Då test-projektet måste länkas samman med det man vill testa måste man dessutom ange målpaket samt definiera en test-körare. Innan körning av test laddas testprojektet in i emulatorn, varpå en ny instans av mål-applikationen skapas. Därefter körs definierade testfall mot komponenten som ska testas.

Android-komponenter körs och skapas normalt av operativsystemet, vilket ger svårigheter att bedriva testning mot sådana objekt. Android tillhandahåller därför funktionalitet för att kringgå den begränsningen i form av ett instrumentationsobjekt. Instrumentationen möjliggör, som exempel, kontroll av en aktivitet utanför dess livscykel, genom att ”hooka” in i systemet. På så sätt kan man via instrumentationen styra komponenten, samt skicka händelser, på ett sätt som normalt sköts av

systemet. Android erbjuder även stöd för mockning av vissa systemklasser, vilket ger ytterligare stöd för att bedriva meningsfull testning.

Beroende på vad man vill testa så finns det ett antal olika bas-klasser tillhandahållna. De används för att testa Android-komponenter eller objekt som är beroende av Android. Test av Android-komponenter kan användas för att testa dem i isolation eller som del av applikationen. Bas-klasserna är alla baserade på JUnit och erbjuder därför liknande funktioner. Exempel på liknande funktioner är skapade av test-förutsättningar, upprensning efter test samt organisation av testfall. Vi begränsar oss till att beskriva en handfull av dem. (”Testing Fundamentals” 2011).

- *AndroidTestCase*
  Används för att testa klasser som, på ett eller annat sätt är beroende av Android. Exempelvis, en test-svit vilken ämnar testa en klass som interagerar med en databas bör ärva från AndroidTestCase.
- *ActivityInstrumentationTestCase2*
  Förmodligen den viktigaste testklassen. Används för att testa en aktivitet under en normal instans av applikationen. Det vill säga, aktiviteten som ska testas körs och interagerar med systemet på samma sätt som den skulle gjorts om applikationen och aktiviteten körs normalt. Dessutom finns möjligheter att, med hjälp av instrumentation, manipulera aktiviteten genom att anropa dess livscykel-metoder och sända händelser för att interagera med användar-gränssnittet.
- *ServiceTestCase*
  Ämnad till att användas för att testa en tjänst. Test-sviten antar att man vill köra testet i isolation och skapar automatiskt mock-objekt av nödvändiga applikations-klasser. Vill man så kan man tillhandahålla sina egna implementationer av mock-klasserna. Möjlighet finns att starta och binda till en Service på samma sätt som från en aktivitet.

4. Projekt JoggKompis/IronLog Runner

Under examensarbetet utvecklades en applikation till företaget CodeKick. I det här kapitlet kommer vi beskriva utvecklingen av applikationen. Viktiga verktyg och hjälpmedel i form av tredjeparts bibliotek och ramverk kommer att sammanfattas. Medföljer gör även en beskrivning om hur applikationen fungerar, samt även viktiga delmoment och tankar som uppkommit under utvecklingen av applikationen.

4.1 CodeKick

CodeKick Solution(http://codekick.com), nedan kallat CodeKick, är ett IT-konsultföretag med kontor i Göteborg. Företaget verkar inom systemutveckling i .NET riktat mot webben. Vi kom i kontakt med CodeKick via Nationella Exjobbs-poolen(http://www.xjobb.nu). Där vi valde att lämna vårt intresse för deras idé om ett projekt riktat mot den mobila marknaden. Huvudsyftet med projektet är att tillsammans med CodeKick utreda, samt eventuellt implementera en applikation till antingen iPhone OS (iOS) eller Android.

4.1.1 Dumbbell.se

Applikationen ska främst fungera som ett träningskomplement till websidan Dumbbell.se samt dess syster-sida IronLog.com, vilken kom att lanseras under projektets gång. Dumbbell.se riktar in sig mot svenskspråkiga användare och IronLog.com mot engelskspråkiga. Gemensamt för webbsidorna är att de fungerar som ett socialt medium för träningsutövare, där man kan publicera, antingen privat eller publikt, en träningsdagbok samt kostdagbok och dessutom diskutera ämnen relaterade till kost och träning. Applikationen är tänkt att fungera vid utomhusaktiviteter där utövaren fokuserar på konditions-träning så som löpning, cykling eller promenad. Applikationen skall även efter avslutat pass ha möjlighet att synkronisera träningspass med Dumbbell.se.

4.1.2 Val av plattform

Tillsammans med CodeKick valde vi att förverkliga idén på Android. Vilket vi tyckte var lämpligt, då våra tidigare erfarenheter av Java samt ytlig kännedom om Android SDK möjliggjorde att vi snabbt kunde påbörja utvecklingsfasen. Dessutom anser vi att Android SDK är mer lättillgängligt än iOS då det inte krävs någon licens och utvecklings-verktygen är öppna och väl-dokumenterade.

4.1.3 Lokaler

Då CodeKick saknade plats på deras huvudkontor fick vi kontorsplats hos Business Intelligence Partner (BI Partner), Kyrkogatan 20-22, Göteborg. Förutom själva arbetsplatsen fick vi tillgång till hårdvara i form av skärmar, möss.

4.2 Projektmål

Nedan listas mål för projektet, både från vår samt CodeKicks synvinkel.

4.2.1 Vårt mål

Vårt mål med utvecklingen av applikationerna sammanfaller med rapportens syfte, samt att så långt som möjligt tillgodose CodeKicks krav på applikationen.

4.2.1 Projektmål CodeKick

Utveckling av ett träningskomplement till Dumbbell.se samt IronLog.com, i form av en Android applikation. Applikationen skall främst vara ett komplement vid löpning, cykling och promenad. Applikationen skall även fungera som marknadsföring för dagböckerna, det vill säga hög prioritering har lagts på möjligheten till synkronisering med, samt möjligheter för nya användare att registrera sig på, webbsidorna.

Koden och projektets uppbyggnad skall utvecklas så att CodeKick som förvaltningsorganisation efter avslutat projekt skall kunna ta över koden och på ett lätt och tillförlitligt sätt fortsätta arbetet. Detta innefattar dokumentation i form av test, samt att fokus läggs på god kod-kvalitet.

4.2.3 Projekt-metodik

Utvecklingen av applikationen skedde med ett agilt förhållningssätt i allmänhet och ett testdrivet i synnerhet. Följande punkter har, både för applikationen i sig samt projektet som helhet, varit riktlinjer under projektets gång.

Täta avstämningar: Då det oftast är svårt att veta i detalj från början hur ett system bör fungera har fokus lagts på att ha täta avstämningar där vi gått igenom hur utvecklingen fortlöpt samt identifierat problem och diskuterat prioriteringar. Detta för att både vi och uppdragsgivaren ska ha samma bild av hur utvecklingen fortgått samt att det blir mycket enklare att göra justeringar och omprioriteringar än om alla synpunkter kommer i slutet av projektet. Vi försökte att en gång i veckan ha ett avstämningsmöte där kod-granskning bedrevs samt problem och idéer diskuterades.

Användarvänlighet: Applikationen skall sätta användaren i fokus. Det vill säga applikationen skall vara självinstruerande. Man bör slippa att läsa någon manual för att kunna använda applikationen. Dessutom skall applikationen upplevas som snabb, utan frustrerande väntetider. Applikationen skall även ha ett enkelt och tilltalande gränssnitt.

Kod-kvalitet: Då koden som vi producerats skall tas över av förvaltningsorganisationen kräver det att koden är enkel och självförklarande. Fokus har lagts på kod-granskning och refaktorering.

Tester: Fokus ska läggas på att testa både komponenter som isolerade delar i applikationen, samt delar av applikationen som helhet. Testerna ska vara enkla och självförklarande.

4.3 Utvecklingsverktyg

Nedan listas de utvecklingsverktyg vi använt under projektets gång. Alla har varit viktiga och underlättat vårt arbete.

4.3.1 IntelliJ IDEA

Den integrerade utvecklingsmiljön för projektet blev IntelliJ Community Edition, utvecklat av företaget JetBrains. Den första versionen av IntelliJ lanserades i januari 2001, och var då marknadens enda utvecklingsmiljö Java- med avancerad rekonstruering samt kod-navigeringsverktyg integrerat. IntelliJ inkluderar bland annat UML- liknande klassdiagram, beroendeförhållande, analys av dataflöden samt stöd för Apache Maven, Spring 3.0.

Vi valde att använda gratisversionen IntelliJ Community Edition som har ett bra stöd för Android-utveckling. De specifika hjälpmedel IntelliJ Community Edition erbjuder för Android är:

- *Android SDK och AVD Manager*
  IntelliJ Community Edition är integrerat med *Android SDK and AVD Manager*, vilket ger stöd för samtliga versioner av *Android SDK* och möjligheten till att skapa och konfigurera virtuella enheter(AVD). Utöver integreringen kan IntelliJ automatiskt skapa olika typer av projekt, så som *Android Applications*, *Android Libraries,* moduler för test, samt automatiskt genererad kod till aktiviteter, xml filer med mera.
- *Android Logcat*
  Gör det möjligt att i realtid se alla logg-meddelanden och undantag som produceras under debuggningen av en applikation. Logcat har även support för spår-navigering och messa.
- *Android-testning*
  IntelliJ har möjligheten att skapa moduler använda till testning av Android projekt. Dessa test kan sedan köras och utvärderas med hjälp av det inbyggda gränssnittet för testning.
- *Export och import*
  IntelliJ har möjligheten att exportera och importera olika Android projekt, i form av specifika IntelliJ projekt (.iml) eller signerade Android applikationer (.apk). Samt konvertera andra projekt skapade i liknande utvecklingsmiljöer.
  (se http://www.jetbrains.com/idea/features/google_android.html för mer info).

Förutom fördelarna med det breda stödet för Android använder även CodeKick IntelliJ som deras utvecklingsmiljö. Vilket gör synkroniseringen mellan utvecklare och uppdragsgivare lättare, då alla har liknande projektstrukturer och således underlättar förbindningen av samma projekt hos alla parter.

4.3.2 Git

Vi använde Git tillsammans med GitHub som versionshanterings-system. Git är ett distribuerat versionshanterings-system designat för snabbhet och effektivitet. Detta för att snabbt och enkelt kunna dela koden mellan utvecklarna och uppdragsgivaren men även ge möjlighet att arbeta med samma projekt i olika versioner (branches/repositories). En gren kan hålla olika versioner av projektet och uppdateringar mot en gren påverkar inte andra. Byter man gren sker uppdatering av projekt-filer automatiskt, lokalt på utvecklarens dator. Förvaring av grenarna startades på GitHub, vilket senare klonades till våra datorer, vilket innebar att vi kunde göra ändringar i en gren lokalt parallellt med varandra, för att sedan lägga upp dem på GitHub. Konflikter för ändringar kan komma att uppstå, men Git har bra verktyg för att hantera sådana.

Vi använde främst två separerade grenar, develop och master. Develop användes för att testa och utveckla projektet och master för att spara en stabil version av projektet. Därmed blev master en utgångspunkt för releaser. När projektet i develop är mogen för release sammanfogas den med master. Den smidiga hanteringen av grenar underlättade även skapandet av skarpa buggfixar, då en ny gren kan skapas utifrån master i vilken man gör nödvändiga ändringar för att sedan sammanfoga med master igen.

Vidare användes Wiki-tjänsten på Github för att publicera protokoll från våra avstämningsmöten med CodeKick, samt nödvändig information om projektet i övrigt.

4.3.3 ReSharper

ReSharper är ett refaktorerings-verktyg framtaget av JetBrains. Då refaktorering är ett centralt begrepp inom testdriven utveckling, krävs det ett kraftfullt verktyg för att underhålla och skapa en god kod-kvalitet. ReSharper utför statisk kod-analys, vilket används för felkorrigering, automatisk kod-avslutning, projekt-navigering, formatering samt kod-optimering.

4.3.4 Kanbantool

För att kunna anpassa oss till förändringar och skapa en detaljerad planering för den närmsta tiden användes Kanbantool. Kanbantool är ett verktyg för att visualisera arbetsflöden. Via Kanban board kan man skapa diverse ”sticky notes” (virtuella post-it lappar) för att organisera arbetet. För att prioritera visa uppgifter, eller veta exakt vilka moment som skall genomföras innan nästa skarpa release. Under veckomöten med CodeKick planerade vi delmål och visioner, som vi sedan bröt ner i mindre mer specifika beståndsdelar. Dessa mindre moment skulle motsvarade max fem timmars arbete och vi kunde på det sättet hålla koll på vad kollegan gjorde, samt snabbt ändra planeringen.

4.3.5 Android SDK 2.1Eclair

Krav på applikationen var att den skulle fungerar på så många olika typer av telefoner som möjligt, utan att försämra funktionaliteten. Vidare krävdes bra stöd för olika skärmstorlekar, det vill säga skärmar primärt anpassade för telefoner (låg, mellan och stor upplösning), vilket lanserades i Android 1.6. Då det flesta av producenterna utav Android-baserade telefoner valt att lansera eller var på väg att lansera uppdateringar för sina modeller till Eclair eller senare blev således valet Android SDK 2.1 Eclair, API -version 7. Eclair är en mindre plattforms-release som lanserades i januari 2010. Plattformen inkluderar diverse buggfixar samt även en del nya API-ändringar. SDK:n kompletterades med versionen innehållandes Googles API.

4.4 Tredjeparts-bibliotek

Applikationen samt projekten använder sig av diverse tredjeparts-bibliotek för att underlätta utvecklingen, samt testningen av applikationen. Nedan följer korta beskrivningar av de olika bibliotek vi valt att använda i projektet, samt lägre beskrivningar för rapporten mer viktiga.

- *AChartEngine*
  Används i applikationen för att presentera tempo, höjdskillnad samt hastighets-grafer. AChartEngine är ett open source-ramverk som distribueras under Apache 2.0 license. Vi valde att göra diverse små ändringar i källkoden för att tillmötesgå våra krav.
- *JUnit*
  Används för att skapa enhetstester för klasser som inte är beroende av Android SDK. Detta gör testerna snabbare då det inte är beroende av en lång laddningstid samt eventuellt emulatorn.
- *JSON*
  JSON (JavaScript Object Notation) är ett kompakt textbaserat format för att utbyta data. Används internt i applikationen men främst för synkronisering mellan applikationen och webtjänsterna.
- *GSON*
  GSON är ett Java-bibliotek utvecklat för att konvertera Java-objekt till JSON-strängar,

inklusive JSON-strängar till motsvarande Java-objekt. Används vid all JSON-konvertering i applikationen

- *AdMob*
  AdMob SDK för Android används för att implementera annonsering i applikationen.

4.4.1 Robotium

Används för att underlätta instrumentations-testning. Robotium låter dig skriva automatiserade UI tester som automatiskt klickar runt i en del av applikationen, ej frånkopplad från övriga delar och kan därmed användas för black box-testning. Det vill säga test som beter sig som en användare. Robotium bygger på ActivityInstrumentationTestCase2 och dess instrumentation.

Fördelarna med Robotium är: Ett Robotium test är väldigt kod-effektivt, vilket leder till att koden blir lättläst. Då ett test inte kräver samma mängd kod går det snabbare att skapa aktivitets-tester. Kodens struktur leder även till att en tredje person lättare kan förstå sig på testet, utan krav på Android kunskaper. Detta är väldigt viktigt då koden för testen i sig skall vara dokumentationen för applikationen. Ytterligare fördel är automatisk synkronisering med testkoden och applikationen som testas. Exempelvis kan man med enkla metod-anrop instruera testet att vänta på att en händelse.

Test med Robotium

```
@Smoke
public void test_login_withNotValidPassword_errorToastShows() {
    EditText userName = (EditText)_activity.findViewById(R.id.usernameInput);
    EditText password = (EditText)_activity.findViewById(R.id.passwordInput);
    _solo.enterText(userName, "a");
    _solo.enterText(password, "a");

    _solo.clickOnButton("Logga in");

    assertTrue(_solo.searchText("Fel användarnamn eller lösenord, var god försök igen."));
}
```

Fig 4.1 Ett Robotium-test.

4.4.2 RoboGuice

Bygger på Googles Guice med ytterligare tillägg för Android. Guice används enklast för att injicera beroenden i klasser(Dependency Injection). För att beskriva det i korta drag går det ut på att Guice tar hand om att skapa och förmedla objekt som är satta som beroenden hos en klass. Som ett exempel, kan man markera att en klass är beroende av ett antal andra klasser genom att annoterar dess konstruktor, vilken tar instanser av klasserna den är beroende av som parametrar till sin konstruktor, med @Inject. Man kan även binda ett interface A till klasser som implementerar A. Det görs i speciella klasser och man kan bestämma vilka bindningar som ska gälla. Det medför att man kan använda sig av mocknings-mönstret på ett smidigt sätt, då man kan sätta vilka bindningar som ska gälla i testklasserna, använda andra i produktionskoden och på så sätt slippa besväret med att sätta beroenden manuellt i testklasserna. RoboGuice erbjuder ytterligare tjänster för Android i form av injektion av vyer och resurser, vilket gör koden mer lättläslig.

Normalt utförande i Android

```
class AndroidWay extends Activity{
   TextView _sampleTextView;
   Drawable _sampleDrawable;
   LocationManager _sampleLocationManager;
   String _sampleString;

   public void onCreate(Bundle savedInstanceState){
      super.onCreate(saveInstanceState);
      setContentView(R.layout.main);
      _textView = (TextView)findViewById(R.id.sampleTextView);
      _sampleDrawable = this.getResources().getDrawable(R.drawable.sampleDrawable);
      _sampleLocationManager = (LocationManager)getSystemService(Activity.LOCATION_SERVICE);
      _sampleString = this.getString(R.string.sampleString);

      _textView.setText("Skriv ut sampleString: "+_sampleString);
   }
}
```

Fig 4.2 Normalt sätt att skapa en aktivitet och initiera resurser i Android.

Med Roboguice

```
class RoboWay extends RoboActivity{
   @InjectView(R.id.sampleTextView)                TextView _sampleTextView;
   @InjectResource(R.drawable.sampleDrawable)      Drawable _sampleDrawable;
   @InjectResource(R.string.sampleString)          String _sampleString;
   @Inject                                         LocationManager _sampleLocationManager;

   public void onCreate(Bundle savedInstanceState){
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);
      _sampleTextView.setText("Skriv ut sampleString: "+_sampleString);
   }

}
```

Fig 4.3 Samma aktivitet som ovan, fast med RoboGuice.

4.5 Utvecklingen och uppbyggnad av JoggKompis/IronLog Runner

Nedan följer en presentation av applikations uppbyggnad, hur vi utvecklade applikationen samt fyra större kategorier Trackning, Statistik, Kommunikation och Övrigt. Dessa kategorier anses vara fyra större segment i applikationen och syftar till beskrivning. Vi ger därför en presentation av dessa, samt vilka klasser och funktioner dessa har. Dessutom ger vi en översiktlig beskrivning av hur testning av applikationen har gått till.

4.5.1 Tidsplan

Arbetet med projekten fortlöpte under hela examensarbetet. Det vill säga läs period 4 (vecka 11-22) 2011, arbetstiden per vecka motsvarade 40 timmar. Arbetet skede främst i BI partners lokaler, men även hos CodeKick i form av veckomöten.

4.5.2 Krav för hårdvaran

Som tidigare beskrivet skall applikationernas huvudsyfte vara ett komplement vid konditionsträning. Detta kräver att användaren har tillgång till telefonens inbyggda GPS. Då telefoner med plattformen Android 2.1 kräver stöd i form av en GPS mottagare kan vi garantera spårning av telefonens rörelsemönster med hög precision. Dessutom har applikationen huvudsyftet att fungera som marknadsföring för hemsidorna, därför bör möjligheten att ansluta till internet finnas. Dock kan applikationen köras i så kallat offline-läge, men är inte rekommenderat. Vidare behövs stöd för touchscreen, skärmstorlekar som helst ligger i spannet small, medium, large, samt även ett mindre internt utrymme på runt två MB, beroende på hur många pass som sparats lokalt. Dock har telefoner med plattformen Android 2.1 eller högre oftast dessa krav som systemkrav.

4.5.3 Behörighetskrav

Då applikationer teoretiskt sätt kan användas för illasinnade syften kräver Android Market användarens godkännande för att aktivera vissa funktioner, då dessa eventuellt kan vara integritetskränkande, kosta pengar eller på annat sätt påverka användarens säkerhet. För att installera applikationen kräver vi därför användarens godkännande på följande punkter.

- **Kontroll av maskinvara** -*Tillåter applikationen att ändra globala ljud-inställningar som till exempel volym.*
  Används för kontroll av hörlursutgången. Om användaren valt att enbart få röstfeedback via hörlurarna
- **Hitta din plats** – *Åtkomst till detaljerade platskällor via enhetens GPS*
  Används för att spåra telefonens rörelsemönster med hjälp av den inbyggda GPS mottagaren.
- **Nätverkskommunikation** – *Fullständig internetåtkomst. Tillåter applikationen att hämta och skicka data via antigen WLAN, 3G eller GMS nätet samt även se status för dessa.*
  Möjliggör synkronisering med Dumbbell.se eller IronLog.com

4.5.4 Spårning

Då applikationens huvudsyfte är att spåra användarens rörelsemönster behövs en bra struktur för detta. Vi valde att implementera en typ av klient/server-struktur. Vilket i teorin delar in vissa av applikationens delar som klienter eller en server. Server-komponenten ska tillhandahålla diverse funktioner eller tjänster till en eller flera klienter. I vår implementation kan TrackingService, vilken

ärver från Android-komponenten Service, ses som en serverkomponent. Den är lokal för applikationen, långt körande samt möjliggör klienter att binda till sig. Under uppstart initierar den diverse trådar, för hantering av spårningsfunktionalitet, samt komponenter så som positionslyssnare, funktion för röstfeedback samt håller koll på statistik härledd från inkomna GPS-punkter. Servicen i sin tur kommunicerar med en eller flera klienter i form av aktiviteterna TrackingActivity, för start och stopp av pass samt visning av realtids-statistik eller RealTimeMapActivity för en realtids-karta. Dock kan tilläggas att dessa inte kan visas samtidigt för användaren under körning, se tekniskbeskrivning om Androids aktivitetstack.

Kommunikationen mellan Servicen och dess klienter sker med hjälp av Handler-objekt. Dessa skickar beroende på vad som skall göras meddelande mellan servicen och aktiviteten. Om ingen aktivitet är bunden till servicen körs servicen ändå, tills en aktivitet har anslutit och begärt antingen information, eller meddelar att applikationen ska avslutas. Fördelarna med denna struktur gör att vi inte är beroende av en specifik aktivitet eller tillstånd i applikationen för att spåra rörelsemönstret. Vi kan alltså avsluta, backa ur, applikationen utan att detta påverkar spårningen. Dessutom kan vi beroende på användarens val visa diverse information.

Vidare körs servicen i förgrunden, för att göra användaren uppmärksam på att spårningen är igång samt även se till så att Android inte mot förmodan stänger av vår service. Då en service default är satt att köras i bakgrunden, vilket ger operativsystemet möjlighet att döda servicen om denna behöver utvinna mer minne, vilket sker rätt frekvent då minnet är en bristvara hos Android-baserade system.

Android spårar telefonen med hjälp av leverantörer, vilka kan leverera information om position via GPS eller WiFi. I vår implementation fastställs positionen enbart via GPS. Detta är för att det systemet oftast är mest tillförlitligt när det kommer till positionens exakthet. Vår implementation kräver, bland annat, att en punkts exakthet är på minst 20 meter. Exaktheten varierar rätt kraftig mellan olika telefon modeller. Så vi har därför blivit tvungna att via manuell testning på olika typer av hårdvara, samt baserat på användarfeedback välja det mest lämpade värdet.

I vår implementation används två olika implementationer av positionslyssnare. Vid spårning används TrackerLocationListener. Vilket beroende på tid eller distans begär punkter från GPS-mottagaren. Utöver lyssnaren vid spårningen används en GpsFixListener som analyserar alla nyregistrerade punkter som GPS-mottagaren får för att påskynda arbetet med att hitta en startpunkt. Vidare filtreras punkterna under spårningen för att eliminera och ta bort en del missvisande punkter.

4.5.5 Statistik

Då applikationen har möjligheten att presentera diverse information baserat på det insamlade punkterna krävs olika klasser som antingen gör beräkningar eller visar resultatet från beräkningarna. Navet för beräkningarna är klassen DataCalculator. DataCalculator gör beräkningar baserat på distans och tid, för att låta applikationen informera användaren om sträcka, tid, tempo samt hastighet. Klassen har även stöd för att beräkna enheter baserade på det internationella enhetssystemet, eller det amerikanska enhetssystemet. Då vi valt att gå ifrån Androids egna positions-klass blev vi tvungna att skapa en egen funktion för att beräkna distans mellan två olika punkter.

Under spårning av telefonen initieras ett objekt av DataCalculator i TrackingService. DataCalculator populeras därefter med punkter från lyssnarna, för att sedan, i realtid, beräkna statistik. Därefter skickas resultatet från beräkningarna, genom våra Handlers, till tjänstens klienter för visning i realtid.

När sedan ett pass är avslutat eller om användaren vill sin historik visas informationen i tre olika informationsaktiviteter.

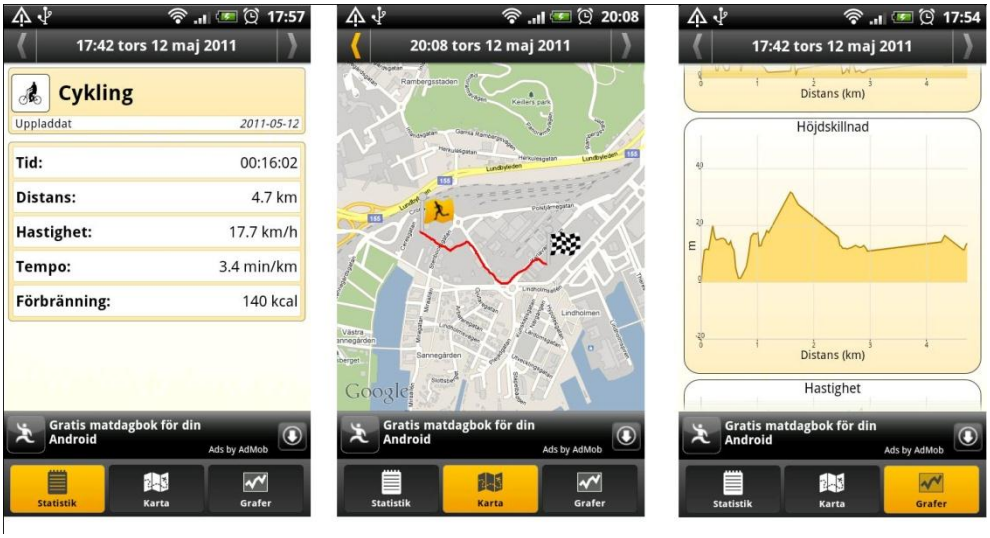


Fig 4.4 Skärmdumpar från det tre olika visningsaktiviteterna som presenterar statistik om passet.

Aktiviteterna håller olika typer av vyer som presenterar olika typer av statistik. För att kunna möjliggöra layouten implementerades en värdaktivitet som håller visningsaktiviteterna. Värdaktiviteten är av typen TabAcitvity vilket är en förlängning av Activity, med skillnaden att de kräver en TabLayout. En TabLayout måste tillhandahålla av en TabHost, vilket är en container innehållandes Intents till de underliggande aktiviteterna och en TabContent vilket är innehållet. Det vill säga TabContent håller aktiviteten som just vid tillfället är synlig. Vidare håller värdaktiviteten information om vilket pass som skall visas i form av en tränings-DTO(Data Transfer Object) (Se 4.5.6 Kommunikation)

SessionActivity (fig 4.4, vänster) visar allmän information om passet baserat på beräkningar utförda av DataCalculator. Aktiviteten visar information om passets träningstyp, distans, hastighet, tempo samt om passet synkroniserats med någon av webbsidorna även förbränning.

MapPlotActivity (fig 4.4, mitten) bygger på Google Maps kart-vy. Kartan målas ut med hjälp av överliggande objekt, som har en egen implementerad rit-metod. Utöver själva sträckan målas även start, stop och kilometer flaggor ut.

GraphActivity (fig 4.4, höger) håller tre olika GraphicalViews, som implementeras från AChartEngine biblioteket. Graferna presenterar kurvor över höjdskillnad, hastighet och tempo för passet.

4.5.6 Kommunikation

Applikationens klasser kommunicerar både internt och extern. För att underlätta sändningen av olika objekt centraliserades och konverterades visa klasserna till att ha strukturen hos ett Data Transfer Objekt, DTO. Så att konvertering mellan objekt hämtade från servern eller skapade internt förenklas.

Exempel på sådan klass är TrainingDto, vilket har samma publika variabler som DTO:n på Dumbbell.se, men har en statisk metod för att skapa ett nytt objekt med hjälp av DataCalculator.

TrainingSessionDto

```
public class TrainingSessionDto {
    public String Key;
    public long Date;
    public long Time;
    public double AverageSpeed;
    public double AveragePace;
    public double Distance;
    public LocationDto[] Locations;
    public int TrainingId;
    public String TypeName;

    public static TrainingSessionDto fromCalculator(DataCalculator calculator) {
        TrainingSessionDto dto = new TrainingSessionDto();
        dto.AverageSpeed = calculator.getAvgSpeed();
        dto.Distance = calculator.getTotalDistance();
        dto.Time = calculator.getTotalTimeMilliS();
        dto.Locations = calculator.getLocationsGSON();
        dto.AveragePace = calculator.getAvgPace();
        return dto;
    }
}
```

Fig 4.5 TrainingSessionDto.

Om objektet sedan skulle skickas konverterades det till ett JSON-objekt via GSON-konvertering, och skickades därpå vidare som en JSON-sträng. Vilket hos mottagaren lätt kan konverteras tillbaks till ett liknande objekt via omvänd konvertering.

Synkroniseringen med webbsidorna sker i sin tur via CodeKicks web-API. För att kunna kommunicera med servern implementerades en kommunikations bas-klass. Vilket kommunicerar med webtjänsten i form av http GET och http POST-anrop. Bas-klassen kunde sedan användas vid olika typer av synkronisering så som vid inloggning, registrering samt uppladdning och borttagning av pass på servern.

För att möjliggöra interaktion mellan användaren och applikationen i kommunikationssyften skapades RegistrationActivity och LoginActivity. Aktiviteterna låter användaren mata in, samt göra val angående inloggning samt registrering. Om användaren lyckas logga in eller registrera ett konto sparas sedan information om användaren i en UserSessionDto. DTO:n består av information om den inloggade användaren och en ticket vilket används vid synkronisering av data. För att sedan göra det möjligt att manuellt synkronisera samt byta användare skapades aktiviteten DumbbellActivity. DumbbellActivity visar det just nu inloggade användarnamnet, samt möjliggör, via en knapp, manuell synkronisering. För att veta vilka pass som inte är synkroniserade söks alla tillgängliga tränings-DTO:er igenom efter ett tränings-ID med värdet noll. Om passet inte är synkroniserat inleds ett försök att synkronisera passet. När sedan passet är korrekt sparat på servern returnerar servern ett giltigt tränings-ID skilt från noll.

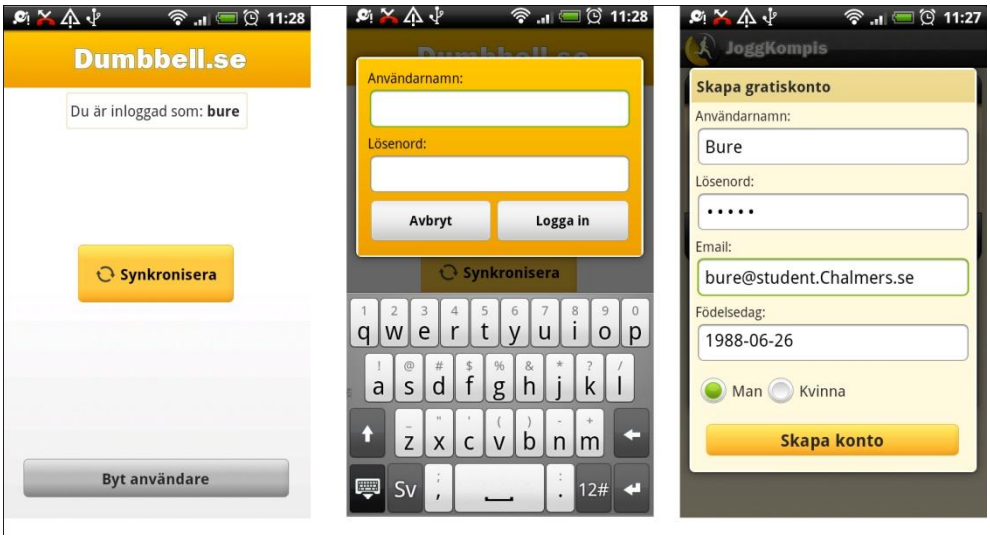


Fig 4.6 skärmdumpar på DumbbellActivity (vänster), LoginActivity (mitten), RegistrationActivity (höger)

4.5.7 Övrigt

Under ovanstående rubrik kommer vi ta upp centrala och viktiga klasser som inte hamnar under någon av ovanstående kategorier, eller som behöver förklaras djupare.

Då Androids egna klass för positionspunkter ansågs innehålla för mycket data skapades en lokal egen Location klass, med strukturen hos en DTO. Detta för att främja mindre minnes-allokering under körning men även simplifiera sparningen på webbsidorna. Klassen består, bland annat, av variabler för longitud, latitud, höjd och tid. Förutom fördelarna med mindre data kan vi även applicera vanliga enhetstester på klassen då den saknar beroendet till Android SDK, vilket möjliggör kompilering i Java VM istället för Dalvik VM.

Då testdriven utveckling strävar efter den lättast möjliga lösningen använder vi oss av en key/value-databas för sparningen av lokal data. Då Android 2.1 har ett inbyggt stöd för SQLite-databaser valde vi att gå ifrån den tidigare iden om att spara tidigare sessioner i en textfil, vilket inte ansågs vara tillräckligt, till att istället implementera en key/value-databas. Key-value-lagring innebär att all data, oavsett om de är relaterade eller ej, sparas i samma tabell. I praktiken innefattar det en databas beståendes av två kolumner. Den första kolumnen representerar nyckelvärdet och den andra kolumnen består utav sparad data, i vårt fall en JSON sträng som är en konverterad DTO-klass.

Då CodeKick, efter några veckor in i projektet, bestämt sig för att lansera en engelsk version av Dumbbell.se behövdes det skapas en ny applikation(IronLog Runner). Den nya applikationen kräver synkroniseringsmöjligheter riktade mot IronLog men med ett annat språk, grafik samt ytterligare funktionalitet så som inställningar för enhetssystem. Därför konverterades JoggKompis, till ett Android-bibliotek. Detta gör det möjligt att beroende på vilken applikation som ärver biblioteket låta viss funktionalitet eller språk vara inställt, även då två olika applikationer ärver från samma bibliotek. Detta möjliggör dessutom att i framtiden lansera en betalversion med ytterligare funktionalitet men utan att skapa ett eget projekt.

Vidare implementerade även en inställnings-aktivitet som gör det möjligt att ställa in diverse olika inställningar. Inställningar så som Röstfeedback i form av alltid, aldrig eller med hörlurar. Möjlighet att automatisk synkronisera efter avslutat pass. Inställningarna sparades sedan i Androids

SharedPreferences för applikationen. SharedPreferences ger möjligheten att spara data mellan sessioner av applikationen. Dessa inställningar kan hämtas ut av en aktivitet eller service som är implementerad i samma paket som inställningen.

Då gränssnittet är viktigt för helhetsintrycket av applikationen valde vi att skapa diverse NinePatch. NinePatch är modifierade digitala bilder som har formatet Portable Network Graphics(.png). Klassen NinePatch tillåter uppritning av en bitmap i nio sektioner. De fyra hörnen är oskalade, det fyra kanterna skalas i en axel, samt mitten skalas i båda riktningarna. Normalt skapas en yta, med formen av en rektangel, som sedan innehållet placeras inuti. Det möjliggör att man kan skapa bilder som skalas på ett eget definierat sätt, även om upplösningen varierar. Detta användes till en början för samtliga knappar i applikationen.

4.5.8 Design

Då applikationens utseende är väldigt viktigt för helhetsintrycket samt användarupplevelsen lades mycket tid på att skapa ett tilltalande samt funktionellt gränssnitt. Då kravet även var att applikationen skulle vara självinstruerande behövdes fokus läggas på att göra användargränssnittet så enkelt som möjligt. En funktionell skillnad från standardiserade Android applikationer är att vi lagt TabHosten i värdaktiviteten lägst ned på skärmen skilt från Androids sett att lägga sådana komponenter högst upp. Detta tycker vi underlättar hanteringen av byte mellan aktiviteterna då vi oftast har lättare att peta långt ned på telefonen istället för högt upp. Dessutom kommer dessa knappar att användas mer frekvent än bytet av tidigare pass, som nu ligger högst upp. Se fig 4.4

Då Android förespråkar att vi antingen använder NinePatch (se 4.5.7) eller skapar designen med hjälp av XML valdes till en början NinePatch. Detta uppdagades rätt snabbt att färgåtergivningen samt även fast bilderna var skalbara skalade underligt på visa modeller gick vi över till att använda XML så långt som möjligt. Då vi vet att applikationen fungerar som tänkt på alla typer av hårdvara.

icon_background.xml

```
<?xml version="1.0" encoding="utf-8"?>
<layer-list xmlns:android="http://schemas.android.com/apk/res/android">
<item>
   <shape android:shape="rectangle">
   <corners android:bottomLeftRadius="@dimen/corner_radius"
         android:bottomRightRadius="@dimen/corner_radius"
         android:topLeftRadius="@dimen/corner_radius"
         android:topRightRadius="@dimen/corner_radius"/>
   <gradient android:angle="90" android:startColor="#ffffff"
          android:endColor="#ffffff" />
   <stroke android:width="1px" android:color="#222222" />
   </shape>
   </item>
</layer-list>
```

Fig 4.7 Design med hjälp av .xml. Används på samma sätt som en drawable.

Fig 4.8 Skärmdumpar på aktiviteter som inte tidigare visas i rapporten. Val av träningstyp (höger), TrackingActivity (mitten), val av inställningar (höger)

4.5.9 Test av applikationen

Testdelen av projektet består av två moduler, ett för enhetstester samt ett för tester av Android-komponenter. Enhetstestet behöver ligga i en separat modul som kör testen i JVM, då vi insåg tidigt att man inte kan förlita sig på Androids testramverk för att köra enhetstester. Det tar alldeles för lång tid att köra testen, från det att man tryck på start. Lyckligtvis är det i IntelliJ enkelt att dela upp ett projekt i moduler, sätta beroenden dem emellan.

Modulen för Android-test består nästan enbart av testklasser vars bas-klass är ActivityInstrumentationTestCase2, till största del på grund av att det är den enda klass Robotium stöder. En aktivitet är det användaren interagerar med och ofta beroende av andra Android-komponenter, så vi anser att de bäst testas genom black box-testning trots att möjlighet ges att testa dem i isolation.

Även test av vår tjänst går implicit via nämnda klass ActivityInstrumentationTestCase2, då vid test av TrackingActivity. Vi beslutade oss för att inte använda ServiceTestCase, då vi hade svårigheter i att se hur den kan användas för att testa vår Service på ett meningsfullt sätt. Ytterligare motivering är att aktiviteten och tjänsten är så pass starkt beroende av varandra, det är såpass lätt att skriva test med Robotium och att vi ändå vill testa den delen som helhet.

Mockning av beroenden, både för enhets samt Android-testen sker med hjälp av RoboGuice. För att detta ska uppnås definierades en RoboGuice-modul, där bindningar till mock-klasser sker. RoboGuice ger dessutom ytterligare fördelar i form av kod-renhet.

5. Resultat

Det ska vara möjligt att testa delar av en applikation i isolation från övriga delar, i form av enhetstest. Enhetstesten måste vara snabba att köra.

Androids testramverk erbjuder möjlighet att skriva automatiserade enhetstest. Dock räcker det inte för att uppfylla kriteriet då körningen av testen blir för långsam. För att kringgå begränsningarna måste man i utvecklingsprojektet inkludera ett delprojekt, där enhetstesterna körs i JVM.

Det ska vara möjligt att testa en applikation i sin helhet. Testen ska vara automatiserade, lättlästa samt enkla att implementera.

Androids testramverk erbjuder en bas-testklass där det är möjligt att, automatiserat, testa en aktivitet som körs och interagerar med systemet på samma sätt som om den kördes normalt. Men endast användning av den klassen räcker inte för att uppfylla kriteriet då testen blir både svår-skriva samt svårlästa. För att kriteriet ska bli uppfyllt anser vi det vara nödvändigt är att inkludera Robotium som tredjeparts-bibliotek. Dock kan man få svårigheter att uppfylla kriteriet om den applikationen man utvecklar i för hög grad beror av aktiviteter tillhörandes externa applikationer.

6. Diskussion och slutsats

Nedan kommer vi att diskutera vårt resultat, diskutera allmänt om testdriven utveckling på Android-plattformen, presentera en slutsats samt redogöra för erfarenheter via skaffat oss under projektets gång.

6.1 Diskussion av resultat

Den bas-testklass som erbjuds av Android, med vilken enhetstestning kan bedrivas, är AndroidTestCase. Eftersom klassen är baserad på JUnit så innebär det att man kan skriva automatiserade test. Dock, då testpaketet måste laddas in och köras i emulatorn innebär det att, från när man har tryckt start, det tar lång tid att köra testen. Vi anser det därför vara ett måste att ha ett separat projekt, enbart för enhetstester, som kör testerna i JVM. Detta kräver en bra IDE, där man på ett enkelt sätt kan sätta beroenden mellan enhetstest-projektet och Android-projektet. Konflikterna mellan olika VM medför dessutom att man måste ”wrappa” Android-klasser om man vill enhets-testa objekt beroende av dem. Detta har dock inte setts som ett stort hinder i utvecklingsarbetet och bör inte heller påverka bedömningen av resultatet.

Android erbjuder, i och med testklassen ActivityInstrumentationTestCase2, möjlighet att testa en delmängd, med möjlighet till interaktion med övriga delar, av en applikation. Till exempel kan aktiviteter, vilka vanligtvis används långt senare i applikationens livscykel köras direkt och interagera med dess omgivning (jämför med att man skulle klicka igenom applikationen i emulatorn varje gång man vill kontrollera något i en specifik del). Dock har den upplevts som svår att använda, vilket medför svårigheter att ha en test first-approach, men tack vare Robotium, finns det möjlighet att skriva black box-test som både är lätta att skriva samt lättlästa. Robotium tillhandahåller enkla mekanismer för använde av UI-komponenter, samt synkronisering med händelseförlopp hos test-objektet vilket leder till att det blir naturligt att skriva dem som användarfall och uppfyller därmed ett viktigt dokumentationssyfte. Dock finns det en begränsning i att man för närvarande endast kan testa en applikation åt gången. Då Android erbjuder applikationer, via Intents, möjlighet att aktivera komponenter i andra applikationer kan problem uppstå. Exempelvis kan användaren sakna en telefoninställning, varvid test-applikationen anropar en aktivitet från en extern applikation där

inställningarna kan ändras. Påverkas då test-applikationens tillstånd beroende på inställningarna användaren valt, vill man testa det. Detta går inte för närvarande då test-klassen inte kan styra aktiviteter bortom aktiviteter tillhörande applikationen under test och medför att hur stor del av applikationen som kan testas funktionellt begränsas av hur mycket den beror av andra applikationers komponenter.

Man kan visserligen enkelt testa applikationen manuellt i emulatorn, men många av fördelarna med en testdriven approach går ändå förlorad. En test first-approach kan visserligen hållas, men kommer i längden att bli tröttsam. Vart man ska dra gränsen är dock helt omöjligt att säga och utövaren får göra en personlig bedömning där hänsyn tas till hur viktig funktionaliteten, samt hur stor del av applikationen, som inte kan testas är.

6.2 Övrig diskussion

Då marknaden för Android-applikationer är expansiv samt att mål-enheterna skiljer sig åt upplever vi att testdriven utveckling som praktik bör lämpa sig väl. Det upplevs vara gynnsamt att ha tidigare test att förlita sig på när modifieringar behöver göras och ny funktionalitet tillkommer. Korta releaser ger troligen en fördel då marknaden ändrar sig snabbt, samt att ny plattformsfunktionalitet introduceras relativt ofta. Dessutom är det väldigt smidigt att släppa frekventa uppdateringar med hjälp av verktygen Android Market tillhandahåller. Applikationerna är oftast av storleken att man utan problem kan skissera en översiktlig lösning som ett långsiktigt mål utan att omfattande förarbete behöver göras, samtidigt som plötsliga ändringar kan prioriteras in i arbetet.

Oftast blir Android-kod, exempelvis involverande anrop till systemtjänster eller initiering av vyer, ganska rörig då man oftast behöver göra nästlade funktionsanrop. RoboGuice kan då vara till stor hjälp då den med en enkel annotations-syntax sätter upp dylika anrop. Ytterligare en fördel RoboGuice ger är enkel sättning av beroenden mellan klasser. En klass kan ha många beroenden vilket medför rörig kod när beroenden ska sättas. Dessutom har beroende-injicering den goda effekten att mockings-mönstret enkelt kan följas, som dessutom, i fallet med ActivityInstrumentationTestCase2 ger att även aktiviteterna kan testas tämligen modulärt. Vi anser det därför vara lämpligt att använda sig av RoboGuice om man ska tillämpa en testdriven metodik på Android.

6.3 Slutsats

På det stora hela kan vi sluta oss till följande. Testdriven utveckling som metodik, utifrån de kriterier vi använt, kan i de flesta fall praktiseras på Android, dock med hjälp av en bra IDE och Robotium. De fall där det inte är tillämpbart är de där applikationen man utvecklar är allt för beroende av andra applikationers komponenter.

Syftet med att undersöka och inhämta kunskap om så väl testdriven utveckling som Android-utveckling, samt att använda lämpliga ramverk anses vara uppfyllt. Kunskapen vi nu besitter anser vi vara god och kommer att vara oss till hjälp i framtida projekt.

6.4 Kritisk diskussion

Om vi skulle få ett liknande projekt idag skulle vi troligtvis försöka ta det lite lugnare i början. Det vill säga, läsa på mer om plattformen innan den skarpa programmeringen påbörjas. Android är lätt att komma igång med, men har ett stort djup. Ett relaterat misstag var att inte omgående skapa en gren i Git, vilken hanterar mer utforskande programmering, relaterad till inlärning. Detta innefattar extra

viktigt om man bedriver testdriven utveckling, då metoden kräver att man har god kunskap om plattformen vilket vi, till viss del, inte hade i början.

Vidare skulle vi inte valt att implementera en avancerad struktur på applikationen i ett för tidigt skede. Vi identifierade tidigt att vi ville att användaren ska ha stor frihet i hur han kan hoppa ut från aktiviteter, trots att ett pass är påbörjat. Detta visade sig kunna leda till problematik med back-stacken, vilket inte upptäcktes i de tester vi hade.

En större mängd tid har gått åt till att lära sig att använda Androids testramverk och applicera det på vår applikation. I ljuset av de erfarenheter vi nu tillskansat oss kommer vi i framtida projekt att omgående introducera RoboGuice och Robotium omgående. Dessutom finns det förmodligen ytterligare bibliotek att använda sig av. Exempelvis verkar Robolectric verkar lovande, då möjlighet ges att köra enhetstester av Android-komponenter utanför emulatorn.

Vidare bör designen av applikationen till största del skapas med hjälp av XML istället för NinePatch. För att underlätta att applikationen ter sig lika på olika typer av hårdvara. Vi vill att applikationen oavsett färgåtergivning eller skärmstorlek, skall ge det utseende som är tänkt.

För applikationer innehållandes en Service, vars stora syfte är att vara värd för trådar, kommer vi att inte bedriva testning mot Servicen med hjälp av ServiceTestCase2. I stället kommer black box-testning, med ActivityInstrumentationTestCase2 och Robotium att användas.

Android-applikationer kommer oftast i flera versioner. För att på lättast möjliga sätt kunna skapa, underhålla och hantera dessa likande applikationer borde man i ett tidigare skede implementera eller konvertera applikationen till ett Android-bibliotek och tillhörande applikations-specifika paket. På så sätt har man möjlighet att lättare strukturera paketstrukturen samt underlätta vid ny utvecklingen av samtliga applikationer då man slipper göra stora ändringar i applikationen vid konvertering.

7. Referenser

Ableson, F.W., Sén, R., & King, C. (2011). Android In Action

About SQLite. (n.d.). Retrieved June8, 2011, fromhttp://www.sqlite.org/about.html

Activity. (2011). Retrieved June8, 2011, fromhttp://developer.android.com/reference/android/app/Activity.html

Application Fundamentals. (2011). Retrieved June8, 2011, fromhttp://developer.android.com/guide/topics/fundamentals.html

Backward Compatibility for Applications. (2011). Retrieved June8, 2011, fromhttp://developer.android.com/resources/articles/backward-compatibility.html

Handler.(2011). Retrieved June8, 2011, fromhttp://developer.android.com/reference/android/os/Handler.html

Hashimi, S., Komatineni, S., & Maclean, D. (2010). Pro Android 2

Intent. (2011). Retrieved June8, 2011, fromhttp://developer.android.com/reference/android/content/Intent.html

JUnit. (n.d.). Retrieved June8, 2011, fromhttp://junit.sourceforge.net/

Location. (2011). Retrieved June8, 2011, fromhttp://developer.android.com/reference/android/location/Location.html

Managing Virtual Devices. (2011). Retrieved June8, 2011, fromhttp://developer.android.com/guide/developing/devices/index.html

Martin, R.C., & Martin, M. (2007). Agile Principles, Patterns, and Practices in C#

Messenger. (2011). Retrieved June8, 2011, fromhttp://developer.android.com/reference/android/os/Messenger.html

Obtaining User Location. (2011). Retrieved June8, 2011, fromhttp://developer.android.com/guide/topics/location/obtaining-user-location.html

Osherove, R. (2009). The Art of Unit Testing with Examples in .Net

Services. (2011). Retrieved June8, 2011, fromhttp://developer.android.com/guide/topics/fundamentals/services.html

Sommerville, Ian. (2010) Software Engineering 9ed

Testing Fundamentals. (2011). Retrieved June8, 2011, fromhttp://developer.android.com/guide/topics/testing/testing_android.html

Tasks and Back Stack. (2011). Retrieved June8, 2011, fromhttp://developer.android.com/guide/topics/fundamentals/tasks-and-back-stack.html

Using DDMS. (2011). Retrieved June8, 2011, fromhttp://developer.android.com/guide/developing/debugging/ddms.html

What is Android. (2011). Retrieved June8, 2011, from http://developer.android.com/guide/basics/what-is-android.html