WatchMe: Projektreflektioner

Refl Team

Att utveckla en applikation i grupp med andra skiljer sig mycket från att utveckla på egen hand. Olika uppgifter kan delegeras till medlemmarna i gruppen som sedan utför dem, vilket betyder att vissa väl avgränsande delar av utvecklingsprocessen inte nödvändigtvis behöver vara annorlunda. Vad som skiljer sig är däremot det jobb som krävs för att sammanfoga det varje individ utvecklar på egen hand till något som bildar en helhet.

En av de största skillnaderna när man utvecklar ett program i grupp är all tid som måste läggas på kommunikation gruppmedlemmarna emellan. Man måste bland annat komma överens om vad det är man egentligen ska utveckla, vem som ska göra vilken del av detta, hur olika uppgifter ska utföras, etc. Att tillsammans med andra människor bestämma hur ett visst UI-segment ska se ut, vilken som är den bästa designen på koden att använda, och så vidare är en helt annorlunda process än att, som när man utvecklar ett program själv, ta sådana beslut på egen hand. Framför allt är det mer tidskrävande. Man kan alltså inte räkna med att en arbetsuppgift som på egen hand hade tagit 40 timmar i en grupp på fyra individer tar 10 timmar. Den tid det tar att bestämma vad som ska göras innan man kan börja producera kommer att vara längre vid arbete som sker i grupp.

När man utvecklar något i grupp krävs också av gruppmedlemmarna att de har förmågan att anpassa sig. Dels i fråga om att anpassa sin tid så att man kan ha möten i gruppen, men kanske framför allt i fråga om hur man föreställer sig den färdiga produkten och hur man arbetar. Alla har vi olika erfarenheter, olika ideer om vad som är kvalitativt och olika visioner om hur den färdiga produkten bör se ut. Att ha möjlighet att tumma på den egna bilden kan vara ansträngande för en individ och är ingenting man behöver bry sig om då man utvecklar på egen hand. För att åstadkomma en gemensam slutprodukt är det emellertid en nödvändighet.

Att jobba i grupp när man utvecklar ett program innebär även en hel del fördelar som gör att processen blir mer effektiv och slutresultatet kan bli bättre än när man programmerar på egen hand. Den viktigaste fördelen är kanske att alla gruppmedlemmar har olika kunskaper som, om gruppen fungerar bra, tillsammans genererar en större kunskapsbas som grund för utvecklandet än när man jobbar på egen hand. Om detta går att utnyttja kan många delar av utvecklingen effektiviseras och ske snabbare. I vårt fall hade till exempel vissa medlemmar tidigare programmerat mot databaser. Programmet vi bygde innehöll en databas och för de medlemmar som inte besatt kunskap om databaser innebar det en stor tidsmässig vinst att snabbt kunna få hjälp av någon som hade kunskap.

När man utvecklar ett program i grupp delegeras olika uppgifter till olika medlemmar som utför dem på egen hand. Detta innebär för varje individ i gruppen att programmet växer på håll som individen själv inte varit med och bidragit till och därför inte har koll på. När man utvecklar själv tvingas man själv producera alla delar av programmet. Man kanske börjar med modellen, och sedan går vidare till UI:t eller databasen.

Följden blir att man vet hur alla delar av programmet fungerar, vad varje kodrad gör, hur programmet är sammansatt, etc. I ett grupparbete däremot kanske en individ får i uppgift att skriva modellen, en UI:t och en databasen. Förr eller senare måste emellertid komponenterna sättas ihop och kommunicera med varandra. Då uppstår problemet att eftersom man själv inte gjort de andras komponenter vet man inte hur dem fungerar. En stor del av programmerandet när man utvecklar i grupp går således ut på att försöka förstå vad nån annan har gjort och hur man ska programmera mot detta. Här är det viktigt att kommunikationen i gruppen fungerar bra.

Vi känner att vi under projektets gång tydligt har märkt av alla fördelar som följer med att jobba i grupp. Detta för att samarbete och kommunikation fungerat utomordentligt i vår grupp. Alla gruppmedlemmar har kompletterat varandra bra kunskapsmässigt och ...

Refl Doc

Även om vi jobbade med Scrum där dokumenterande utgör en förhållandevis liten del av processena har ett flertal dokument nedtecknats. Detta har gjorts med hjälp av Googles dokumenthanteringssystem Google Drive. Fördelarna med Drive är bland annat att dokumenten är publika och uppdateras i realtid när någon ändrar dem. Dokumenterandet har ofta skett i samlad grupp. Detta är inte alltid det mest effektiva alternativet, men tack vare Google Drives möjlighet att låta flera personer se och redigera samma text har vi lyckats engagera alla i gruppen under aktiviteter såsom testrapportering, Scrum-möten, m.m.

De flesta dokument som nedtecknats i denna kurs har emellertid varit av en sådan art att de ger upphov till diskussioner som med fördel avhandlas tilsammans i gruppen. Detta gäller till exempel Defenition of Done. Vid nedtecknandet av denna uppkom frågor om vad som den egentligen skulle innehålla. Sitter man då i samlad grupp kan man tillsammans besluta om detta.

I andra fall krävdes alla gruppmedlemmars närvaro eftersom kunskap om alla delar av programmet behövdes vid dokumenterande. Detta gällde till exempel vid skrivande av Release Notes och Test Report. Om alla ändringar i programmet ska tas med och alla delar av programmet ska testas är det fördelaktigt om alla som utvecklat någon del av programmet är närvarande så att man testar alla delar av programmet, ingen bug missas att ta med, etc. Detta har också för vår del blivit en naturlig situation att få större överblick av de delar av programmet som de andra gruppmedlemmarna utvecklat.

Att jobba med Scrum som metod innebär att dokumenterandet sker utspritt under utvecklingsprocessen. Detta gör att behovet av att versionshanterandet av dokumenteten minskas. Om man jobbar enligt en Vattenfalls-process görs en stor del av dokumentationen tidigt i processen. Man gör till exempel en RAD och en SDD som beskriver programmet och om programmet inte blir riktigt som man tänkte sig från början, vilket ofta är fallet, måste RAD:en och SDD:n anpassas efter detta. I Scrum tillkommer dylika dokument, till exempel Architecture Specification, något senare i processen då man har något bättre koll på hur programmet verkligen kommer att se ut. Man behöver då inte versionsanpassa dessa dokument i lika hög grad eftersom att programmet inte ändrar sig så mycket gentemot dem.

Vissa dokument kräver emellertid versionanpassning även vid anammandet av Scrum. Detta gäller till exempel i allra högsta grad Sprint Planning, som ju ändras varje vecka då man inleder en ny sprint och förflyttar User Stories från Backlogen till den nya Sprint Plan:en. Generellt sett gäller egentligen att alla dokument behöver anpassas till en viss grad, men eftersom att dokumenterandet sker mer utspritt under utvecklingsprocessen, inte i lika hög grad som när man anammar Vattenfallsprocessen.

En del av dokumentationen i detta projekt finns även i det digitala projektverktyg vi använde oss av: Pivotal Tracker (pivotaltracker.com). Det är ett webbaserat verktyg för team, och förhåller sig bra till agila utvecklingsmetoder i mjukvaruprojekt (se mer nedan). I de olika typerna av stories som finns i vårt Pivotal-projekt har vi använt oss av beskrivningar och kommentarer för att kommunicera och dokumentera. Pivotal har använts för att skapa vår backlog, och för att rapportera buggar. För varje inlämning har vi flyttat över Stories från Pivotal Tracker till ett dokument i Google Drive, som sedan har lämnats in.

De bästa projekten som finns tillgängliga som öppen källkod är oftast dem med bra dokumentation. API-dokument, ReadMe, stilguider är strukturerade och uppdaterade. Det finns till och med en ideologi, *Readme Driven Development*, som manar utvecklaren att skriva Readme-filen först (läs mer om ämnet här: http://tom.preston-werner.com/2010/08/23/readme-driven-development.html). Under projektet har vi lärt oss mer om att skriva strukturerade Release Notes, testrapporter, och andra nyttiga dokument inom mjukvaruutveckling.

Vi har trivs mycket bra med att använda scrum som utvecklingsprocess. Framför allt för att det ger en bra överblick över var man ligger i projektet, via Pivotal har vi haft möjlighet att hela tiden kunna se vad de andra gör och vad som ska hinnas med under en specifik sprint. Alla medlemmar i gruppen känner att scrum är en utvecklingsprocess som vi helt klart skulle kunna tänka oss att jobba med igen.

Refl SE

Vi valde att jobba enligt Scrum-processen när vi utvecklade vårt program. Vi har alla tidigare gått en kurs där Vattenfall anammades och detta gjorde att lärorika jämförelser kunde dras. Skillnaderna mellan processerna var flera. Vissa talade i fördel för Scrum och vissa för Vattenfall.

Något som tilltalade oss hos Scrum var den snabba, iterativa modellen som ofta används. Att snabbt producera prototypkod för att iterera över har passat oss bra, och att dela upp projektet i mindre delar känns naturligt och lättare att jobba med. Att varje vecka ha haft en färdig produkt som kan levererats har varit skönt, eftersom vi har haft milstolpar att rätta oss efter. Det binds samman med versionshanteringen av applikationen, där vi för varje veckovis sprint har släppt en ny version.

Även när man i denna kursen använt Vattenfall ska man ju ha haft en färdig release varje vecka och implementerat olika features beroende på hur viktiga de är. Vi tycker dock att detta arbete genom Scrum har blivit extra tydligt. Man har varje vecka under sprint planning verkligen fått väga user stories mot varandra och evaluera hur mycket man kommer hinna med till nästa release. Vid varje veckas slut har man haft en färdig produkt och det har varit enkelt att dynamiskt under utvecklingsprocessens

gång anpassa hur programmet ska se ut efter hur mycket tid vi har och hur mycket tid olika features tar att implementera, vilket är mycket enklare att förstå när man börjat implementera det än i ett planeringsstadium.

Vad som också var positivt med Scrum var att man kom igång med kodandet tidigt i processen. Planering och dokumentation är vitalt i en utvecklingsprocess, men eftersom man inte direkt producerar något kan det ibland kännas som man inte kommer framåt när man ägnar sig åt det. Man rör sig inte mot de 4 000 rader kod som ska finnas om några veckor. Detta kan vara stressande, speciellt i början av processen när man inte har implementerat mycket. Denna stress slapp man vid anammandet av Scrum då man kunde komma igång med småsaker redan första veckan.

En annan nackdel med den planering som finns i början när man jobbar enligt Vattenfalls-metoden men som man slipper i Scrum är avvägningen som krävs i hur mycket tid man bör lägga på planeringen. Ju mer man planerar i början av processen, detso bättre grund har man för det framtida byggandet av programmet. Man måste emellertid ha tid kvar till själva byggandet och vad som är lagom tid att lägga på planerande och lagom tid att lägga på implementerande kan vara svårt att avgöra. Kanske lägger man en oerhört bra grund med noggrann planering men hinner inte göra denna planering rättvisa eftersom man inte har tid nog kvar att implementera den. Vid anammande av Scrum märker man under projektets gång hur lång tid som är värt att lägga på diskussioner om modellens design, dokumenterande, et cetera och hur mycket tid man måste lägga på faktiskt kodande.

Denna frånvaro av planering tidigt i processen är emellertid också en av Scrums största nackdelar. När alla gruppmedlemmar sätter igång med programmerande tidigt utan att grundligt ha kommit överens om vad man tillsammans ska åstadkomma kan det hända att man jobbar åt olika håll. Detta kan innebära att man måste göra om saker eller måste förkasta saker som gjorts i onödan. Vid anammandet av Scrum vet man genom user stories vad som ska göras, men det är kanske hur detta ska göras som till större del är upp till var och en än vid Vattenfalls-metoden.

Slutsatsen är ändå att i ett kort projekt som detta (på ett fåtal veckor) är Scrum att föredra. Detta för att man snabbare kommer igång vilket är viktigt när tiden att färdigställa något är knapp. Vattenfall minimerar förvisso felsteg i processen men när det handlar om ett så litet projekt som vi har gjort i denna kursen blir inte de eventuella felstegen så stora, och är relativt enkla att rätta till. Extra passande har Scrum varit eftersom vi utvecklat ett program till android, något som ingen av oss testat innan. När man har dålig koll själva tänket kring det man ska programmera mot, till exempel vad är en Activity, hur är kopplingen mellan olika Activities, var placerar man UI-kod/kontroller-kod, et cetera, är det svårt att planera allt för mycket i förväg. Bättre har varit att testa på lite, lära sig, och sedan allt eftersom upptäcka hur programmet borde se ut.

Projekthanteringsverktyg: Pivotal Tracker

Vår användning av ett agilt projekthanteringsverktyg har varit till stor hjälp. Det heter Pivotal Tracker (pivotaltracker.com), och inbegriper många av Scrums metodologier och begrepp. Pivotal Tracker är byggt kring Stories, som kan kategoriseras som Features, Bugs, eller Chores. När man första lägger till en ny Story hamnar den i en Icebox. Därifrån drar man in den till Backlog, där man prioriterar alla Stories,

och delar upp dem i Sprintar. De längst upp i ordningen hamnar i kolumnen Current.

Det finns olika lägen för varje Story i Pivotal Tracker: Not started, Started, Finished, Delivered och Done. På så sätt har vi alltid haft koll på sprintens fortskridande, och vem som gör vad. När en Story är avklarad sätter man den som Finished. När den är färdig att godkännas bör den göras till Delivered, och då kan en annan medlem i gruppen inspektera Storyn, och godkänna den om allt ter sig till vår Definition of Done. I vårt projekt har en story markerats som Finished när en gruppmedlem är klar med den på sin lokala dator, för att sedan sättas som Delivered när koden har pushats till den aktuella sprintbranchen för allmän inspektion av de andra.

Vi har även använt Pivotal Tracker för buggar och allmänna att-göra-saker. Just för buggar i applikationen hade vi kunnat välja GitHubs mycket eleganta Issue Tracker, som erbjuder en mängd användbara funktioner. GitHubs buggrapporteringssystem hade varit bra att använda för spårbarheten av buggar genom kedjan "upptäckt, registrering, commit, och slutförande". Anledningen är att vi kände att information lätt skulle bli fragmenterad och oöverskådlig om det skulle existera på Pivotal Tracker, Google Drive, och GitHub samtidigt. Att ha buggrapporter inne i Pivotal, nära övriga Stories, kändes mer naturligt, och därför var vi villiga att offra GitHubs funktioner som är tätare bunden till koden. Ser man på det genom ett Scrum/Agile-perspektiv så ska vissa buggar existera i Pivotal Tracker för kunden att se. I vissa fall är det även upp till kunden eller beställaren att själva rapportera buggar, och då är Pivotal ett mer självklart val än GitHub.

Det har gått väldigt bra att använda verktyget under projektet, och vi rekommenderar det starkt till andra. Den överblick och smidighet som Pivotal erbjuder har gjort att vår kunskap om Scrum har fördjupats ytterligare.

Refl Coverage

Främst i den senare delen av projektet har vi använt oss av analyseringsverktyg för bland annat täckningsgrad för automatiska tester och kvalitet på vår källkod. De som använts har varit Emma, FindBugs och Sonar. Verktygen var hjälpsamma i stor mån, speciellt när vi hårdkörde testning av koden, då vi fick hjälp med att analysera och hitta delar av applikationen som ännu inte testats. Att gå igenom det manuellt hade inte varit optimalt. Emma hjälpte oss visa på kodnivå vilka rader som ännu inte täckts upp av automatiska tester, medan Sonar och FindBugs analyserade kvaliteten på koden: allt i från glömda utskrifter till oanvända importer. Allt som allt kändes det bra att ha ett automatiserat verktyg som utförde det tunga arbetet åt oss.

Det som analyseringsverktyg också kan hjälpa till med är att hålla sig till de kodkonventioner som finns. Man kan till exempel få rapporter som säger *"A local class should not have more than 20 lines"*, vilket kan vara lätt att glömma under arbetet med koden.

Vi har också använt en Eclipse-plugin som heter STAN för att analysera vår applikations paketstruktur och för att få en övergripande bild över de beroenden som finns i vår applikation. När man vill göra något sådant är automatiska analysverktyg oumbärliga. Att manuellt försöka se igenom alla de beroenden som

finns mellan alla olika klasser i en applikation hade tagit enormt lång tid. Med STAN görs detta snabbt och enkelt och visas i ett överskådligt diagram som hjälper en att få en bra överblick över applikationen och visar var potentiellt icke önskvärda cykliska beroenden finns.

Dock är det viktigt att veta att analyseringsverktyg bör ge råd och fingervisningar – inte vara en full sanning som ska efterföljas till varje pris.

General comments

Användandet av automatiska tester

Automatiska tester av kod är nästintill krav för mjukvaruprojekt idag – vare sig man jobbar i grupp eller individuellt; på ett stort affärssystem eller mindre open source-projekt. Tester gör att man kan känna sig säkrare i implementationen av nya funktioner eller buggfixar, eftersom man alltid har sina tester att falla tillbaka på. I gruppen har vi förut arbetat med enhetstester i Java och andra språk, men det skulle visa sig att det var mer komplicerat till en början i Android.

I applikationen har vi en mycket liten domänmodell (endast bestående av två klasser i skrivande stund). Modellen kan enkelt testas med standardklasser, men det övriga systemet är mer komplext. Vi var ovana med designmönstren i Android: det fanns på ytan inget traditionellt MVC-mönster att rätta sig efter. Activities är på ett sätt en View och Controller i ett. Till råga på det fanns det beroenden på inbyggda systemklasser (exempelvis Application, Context, Activity, m.fl.) i Android, vilka vi till en början inte visste hur vi skulle hantera. Android är svårtestat, helt enkelt.

Under projektets senare del fokuserade vi alltmer på automatiska tester. Ett flertal framsteg har gjorts: vi har nu lärt oss mycket mer om att testa databasen och Content Provider, men även det grafiska gränssnittet, vilket har gjorts med testramverket Robotium[1]. Robotium gör det möjligt att genomföra handlingar i gränssnittet, men genom direktiv från kod istället för manuella tester i simulatorn.

I efterhand tycker vi att det hade varit skönare och säkrare med fler automatiska tester tidigare i kodandet. Vi har många gånger manuellt testat applikationen efter att ha lagt till ny kod, bara för att säkerhetsställa funktionaliteten i systemet. Med fler automatiska tester av koden skulle mindre manuellt arbete behövts gjorts, vilket är alltid är skönt. Saker såsom testandet av specifika typer av inmatning- och returdata är idealiskt för automatiska tester, och inget vi skulle ha behövt testat manuellt varje gång. Det som gjorde att vi kom igång en aning senare med tester var att Android har ett eget sätt att göra saker på – det var inte helt enkelt att komma igång och förstå. Ovanligt nog fanns väldigt få kodexempel att utgå ifrån, när man exempelvis vill testa en Activity. Att utveckling för Android var helt ny för oss är också en faktor, eftersom vi har lagt mycket tid på att förstå API:erna, och på att bygga bra applikationskod enligt konventionerna.

Asynkron hämtning av data från IMDb

Några Stories inbegrep dynamisk nedladdning av filmdata från IMDb, genom ett öppet API[2]. Att skapa själva bryggan till API:t var inte svårt (simpla hjälpklasser med HTTP-anrop), men att integrera det på ett smidigt sätt och på rätt ställe i applikationen var mer komplicerat. Då applikationen är för mobiler ställs högre krav på hantering av resurser och prestanda, på grund av begränsad datorkraft och internetuppkoppling. Därför *måste* all hämtning av data över internet ske asynkront i bakgrunden på en

[1] Robotium: http://code.google.com/p/robotium/

[2] TMDb: http://api.themoviedb.org/2.1/

egen tråd, separerad från UI-tråden. I ett tidigt skede byggde vi upp all kommunikation med API:t i egna klasser som ärvde Androids AsyncTask, som just är lämpad för asynkront arbete i bakgrunden. Lösningen fungerade till en början, men ett antal problem uppstod: allt från prestandarelaterade problem till att för mycket kod var tvungen att ligga i Activityn.

Därför utnyttjade vi en annan metod som bygger på att man implementerar en egen adapterklass, och i denna arbetar med klassen Filter för att utföra arbetet. Även här körs vissa metoder på en "worker thread", alltså inte på UI-tråden. Denna lösning fungerade bättre, då det tillät oss att lyfta ut mer kod från våra Activities, och därmed få klasser som utför smalare uppgifter.

Databasen och Content Provider

En stor fördel med att utveckla för android är att SQLite finns inbyggt i operativsystemet. Detta innebär att du själv inte behöver tänka på att ha en databasserver att kommunicera med, uppsättningen av en databas är väldigt enkelt och det går snabbt att sätta sig in i även om man inte gjort det förut. Metoderna som finns för att kommunicera med databasen är lättanvända för personer som inte kan någon SQL-syntax, men det finns även metoder som möjliggör rå SQL-kod vilket krävs för mer komplicerade funktioner, som JOIN till exempelvis.

Applikationens första iteration använde sig bara av en samling hjälpfunktioner som kommunicerade direkt med databasen. Det fungerade helt upp till förväntan, men efter att ha läst på mer om databashantering i Android insåg vi att användande av Content Provider krävdes för att andra applikationer skulle kunna få tillgång till vår data. Syftet med en Content Provider är alltså att det gör applikationen mer utbyggbar, och även lättare att testa.

Utmaningen med att använda en Content Provider är att det är svårt att förstå sig på hur metodanrop till den fungerar. Applikationen får tillgång till data genom en Content Resolver-klient. Klienten har metoder med identiska namn som providern. I metoderna skickar man med en URI som i praktiken innebär vilken tabell i databasen som ska påverkas. Det är svårt att förstå sig på hur Content Resolver fungerar, och man har ingen kontroll själv på vad som händer under tiden från att man kallar på den tills att metoden i Content Provider tar emot anropet från den. Ett antal oväntade beteenden uppstod under arbetets gång, exempelvis icke förväntade returvärden, som man upptäckte först genom djupare debugging.

Val av presentation av listdata

I den första iterationen av applikationen använde vi oss av en simpel implementation av en standardklass i Android, en så kallad ArrayAdapter, för att presentera filmer från databasen som en lista (se branch "`v0.1`", fil: "`se.chalmers.watchme.activity.MainActivity`", rad: `31`). Snart byggde vi en egen version av denna, MovieItemAdapter, som inte bara visade en films titel utan också datum och betyg (se branch "`sprint3`", commit: `b0b4f3646d6e16a22cd56ccee1c8e40ec92e34a8`). Under tiden som vi implementerade en Content Provider i systemet såg vi dock fördelarna med att använda tekniken SimpleCursorAdapter tillsammans med en CursorLoader istället. Fördelarna är att man binder kolumner från databasen direkt till områden i vyn, till exempel ett textfält. CursorLoadern bestämmer vilken Cursor som ska användas, vilket betyder att många databasfrågor körs i bakgrunden automatiskt, och

sedan presenteras till vyn utan mycket egenskriven kod. MovieItemAdaptern byttes således ut mot en SimpleCursorAdapter.

Nackdelen med att använda en CursorAdapter istället för en ArrayAdapter är att en utvecklare som kommer utifrån inte lika lättöverskådligt kan se i koden vad det är som presenteras. När man definierar en ArrayAdapter säger man vilket objekt som ska presenteras, vilket ger väldigt lättläslig kod. Användande av Cursors kräver en aning större förståelse för hur databasen och koden omkring är uppbyggd.

Dock är vi nöjda över valet, då det gav oss en djupare insikt om hur man kan arbeta med data i Android, utan att nödvändigtvis hantera informationen konkret genom exempelvis genom råa SQL-frågor och lågnivåanrop.

Vid ett tillfälle påbörjades även byggandet av en egen SimpleCursorAdapter, TagMovieListAdapter (se branch "`tag_movie_list`", commit: `40f19b6b5f8aadf19967fc34f83eea2d5e23a9c6`). Detta var när en lista utav filmer som innehöll en specifik tag skulle implementeras. Tanken var att SimpleCursorAdaptern skulle innehålla en lista med alla filmer, men endast visa upp de som innehöll en specifik tag. Metoden visade sig dock vara osmidig. Bättre var att databasen skötte filtrerandet och gav en lista med filmerna som innehöll en specifik tag till SimpleCursorAdaptern som sedan visade upp denna.

Oväntade beteenden hos Cursor

Det är inte tillåtet att lägga till samma film två gånger i applikationen. I Content Providers metod för att lägga till filmer gör vi därför en databasfråga för att se att titeln inte redan finns med. Om det inte går att flytta till första raden i Cursorn (praktiskt sett, det finns ingen rad i cursorn alltså ingen film med den titeln) så sätter vi in filmen. Då vi gjorde den här kollen fick vi en bugg där filmlistan inte uppdaterades, då man la till eller tog bort filmer, förrän man bytte activity. Av en slump märkte vi att listan uppdaterades som den skulle om vi innan kollen (går det att flytta till första raden) kallade på getCount hos cursorn.

Ett annat oväntat beteende hos cursor är att vi i vissa fall då vi borde få en cursor med 0 rader istället har fått en cursor med 1 rad där värdena i kolumnerna är null. Det här sker endast då vi använder oss av android.database.cursor. Skriver vi samma SQL-kommando direkt i adb får vi tillbaka precis som önskat, en cursor med 0 rader.

Beteenden som dessa har krävt mycket extra (onödig) tid hos oss, men är också något som lärt oss att allt inte alltid fungerar som man tror. Detta har gett oss träning i felsökning.

Emulatorn

Något som krävt mycket tid vid utvecklandet av applikationen är manuell testning. I vårt fall har vi använt programvara i Androids SDK som emulerar en Android-telefon. Emulatorn tar ofta lång tid att starta, och man måste ofta försöka ett godtyckligt antal gånger innan den faktiskt fungerar. Resultatet kan bli att man sitter i 10-15 minuter och försöker starta emulatorn innan man kan testa koden man skrivit. Detta är

oerhört ineffektivt och irriterande.

En lösning som vi tror hade effektiviserat vårt arbete mycket i detta avseende hade varit om alla hade varsin fysisk Androidtelefon att köra applikationen på. Vi hade då fått en tydligare bild av applikationen ur prestanda- och användarperspektiv, då emulatorn är långsammare och inte lika responsiv som ett riktigt Androidsystem.

Angående kommentarer

I de betygskriterier som satts för kursen finns en post som säger att andelen kommentarer i koden ska vara 40 procent för att maximal poäng ska erhållas. Vi har emellertid medvetet valt att lägga oss på en lägre nivå. Detta beror på att vi anser att vi hade behövt lägga till överflödiga kommentarer för att uppnå 40 procent. Kommentarer är till för att hjälpa någon annan att förstå den kod man har skrivit. Det optimala är dock om koden är förklarande i sig själv – den bästa koden är den som dokumenterar sig själv. Om detta är fallet blir kommentarer snarare ett irritationsmoment och något som stör mer än något som hjälper. På de ställen där vi anser att vår kod beskriver sig själv på ett bra sätt har vi därför avstått från att lägga till kommentarer.

Target/Required SDK

Vi har valt att sätta både target och required SDK som 16 för vår applikation. Eftersom många Androidenheter på marknaden är äldre och inte klarar av detta kan man ifrågasätta detta beslut.

Anledningen till att vi har gjort så är helt enkelt att vi i början av projektet inte hade kunskap om Android-systemens versionsnummer och utbredning. Ingen i gruppen äger en Androidenhet, och när vi började implementera vårt program var vår uppfattning att de flesta enheter borde stödja det senaste API:t som finns tillgängligt. Vi antog också att om så inte var fallet så borde de standardklasser och metoder som finns tillgängliga vara bakåtkompatibla.

Senare i projektet blev vi varseblivna om att vår applikation kanske inte skulle fungera på många enheter eftersom vi använde ett såpass nytt SDK. Vi hade emellertid då kommit så långt i implementationen att vi valde att inte åtgärda detta. En för stor del av programmet hade behövt byggas om för att det skulle vara värt det. Vill man köra applikationen får man helt enkelt köra den senaste versionen av operativsystemet.