Institutionen för datavetenskap
Department of Computer and Information Science

Examensarbete

En undersökning i riskhantering för vidareutveckling av redan existerande projekt

av

Jon Dybeck, Fredrik Präntare, Marcus Jonsson, Mattias Lantz Cronqvist, Anders Söderström, Carl Einarson, Oscar Nöjdh, Martin Bodin

LIU-IDA

2014-05-23

Linköpings universitet

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Sammanfattning

Digimergo är en digitalisering av Emergo Train System, ett system där personal inom räddningstjänst kan öva på olika katastrofscenarion. För att göra Digimergo användbart behövdes ytterligare programvara: ett administrationsverktyg till övningar och en scenarioeditor. I det programvaruutvecklingsprojekt som denna rapport behandlar har ny programvara utvecklats och integrerats med det ursprungliga Digimergosystemet.

I den här rapporten diskuteras vilka risker som existerar när ny funktionalitet skall läggas till till ett gammalt projekt samt hur dessa risker kan minimeras. Rapporten undersöker också vilka utvecklingsmetoder som lämpar sig i projekt där ny funktionalitet ska läggas till befintliga system.

Resultatet visar att den största risken med att utöka befintliga projekt är att underskatta tiden som krävs för att sätta sig in i projektet i fråga. Det mest effektiva sättet att minimera risken för detta är att mycket tidigt studera det tidigare arbetet och utbilda projektmedlemmarna i det gamla systemet. Ytterligare ett angreppsätt är att välja en metod som är flexibel när det kommer till nya risker eller ändringar i projektets plan, förslagsvis iterativa metoder.

Innehållsförteckning

1 Inledning

Digimergo är en digitalisering av katastrofövningssystemet Emergo Train System. Emergo Train System används i nästan hela Sverige och även i andra delar av världen. I detta programvaruutvecklingsprojekt har ett antal delsystem utvecklats till Digimergo.

Projektet genomfördes i samarbete med Linköpings universitet och Katastrofmedicinskt centrum i Linköping. Projektgruppen bestod av åtta teknologer från civilingenjörsprogrammet inom datateknik på Linköpings universitet.

1.1 Definitioner och begrepp

I det här avsnittet följer de definitioner och begrepp som används i rapporten.

Förkortning	Begrepp	Definition
	Digimergo	Digital version av ETS.
DEC	Digimergo Exercise Client	Programmet som spelar upp ett scenario för användaren.
DEM	Digimergo Exercise Manager	Verktyg som används under körning av ett scenario.
DRV	Digimergo Result Visualizer	Verktyg för att visa statistik över ett kört scenario.
DS	Digimergo Server	Huvudsystemet som skickar informationen till alla inkopplade DEC.
DSE	Digimergo Scenario Editor	Program för att skapa scenariofiler.
ETS	Emergo Train System	Katastrofövningssystem.
KMC	Katastrofmedicinskt centrum	Självständig produktionsenhet direkt underställd Landstinget i Östergötland och Linköpings universitet.
MVC	Model View Controller	Designmönster som abstraherar bort det visuella från den bakomliggande logiken.
MVVM	Model View ViewModel	Designmönster som abstraherar bort det visuella från den bakomliggande logiken med stöd för datakoppling.
TDD	Test-driven development	Utvecklingsmetod som bygger på mycket korta iterationer. Testfall skrivs först, därefter metoden.
WCF	Windows Communication Foundation	Ramverk för serviceorienterade applikationer.
WPF	Windows Presentation Foundation	System för att rendera användargränssnitt i Windowsapplikationer.
XAML	Extensible Application Markup Language	Ett deklarativt XML-baserat språk.

1.2 Motivering

De problem som existerar för projektets genomförande är flera och grundar sig främst i att ett existerande system skall byggas ut. Att integrera ett nytt system med ett redan existerande ger upphov till en rad frågor som måste besvaras för att projektet skall lyckas. I denna rapport studeras detta ur ett projektspecifikt perspektiv, det vill säga utifrån utbyggnaden av Digimergo. Paralleller

kan dock dras till andra programvaruutvecklingsprojekt, där existerande system ska integreras med andra eller byggas ut. Då denna typ av projekt är vanliga är det också viktigt att undersöka vilka problem och fallgropar som kan stötas på [1].

1.3 Syfte

Syftet med denna rapport är att förbättra förståelsen kring vilka risker och problem som kan uppstå i projekt som bygger på redan existerande system, samt att undersöka hur dessa risker kan undvikas både innan och under projektets gång. Syftet är också att rapporten ska identifiera de mjukvaruutvecklingsmetoder som är bäst lämpade för sådana projekt.

1.4 Frågeställningar

De frågeställningar som rapporten besvarar är följande:

- Vilka risker uppstår när ett redan existerande projekt skall byggas ut, och hur kan de hanteras?
- Vilken utvecklingsmetod är lämpligast att använda när ett projekt bygger vidare på ett tidigare system?

1.5 Avgränsningar

I projektet ligger fokus till största delen på att utveckla de två huvudprogrammen Digimergo Scenario Editor (DSE) och Digimergo Exercise Manager (DEM) och inte på att förbättra det nuvarande systemet, utöver de fall där det är absolut nödvändigt. Utvecklingen av Digimergo Result Visualizer (DRV) har nedprioriterats, då det var oklart om delsystemet kunde utvecklas inom tidsramarna för projektet.

1.6 Affärsplan

En affärsplan har skapats då detta var ett krav i kursen, se appendix A.

1.7 Erfarenheter

En separat bilaga innehållandes gruppens tekniska- och process-erfarenheter finns, se appendix B.

2 Teori

I det här avsnittet beskrivs, för rapporten, viktiga begrepp och teoretiskt material.

2.1 Delfimetoden

Delfimetoden går ut på att en panel med experter får ett antal frågor. Varje expert ska individuellt besvara frågorna så noggrant de kan, för att sedan diskutera och gemensamt hitta ett rimligt medelvärde av sina svar. Varje fråga behandlas i iterationer, där varje iteration börjar med att experten anonymt bedömer svaret på frågan. Sedan sammanställs resultatet av alla deltagares svar. Resultatet diskuteras och deltagarna får motivera sina svar, med målet att nå en överenskommelse om vilket som stämmer bäst. Sedan börjar de om med nästa fråga tills alla frågor är bearbetade. Delfimetoden är ett bra sätt att approximera exempelvis hur lång tid det kommer att ta att utföra vissa uppgifter. Då alla deltagare antagligen säger olika tider och kommer på olika problem som kan

uppstå med uppgiften kan deltagarna gemensamt komma fram till hur lång tid uppgiften kommer att ta. [2]

2.2 Vattenfallsmodellen

Vattenfallsmodellen är en systemutvecklingsprocess som ofta används inom mjukvaruutveckling, där varje framsteg ses som ett flöde nedåt (som ett vattenfall). Poängen med modellen är att bli klar med varje steg innan nästa steg påbörjas vilket gör det till en väldigt tydlig modell att arbeta med.

Vattenfallsmodellen lämpar sig väl för kortare projekt såväl som projekt där kravspecifikationen inte kommer ändras under projektets gång. Det är då bra med ett sekvensiellt tillvägagångssätt[3].

2.3 Trac

Trac är ett webbaserat projekthantering- och buggrapporteringssystem. Systemet underlättar dokumentation genom att projektets olika komponenter kan tilldelas egna wiki-sidor där relevant informationsfångst kan antecknas. Det finns ett flertal olika moduler som kan hjälpa till med mötesbokningar, Git- och SVN-hantering. Det finns även många externa plugins att installera för att utöka funktionaliteten. [4]

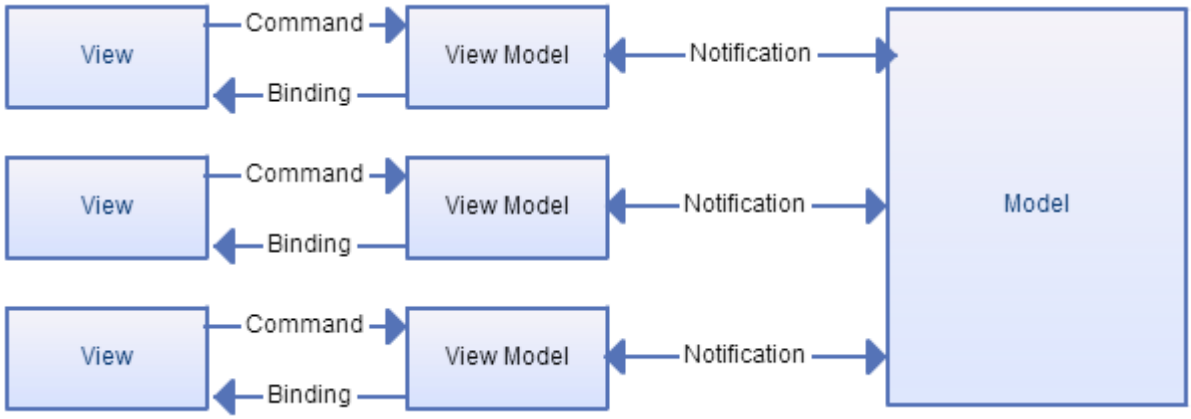
2.4 Emergo Train System

Emergo Train System (ETS) är ett system som används för att träna räddningspersonal vid katastrofscenarion. De som använder ETS får som uppgift att hantera ett katastrofscenario, där olycksplatser visualiseras med hjälp av ett antal whiteboardtavlor. Olika aktörer, som till exempel patienter, sjukvårdspersonal, sjuktransport och räddningsarbetare av olika slag representeras av plastfigurer med en magnet på baksidan [5]. Det finns också ofta tavlor som representerar sjukhus.

Olika delar av katastrofhanteringsprocessen kan övas med hjälp av ETS. Till exempel kan ambulanspersonal träna prioritering av skadade personer och katastrofplatskommunikation med räddningsledare. Prioritering av patienter sker genom triage. Gröna, gula, röda eller svarta klisterlappar placeras på patienter beroende på hur allvarligt skadade dessa är [6]. Instruktören styr övningen och kan under övningens gång ändra i scenariot. Ändringarna kan motsvara de yttre faktorer som påverkar räddningsarbete i en riktig katastrofsituation. Till exempel kan instruktören under övningen placera ut fler patienter, vilket i en riktig katastrofsituation kan motsvara att ett område blivit säkrat och tillgängligt för räddningspersonal. Räddningskommunikation i ett helhetsperspektiv övas under hela övningens gång.

2.5 Model View ViewModel

Model View ViewModel (MVVM) är ett designmönster utformat för att underlätta utvecklingen av grafiska system som använder databindning. MVVM kan förenkla underhåll av system eftersom beräkningar och liknande flyttas bort från vyn och in i modellen vilket tillåter bättre testning. MVVM är likt Model View Controller (MVC) men är mer anpassat för system inom Windows Presentation Foundation (WPF). Se Figur 1. [7]



Figur 1: Visualisering av det generella designmönstret "MVVM".

2.6 Systembeskrivning Digimergo

Innan projektets start bestod Digimergo av två delar: Digimergo Server (DS) och Digimergo Exercise Client (DEC). Dessa två program var översiktligt dokumenterade i orginaldokumentationen, dokumenten beskrev DEC i detalj men DS beskrevs endast kort.

Övningsklienten visades på stora tryckskärmar och skötte all användarinteraktion under övningarna. Där kunde användarna flytta runt aktörer, behandla patienter, para ihop patienter med sjukvårdare eller ambulanser med mera. All användarinteraktion tolkades av övningsklienterna och skickades till servern.

I servern fanns en modell som representerade övningen och vad som skulle visas på klienterna. När en användares gester tolkats i klienten skickades dessa till servern för att modifiera modellen. Servern utförde då beräkningar och kontroller, och om dessa blev godkända skickades resultatet tillbaka till klienten. På så sätt utfördes få beräkningar i klienten.

Vid större övningar kunde även flera klienter kopplas mot samma server. Klienterna jobbade mot en gemensam modell i servern, vilket gjorde att ändringar på en klient även visades i de andra.

Innan starten av projektet som den här rapporten berör hade Digimergo ungefär samma funktionalitet som Emergo Train System (ETS). Fördelarna med att ha ett digitalt system, såsom att enkelt kunna skapa olika scenarion och att snabbt ändra i ett pågående scenario utnyttjades dock inte, och systemet var därför inte redo att tas i bruk. Servern loggade alla ändringar som gjorts, men det saknades en effektivt metod för instruktörerna att använda loggen för att utvärdera övningarna.

Klienten kunde användas i två lägen. I editor-läget kunde nya aktörer (patienter, personal och andra resurser) skapas från en panel vid sidan av programmet. Detta läge passade inte för övning, eftersom

övningsdeltagarna kunde skapa precis vad de behöver för att klara utmaningen. I övningsläget laddades automatiskt ett färdigt scenario in. Övningsläget fungerade, med begränsningen att det var bara det fördefinerade scenariot som kunde köras.

I det här projektet utvecklades två verktyg till det existerande systemet. DEM utvecklades som ett verktyg med vilket instruktörerna kunde styra över och övervaka övningar, något som tidigare saknats. För att göra det möjligt att skapa nya scenarion att använda i DEC's övningsläge utvecklades DSE.

2.6.1 Digimergo Server

DS innehåller en modell som beskriver hur övningen ser ut. Denna modell kan sedan modifieras på olika sätt. Direkt interaktion i DEC kan påverka de olika aktörerna i modellens vyer. Genom interaktion via DEM kan instruktörer göra ändringar som förändrar läget i övningen. DS använder Windows Communication Foundation (WCF) för kommunikation med DEC och DEM.

2.6.2 Digimergo Excercise Client

DEC är den enhet där de som utbildar sig kan interagera med systemet. Under en övning kan flera klienter var uppkopplade mot servern samtidigt. I DEC kan de som utbildas flytta runt aktörer och behandla patienter för att lösa uppgiften i övningen. DEC körs på en PC med multitouch-skärm. På klienterna avbildas den gemensamma modellen som ligger i servern. Ändringar i en klient påverkar därför alla klienter.

2.6.3 Digimergo Scenario Editor

DSE är ett fristående program i Digimergosystemet. Dess uppgift är att skapa en scenariofil (.esf) som sedan kan föras över till servern för att köras i en liveövning. DSE kan även skapa resurspaketfiler (.erf) vilka också används under övningar. Eftersom DSE är helt fristående från de andra delarna av systemet behövdes ingen hänsyn tas till det ursprungliga systemet under konstruktionen. I programmet placeras olycksplatser och sjukhus på en karta eller på en inladdad bild. I dessa platser kan aktörer sedan placeras ut. Events, tidsbaserade händelser som förändrar läget i scenariot under övningens gång, kan också skapas. Ett event kan till exempel vara att ett antal nya patienter dyker upp på en olycksplats eller att ett antal ambulanser når olycksplatsen. DSE är konstruerat i WPF med designmönstret MVVM.

2.6.4 Digimergo Excercise Manager

DEM konstruerades som ett verktyg för instruktörerna att styra över övningar. Med verktyget kan intruktören aktivera events, se statistik över övningen, ändra inställningar, göra anteckningar om resultatet av övningen med mera. När ett scenario ska startas kopplas DEM upp mot servern. Via DEM kan sedan en scenariofil och ett resurspaket laddas, som ska användas i övningen. Flera DEM enheter kan kopplas mot servern samtidigt för att flera instruktörer ska kunna övervaka övningen. Även DEM är konstruerad i WPF med designmönstret MVVM.

2.6.5 Avgränsningar

I det här projektet utvecklades DSE och DEM. Utvecklingen av DRV sköts upp tills det att de andra enheterna var färdiga. För att göra det äldre systemet kompatibelt med det nya behövde DS och DEC utvecklas. Det här projektets främsta fokus har varit utveckling av DSE och DEM. Därför har de

problem som existerade inom DS och DEC inte åtgärdats i någon större utsträckning. Ledorden under arbetet på DS var att inte ändra mer än nödvändigt.

3 Metod

I detta kapitel följer beskrivning av olika metodiker som har använts i projektet.

3.1 Utvecklingsmetod

Det här projektet har utförts enligt vattenfallsmodellen. Projektets första del bestod av en förstudie, följt av en utvecklingfas.

3.1.1 Förstudie

Under förstudien hölls möten med kunden för att ta fram en kravspecifikation som båda parter kände sig nöjd med. Utkast till en hel del artefakter skapades såsom projektplan, gruppkontrakt, arkitekturbeskrivning och kvalitetsplan. En tidsplan skapades även och med hjälp av denna gjordes en uppskattning om kraven var möjliga att utföra inom de givna tidsramarna. Tidsplanen gjordes genom att hela projektgruppen satt ner tillsammans och försökte dela upp kraven i mindre aktiviteter. För att göra en tidsuppskattning på dessa aktiviteter användes Delfimetoden. Som hjälp för att organisera aktiviteterna och för att tidsrapportera användes Trac.

Efter att kraven och aktiviteter var bestämda togs två olika prototyper fram, en för DEM och en för DSE. Dessa gjordes på två olika sätt, DEM-prototypen skrevs i kod och DSE-prototypen skapades med hjälp av bilder i powerpoint.

Det existerande systemet testades genom att testkörningar utfördes i laboratoriet. Efter testningen studerade samtliga medlemmar även källkoden. I samband med detta skedde också utbildning i syntaxen för C# samt versionshantering med Git.

En riskplan skrevs där risker graderades efter sannolikhet att de inträffar. Riskerna baserades på erfarenheter från tidigare projekt. Planen uppdaterades sedan två gånger under projektets gång, i takt med att riskerna ändrades.

3.1.2 Utvecklingsfas

Under utvecklingsfasen delades projektgruppen upp i tre mindre grupper, en grupp för varje delsystem. Fyra personer tillsattes för att utveckla DSE, två personer fortsatte med utveckling på det tidigare systemet och resterande två utvecklade DEM. Alla tre grupperna jobbade efter tidsplanen som tagits fram i förstudien.

Kommunikationen mellan grupperna bestod av minst ett möte i veckan för att stämma av med de andra och se till att projektet fortskred enligt tidsplanen. Dessa möten hölls på måndagar och varade i en timme. Innehållet var oftast en statusrapport på samtliga system, för att hålla hela projektgruppen uppdaterad. Om en grupp hade fastnat med någonting eller hade andra problem togs även detta upp.

De artefakter som skrivits under förstudien bearbetades även under denna fas för att hållas uppdaterade.

3.2 Designmetod

Utvecklingsmiljön som användes under projektet bestod av Microsoft Visual Studio 2013. I detta program fanns Git integrerat, vilket användes för att versionshantera koden. Programmeringsspråket som användes var C# med .NET Framework 4.0.

3.2.1 Användarstudie

Prototyperna var utformade så att testanvändare kunde ges en eller flera specifika uppgifter att genomföra. Som förberedelse inför testet formulerades tre olika uppgifter till båda prototyperna, vilka varje testanvändare prototypen testades på skulle få utföra.

Testerna gick till genom att en uppgift visades för testanvändaren, sedan fick denne frågan; “Hur svår tror du denna uppgift är att utföra i prototypen på en skala från ett till sju?”. Efter att uppgiften var utförd fick testanvändaren frågan; “Hur svårt var det att utföra uppgiften på en skala från ett till sju?”. Under testerna räknade sedan testhandledaren antal interaktionssteg i gränssnittet för att utföra uppgiften. På detta sätt bildades en uppfattning om hur användarvänlig prototypen var. Testanvändaren tillfrågades även under pågående uppgifter hur denne tänkte i olika situationer, vilket också hjälpte till att utvärdera prototypen.

När utseendet på DSE och DEM bestämdes användes prototyperna som skapats i förstudien som grund. DSE-prototypen användes som en mall för DSE och DEM byggdes vidare från DEM-prototypen.

3.3 Testmetod

Testningen i detta projekt är utförd enligt en Master Test Plan (MTP) som har skrivits enligt IEEE 829, en Level Test Report (LTR) samt en Master Test Report (MTR) [8]. Testplanen beskriver hur testning ska gå till samt vilka tester ett system behöver klara för att bli godkänt. Testerna relaterar hårt till kraven för att ingen funktionalitet ska saknas i slutprodukten. Det finns tre nivåer av testning, enhet-, system/integration- och acceptanstestning. Enhetstestningen består av tester som bara relaterar till en viss enhet, integrationstestningen av integrationen mellan flera enheter och att dessa fungerar på ett korrekt sett och acceptanstestningen beskriver om systemet är leveransdugligt.

Level Test Report utförs flera gånger under projektets gång och beskriver vilka tester som hunnit utföras, vilka tester som har klarats av och hur testningen ska fortskrida till nästa rapport. I den kan det tidigt utläsas om något går mindre bra, vilket kan tyda på att en omprioritering kan behövas. Master Test Report beskriver hur det har gått med testningen under projektets gång. Den listar upp vilka tester som gjorts, vilka som är avklarade samt innehåller testledarens rekommendation om systemet är leveransdugligt eller inte.

I början av projektet testades det redan existerande systemet för att avgöra hur stabilt det var. Kodinspektioner utfördes för att bilda en snabb uppfattning om dess uppbyggnad. Testplanen skapades också tidigt för att hålla kvalitén hög genom hela projektet. Under projektets gång testades enheter som började bli klara för att verifiera funktionalitet och hitta buggar i systemet, och anteckningar gjordes i den aktuella Level Test Report. I slutet av projektet testades hela systemet tillsammans för att verifiera att systemet var leveransdugligt och att alla krav var uppfyllda.

3.4 Forskningsmetod

Under projektets gång samlades erfarenheter på främst tre olika sätt, dels personliga kommentarer lagrade i trac, induviduella dokument och slutligen gemensamma diskussioner vid möten. Under den mycket av projekttiden fokuserade gruppen på att endast samla in erfarenheter utan att reflektera över eller diskutera dessa. Först vid projektslut sammanställdes erfarenheterna.

4 Resultat

Här listas resultaten som har samlats in under projektets arbete enligt metoderna i föregående kapitel.

4.1 Utvecklingsprocess

Nedan följer resultaten från utvecklingsprocessen.

4.1.1 Risker

Några av de befarade riskerna inträffade under projektet. 5 av medlemmarna blev sjuka någon gång under projektet vilket ledde till att de missade möten och fick jobba hemifrån. Aktiviteter kopplade till DS blev försenade, på grund av att den existerande kodbasen för DS var betydligt mer komplicerad än förväntat. Fördelning av arbetade timmar var ojämn efter halva projektet. Eftersom systemen var olika långt utvecklade var det svårare att fördela resurserna på det kommande arbetet.

4.1.2 Tidsplanering

Uppskattningen från Delfiundersökningen överskred den totala tidsbudgeten på 2000 arbetstimmar, vilket resulterade i nedskärningar av delsystem och krav. Delsystemet DSV och krav på 3D-funktioner i DSE togs bort. Beslut togs om att endast lägga till stöd i DS för hantering av de nya scenario-filerna och kommunikation med DEM. Inga förbättringar av den nuvarande koden i DS lades till i tidsplaneringen.

4.2 Designmetod

Resultatet från användarstudierna visade att alla förbestämda uppgifter går att utföra under 4 interaktionssteg. Följande koncept i prototypen ansågs enligt testanvändarna vara svåra.

- Navigering i kartvyn.
- Hur markörer placeras ut på kartan.
- Navigering bort från en olycksplatsvy.

4.3 Testmetodik

Nedan följer resultatet av testningen.

4.3.1 Existerande system

Testerna av det redan existerande systemet i början visade att det fanns många kodproblem och buggar, något som innebar att användaren tilläts göra saker som egentligen inte skulle gå att göra. Koden var dåligt skriven och det var svårt att se om någon kodstandard hade följts när den skrevs. Först och främst var kodbasen inte abstraherad eller uppdelad, istället låg nästan all funktionalitet i delade kodfiler. Men det beskrevs inte i dokumentationen eller kommentarerna hur denna delade funktionalitet fungerade. Det var även ett stort problem att de nätverksprotokoll som användes ej

var tilllräckligt dokumenterade eller kommenterade. Detta gjorde att det var svårt att förstå systemet och vidareutveckla koden. Slutligen fanns det inte heller några testfall dokumenterade eller implementerade.

4.3.2 Utvecklade system

Här presenteras resultaten av testerna på de system som utvecklats i detta projekt.

4.3.2.1 Scenario Editor

Testerna visade att alla förstahandskrav samt ett andrahandskrav uppfylldes av DSE.

4.3.2.2 Excercise Manager

Testerna visade att alla förstahandskrav uppfylldes av DEM.

5 Diskussion

Nedan följer diskussion av metod, resultat och arbetet i ett vidare sammanhang.

5.1 Metod

Vattenfallsmodellen som användes ansågs vara en bra projektmodell för utvecklingen av detta system då den är effektiv för kortvariga projekt och för nya system [3]. Alltså passade vattenfallsmetoden väldigt bra för delsystemen DSE och DEM, men mindre bra för utökning av det existerande systemet. Anledningen till detta är att när ett existerande system byggs ut är alla risker inte självklara från början. Med nya system som DSE och DEM blir det mycket enklare att hitta risker och planera arbetet, då risker kan identifieras och hanteras under konstruktionen. Det är också mycket enklare att designa ett nytt system jämfört med att anpassa till ett befintligt system. Om ett system utvecklas från grunden kan det designas fritt och hänsyn behöver oftast inte tas till annan kod. Om ett system ska utvecklas och behöver anpassas till ett befintligt system behöver tid först spenderas på att läsa och förstå koden till det befintliga för att kunna veta hur det nya systemet ska utformas. Detta kan ta mycket tid då det inte alltid är helt lätt att läsa och förstå kod som andra har skrivit. Om koden även är dåligt skriven/strukturerad tar det ännu längre tid. Det kan också vara svårt att veta om systemet kommer klara av de ändringar som ska genomföras innan de faktiskt är implementerade och riskerna kan förstås.

I detta projekt användes Delfimetoden för tidsuppskattning av aktiviteter. Det visade sig dock vara svårt när det saknades kunskap om hur det existerande systemet var uppbyggt. Hade mer tid lagts i början av projektet på testning och på att gå igenom gammal kod så hade rimligare aktiviteter tagits fram och tiden kunnat uppskattas bättre.

Delfimetoden kan vara svår att använda på existerande projekt eftersom att det kan bli mycket gissningar om hur det tidigare systemet fungerade. Om det är oklart exakt hur det existerande systemet är uppbyggt kan tidsuppskattningen bli osäker. Det kan leda till många omplaneringar och nerskärningar av krav i ett senare skede i projektet då det kan inses att vissa aktiviteter tar orimligt lång tid att genomföra. Om istället ett helt nytt system ska byggas kan det utformas så att rimliga aktiviteter enklare kan tas fram.

En prototyp skapades i början av projektet för att undersöka hur DSE och DEM var tänkta att se ut. Detta för att snabba upp utvecklingen i början och fokusera på att få in all funktionalitet. Problemet med det arbetssättet var att kraven från kunden inte var klara när arbetetet på prototypen påbörjades. Efter ett tag när kraven började ändras var det inte lika självklart att börja om arbetet med en ny prototyplösning utan istället hamnade fokuset på att modifiera den nuvarande prototypen för att matcha kraven. För att komma undan det problemet hade det varit bättre om arbetet med att utveckla en prototyp påbörjats senare när alla kraven på systemet var färdiga.

För att testa att prototypen höll måttet utfördes tester på ett antal utvalda personer. Detta medförde att dåliga designbeslut kunde ändras i ett tidigt skede. Ett problem var att när kraven började ändras, och med dem prototypen, fanns ingen tid för nya användartester.

I användarstudien användes antalet interaktionssteg som mått på hur enkla prototyperna var att använda. Om en uppgift tar 3 klick att utföra bedöms det som en enklare uppgift än en som tar 4 klick, oavsett hur svårt det var för användaren att hitta vart den ska klicka någonstans. Detta kan ge en felaktig bedömning.

5.2 Resultat

Eftersom tidsplanen gick över tidsbudgeten togs delsystemet DSV bort. Istället för att ta bort ett helt delsystem hade funktionaliteten i de övriga delsystemen kunnat minskas. På så sätt hade kunden fått alla de delsystem som var tänkt från början, dock med mindre funktionalitet i varje enskild del.

5.3 Arbetet i ett vidare sammanhang

Det är viktigt att reflektera över hur systemet som utvecklats kommer att påverka användarna och om det kan påverka samhället i stort. Ur användarsynpunkt kan systemet användas för att träna räddningspersonal och sambandspersonal. De kan därmed bli säkrare i sina arbetsuppgifter vid stora komplicerade katastrofer eftersom de har fått chansen att öva på hanteringen av många skadade.

Men det finns även en risk när denna personal använder ett digitalt system för att öva beslut som kan vara skillnaden mellan liv och död för katastrofoffer, för till skillnad från övningar med människor kan digitala övningar leda till ett distanserande från patienterna och situationen. Detta är givetvis inte önskvärt för samhället i stort och för att undvika att detta sker föreslås det att kombinera användandet av systemet med övningar med människor.

6 Slutsatser

Den största risken i ett projekt som bygger ut ett existerande system är att underskatta tiden som krävs för att sätta sig in i och modifiera ett gammalt system. Detta kan leda till förseningar. En liknande risk är att kraven på det nya systemet kan bli felaktiga då de baseras på felaktiga antaganden om det gamla systemet.

För båda dessa risker är det mest effektiva motmedlet att mycket tidigt i projektets planeringsfas studera det gamla systemet i detalj och utbilda projektdeltagarna i hur det gamla systemet har konstruerats. Detta för att deltagarna skall vara väl insatta i systemet när de sedan diskuterar krav

och design av de nya delarna som skall läggas till. En sådan granskning visar också om det finns betydande problem med det tidigare systemet.

Ytterligare ett angreppsätt för att göra det enklare att hantera dessa risker är att välja en utvecklingsmodell som naturligt hanterar ändrade förutsättningar. Under projektets gång konstaterades det att vattenfallsmodellen fungerar väl för styrningen av utvecklingen av nya system. Dessvärre kan det uppstå problem i de delar som är nära kopplade med det tidigare systemet. För att göra det enklare att hantera dessa delar bör en iterativ utvecklingsmodell med korta iterationer väljas, för att alltid kunna övervaka och justera projektets riktning om problem uppstår.

Slutligen är det viktigt att projektledare i ett mycket tidigt skede undersöker hur ett projekt kommer att vara beroende av tidigare system. Projektledaren bör undersöka om systemet är väl dokumenterat och om koden har god struktur. Om det visar sig att systemet är bristfälligt på någon av dessa punkter bör projektledaren överväga en iterativ utvecklingsmodell och instruera sina deltagare att studera systemet noggrant.

För framtida studier föreslås det att fokusera mer på hur iterativa modeller kan användas i ett tidigt skede för att upptäcka risker med orginalsystemet. Exempelvis genom att använda korta iterationer där varje deltagare får i uppgift att studera en bit av systemet och sedan förklara hur denna del kan påverka projektet positivt eller negativt.

7 Individuell del

7.1 Jon Dybeck – Tidsplanering och Delfimetoden

7.1.1 Sammanfattning

I den här rapporten undersöks hur precis Delfimetoden är för att skapa en tidsplan för ett mjukvaruutvecklingsprojekt. Även de risker som finns med att använda Delfimetoden för detta ändamål undersöks och diskuteras. Undersökningen görs genomfördes genom att planera ett mjukvaruprojekt via Delfimetoden med åtta medlemmar i delfiundersökningen.

Slutsatsen av undersökningen är att stor vikt måste läggas på deltagarna i delfiundersökningen och att det är mycket viktigt att dessa är välutbildade på projektets domän.

7.1.2 Inledning

7.1.2.1 Motivering

Den största kostnaden i mjukvaruprojekt är oftast arbetstid hos programmerare och annan teknisk personal. Av den anledningen är det viktigt att projektets tidsplan har väldigt liten risk för stora fel. För om tidsplanen innehåller stora fel kan detta leda till långa förseningar och därav överskriden budget.

7.1.2.2 Syfte

Syftet med den här undersökningen är att utvärdera hur väl Delfimetoden fungerar för att ta fram precisa tidsplaner för mjukvaruutvecklingsprojekt.

7.1.2.3 Frågeställning

Hur kan Delfimetoden användas för att skapa en precis tidsplan för ett mjukvaruprojekt? Vilka potentiella risker finns med att använda Delfimetoden för att skapa en tidsplan?

7.1.2.4 Avgränsningar

Den här rapporten kommer inte att behandla risker som uppstår pga externa omständigheter, såsom sjukdom eller förändrade krav. Utan endast risker direkt relaterade till ett projekts tidsplan, så som felaktiga antaganden eller felinformerade deltagare.

7.1.3 Teori

Delfimetoden skapades under 1950-talet och fokuserar på att ta fram pålitliga förutsägelser [9] Metoden gör detta genom att låta ett antal experter svara på frågor som sedan sammanställs och resultatet skickas sedan tillbaka till experterna.

De får då möjlighet att diskutera och eventuellt ändra sina svar. Tanken är att denna process till slut leder till konsensus mellan experterna och att detta konsensus är nära verkligheten.

Sedan Delfimetoden skapades av den amerikanska militären har den använts inom många olika områden. [9] [10] På senare tid har metoden kommit att användas inom mjukvarutvecklings projekt för att kunna skapa relativt precisa uppskattningar inom risker och budget väldigt tidigt i ett projekt.

7.1.4 Metod

För att utföra undersökningen var det viktigt att samla in data från planeringstadiet av projektet. Detta gjordes genom att utföra en Delfiundersökning där samtliga deltagare var närvarande vid ett möte. Under mötet diskuterades det till konsensus nåddes hur mycket tid som varje aktivitet i projektplanen kunde tänkas ta. Sedan lagrades de slutgiltliga tidsuppskattningarna från Delfiundersökningen.

Under projektets gång användes trac för att lagra information om faktisk tidsåtgång för de olika aktiviteterna i projektet. Denna data jämfördes sedan mot de data som Delfiundersökningen gav i början av projektet.

7.1.5 Resultat

När den faktiska tidsåtgången av projektets aktiveter jämfördes med den uppskattade tidsåtgången som Delfiundersökningen givit stod det klart att undersökningen inte alltid motsvarade verkligheten. Det handlade om både mycket stora över och underskattningar av tidsåtgång för olika aktiveter. Om man däremot studerade hur mycket tid som använts sammantaget för exempelvis en applikation så var uppskattningarna bättre än för enskilda aktiviteter.

7.1.6 Diskussion

7.1.6.1 Metod

I efterhand så kan vi konstatera att det är viktigt att Delfiundersökningen utförs under en längre tid för att varje deltagare skall ha tid att fundera och komma med relevanta argument för att försvara sin position. Annars kan vissa deltagare ändra sin uppfattning endast för att andra är snabba med motargument.

7.1.6.2 Resultat

Anledningen till att delfiundersökningen gav så olika stora fel hos enskilda aktiviteter jämfört med applikationer kan ha orsakats av deltagarnas begränsade kunskaper inom området.

Studier om Delfimetoden har även visat att uppskattningar som ligger nära i tiden tenderar att vara optimistiska, vilket kan ha lett till ytterligare underskattningar. [10] Om deltagarna hade varit mer insatta i exakt vad som varje aktivitet innebar, exempelvis i vilken mån de behövde arbeta med det existerande systemet skulle uppskattningarna antagligen vara mer precisa.

7.1.6.3 Påverkan

Ur en företagsekonomisk vinkel är det viktigt att påpeka att domänkunskap är särskilt viktigt under planeringsfasen av ett projekt. Inte bara under projektets utvecklingsfas. För om de som är ansvariga för planeringen av ett projekt inte har god domänkunskap kommer tidsplanen att innehålla fel vilket kommer leda till att projektets budget inte följs.

7.1.7 Slutsats

Delfimetoden kan användas för att skapa tidsplaneringar för mjukvaruprojekt. Ett sett att göra detta är att låta deltagarna i undersökningen rösta för hur mycket tid kommer att ta. Det är dock viktigt att alla deltagarna får tid att överväga varandras argument. Det är även viktigt att undersökningen upprepas under projektets gång så att skiftande omständigheter tas upp i tidsplaneringen.

Den största risken med att använda Delfimetoden för att tidsplanera projekt kommer av att deltagarna måste ha god domänkunskap. Om deltagarna i undersökningen ej har denna kunskap kan detta leda till en planering som innehåller fel som inte uppdagas förrän projektet startat och redan påträffat felet.

7.2 Marcus Jonsson – Utvecklingsledare

7.2.1 Sammanfattning

En utvecklingsledare måste kunna leda ett team. När man leder ett team är det ytterst intressant att veta hur väl teamet presterar. Detta kan man göra genom att beräkna produktiviteten. Det finns flertalet sätt att göra detta och ett par av dessa beskrivs närmare i denna rapport. En utvecklingsledare har naturligtvis fler uppgifter än att bara leda ett team av utvecklare, vilka dessa uppgifter är undersöks också i denna rapport.

7.2.2 Inledning

I de flesta mjukvaruutvecklingsprojekt så finns det en utvecklingsledare i någon form. Denna rapport går in lite djupare på vad just denna roll innebär, samt hur man gör för att mäta produktiviteten hos ett team av utvecklare.

7.2.2.1 Motivering

En utvecklingsledare har oftast stort ansvar i projekt och eftersom jag har haft denna roll i detta projekt kändes det extra intressant att titta lite närmare på hur en utvecklingsledare arbetar i större projekt. Eftersom en utvecklingsledare leder ett team måste denne även kunna utvärdera hur väl teamet presterar. Detta kan man göra genom att beräkna produktiviteten.

7.2.2.2 Frågeställning

Rapporten kommer besvara följande frågor; Vilka är utvecklingsledarens uppgifter? Hur mäter man produktivitet hos ett team inom mjukvaruutveckling?

7.2.2.3 Avgränsningar

De utvecklingsledare rapporten behandlar kommer endast vara utvecklingsledare inom mjukvaruutvecklingsprojekt.

7.2.2.4 Definitioner och begrepp

Arbetsmånad: Om ett projekt tar 2 månader att genomföra för 3 personer, har det tagit 2*3 = 6 Arbetsmånader.

7.2.3 Teori

En utvecklingsledares uppgifter kan variera lite från företag till företag men den mest generella uppgiften är att leda ett team med programmerare. Utvecklingsledaren har ofta en bakgrund som programmerare och är även inblandad i kodandet även om det inte är lika mycket som de övriga i teamet. Utvecklingsledaren har ansvar för att teamet leverar en färdig produkt som uppfyller kraven och måste därför hålla sig kontinuerligt uppdaterad på hur det går för alla som är inblandade i kodandet. Detta innebär att utvecklingsledaren måste ha god kommunikation med alla och man kommer även vara den som får förklara de mer tekniska aspekterna i ett projekt för till exempel kunden.

Hur ska en utvecklingsledare då arbeta för att få sitt team att bli så produktivt som möjligt? För att kunna svara på den frågan måste man först definiera vad man menar med produktivitet i detta sammanhang. Det finns en mängd olika modeller för att mäta ett teams produktivitet, ett par exempel är [11]:

- produktivitet = (antal rader kod / 1000) / antalet Arbetsmånader
- produktivitet = Funktionspoäng / antalet Arbetsmånader

Den senare modellen använder sig av funktionspoäng, dessa beräknas genom att:

1. räkna inputs, outputs, inquiries, entities, interfaces och algoritmer.
2. Värdera svårighetsgraden av funktionerna i varje kategori på en femgradig skala.
3. Beräkna de totala viktade funktionspoängen genom att använda antalen av varje kategori, svårighetsgraderna, och de empiriskt bestämda viktningsfaktorerna.

Detta har visat sig ge en bra uppskattning av mjukvarans storlek, och därför kan man även använda dessa poäng för att beräkna produktiviteten [12].

7.2.4 Metod

För att besvara denna frågeställning så har information hämtats från relaterade jobbannonser där företag söker utvecklingsledare till mjukvaruutvecklingsprojekt år 2014. arbetsuppgifter som dessa jobbannonser har specificerat har sedan jämförts med varandra för att hitta liknelser. Ett flertal metoder och modeller som beskriver hur man mäter produktivitet har analyserats, de som har känts mest relevanta har sedan jämförts och diskuterats mer i detalj. För att testa den ena modellen har projektet i kursen TDDD77 använts för att beräkna produktiviteten. Modellen där man räknar antalet rader kod och delar med antalet arbetsmånader har då använts.

7.2.5 Resultat

De uppgifter som funnits vara mest generella för en utvecklingsledare är följande:

- Leda ett eller flera team av programmerare i ett eller flera mjukvaruprojekt.
- Ansvara för att mjukvaran blir klar i tid och att den uppfyller de krav som ställts.
- Förklara de mer tekniska delarna i mjukvaran för personer utanför utvecklingsteamet som inte har lika djup förståelse för programmet/systemet som utvecklas.
- Vara med och bestämma vilka tekniska lösningar som ska användas.
- Utveckla tester för att kunna producera så felfri kod som möjligt.

[13] [14], observera att dessa ej är fasta källor och kan försvinna med tiden.

För att mäta produktivitet kan man använda en mängd olika modeller. En som har visats ge en bra uppskattning på produktivitet är denna:

Produktivitet = Funktionspoäng / Arbetsmånader

I projektet i kursen TDDD77 så har 5358 antal rader kod producerats på 4 månader av 8 personer, vilket blir 32 arbetsmånader. Då den andra formeln som analyserats användes för att beräkna produktiviteten blev resultatet som följer:

Produktivitet = (5358 / 1000) / 32 = 0,1674357

Definitionen på produktivitet är “ratio of output units produced to the input units of effort” [11]. I detta fall blir det alltså förhållandet mellan rader kod producerade och antalet arbetsmånader spenderade.

7.2.6 Diskussion

Här diskuteras resultatet och metoden.

7.2.6.1 Metod

Att titta på jobbannonser för att ta reda på en utvecklingsledares arbetsuppgifter är ingen hållbar källa. Dessa uppgifter kan fort försvinna eller bli utdaterade. Anledningen till att denna metod valdes var för att det var svårt att hitta en annan bra källa för denna information. För att få en större bild av vilka olika sätt man kan använda sig av för att beräkna produktivitet hade man kunna jämfört fler metoder/modeller. Hade man haft mer tid så hade grundligare tester av dessa metoder kunnat utföras för att nå fram till ett bättre resultat.

7.2.6.2 Resultat

De uppgifter en utvecklingsledare har skiljer sig lite från företag till företag. Det finns dock en del uppgifter som fanns i de flesta jobb-beskrivningar och det är dessa som presenteras under resultatrubriken.

Under teori-rubriken tas det upp två sätt att beräkna produktivitet, man kan räkna antalet rader kod eller så kan man beräkna funktionspoäng och utifrån dessa beräkna produktiviteten. Det finns många fler modeller men i denna rapport behandlas bara dessa två. Problemet med att bara titta på antal

rader kod när man mäter produktivitet är att olika programmeringsspråk behöver olika många rader kod för att lösa samma problem och mer kod behöver nödvändigtvis inte betyda bättre kod [12].

Produktiviteten som beräknades med hjälp av att räkna rader kod som producerats i projektet i kursen TDDD77 behöver nödvändigtvis inte vara ett bra mått på grund av det som diskuterats i föregående stycke.

7.2.6.3 Arbetet i ett vidare sammanhang

Om en utvecklingsledare använder sig av en formel eller modell för att beräkna produktiviteten hos en anställd så är det inte säkert att det alltid ger det korrekta svaret. Att endast värdera en anställd utifrån denna siffra känns väldigt omotiverat och man måste nog väga in fler faktorer för att verkligen bedöma hur bra en anställd presterar. Det kan även kännas nedvärderande för en enskild anställd att endast bli bedömd utifrån en siffra. Om man däremot använder dessa modeller för att beräkna produktiviteten hos ett utvecklarteam som en enhet kan det ge en riktlinje på hur de anställdas sammanlagda produktivitet ligger till.

7.2.7 Slutsats

Det är svårt att säga exakt vilka arbetsuppgifter en utvecklingsledare har i ett projekt men man kan hitta en del generella som gäller i de flesta fall. Det finns väldigt många sätt att beräkna produktivitet på och i denna rapport nämns endast två av dessa. Det kan hända att det finns andra modeller som ger ett bättre mått på produktiviteten hos ett specifikt utvecklarteam.

7.3 Martin Bodin – Dokumentansvarig

7.3.1 Sammanfattning

Rapporten sammanfattar erfarenheter från dokumentansvarig under kandidatarbetet i programvaruutveckling. Den primära frågeställningen är, hur bör versionshantering hanteras i ett större programvaruutvecklingsprojekt för att undvika problem? Problem med versionshanteringen har antecknats under projektets genomförande. Slutsatsen är att majoriteten av problemen kan undvikas med tydligt definierade arbetsområden, god kommunikation, förståelse för utvecklingsverktygen och användning av versionshanteringssystemets undantagsfunktioner.

7.3.2 Inledning

I projektet har programvara till Digimergo utvecklats, en relativt stor uppgift med mycket kod och många komponenter att hålla reda på. För att göra uppgiften överkomlig har, som sig bör i all programvaruutveckling, ett versionshanteringssystem använts. Tanken med denna rapport är att belysa vikten av att använda versionshantering, och att visa strategier och metoder för att använda den problemfritt.

7.3.2.1 Definitioner och begrepp

Begrepp	Definition
VCS	Version Control System, system för versionshantering.
Repositorie	En central server dit alla ändringar synkroniseras
Synkronisering	Att hämta ändringar från ett centralt repositorie och skicka iväg sina egna

7.3.2.2 Motivering

Ett av de största problemen inom programvaruutveckling med fler än 1 utvecklare är: hur kan alla utvecklare jobba simultant, på olika eller samma problem, i projekt med många filer eller få filer eller när de befinner sig på olika platser. Kort sagt, hur kan flera utvecklare samsas om koden i ett projekt, utan att behöva hålla ständig kontakt, samtidigt som de undviker problem och dubbelt arbete? Hur kan utvecklare se sina ändringar och återskapa gamla versioner av sin kod? En lösning på dessa problem är versionshantering. Med versionshantering byggs en tidslinje av ”ögonblicksbilder” upp, en för varje ändring, där ändringar kan ångras, ändras och omformas vid behov. Med versionshantering kan utvecklare återställa sitt projekt till en tidigare punkt och slå ihop sina kodändringar med andras kodändringar. Det är dock inte alltid trivialt att använda versionshantering problemfritt, vilket motiverar en undersökning som denna.

7.3.2.3 Syfte

Syftet med denna rapport är att destillera de erfarenheter kandidatprojektet gav, och sammanfatta detta till konkreta påståenden och uppmaningar gällande smart och effektiv användning av versionshantering.

7.3.2.4 Frågeställning

Den specifika frågeställningen som rapporten ska besvara är följande: Hur bör versionshantering hanteras i ett större programvaruprojekt, för att minimera problem?

7.3.2.5 Avgränsningar

I rapporten görs ingen värdering av olika versionshanteringssystem eller vilket system som bör användas för någon given situation, då projektet från början var låst till att använda git per kundens krav. Rapporten berör därför inget specifikt versionshanteringssystem, istället ligger fokus på de generella principer som är gemensamma för dagens moderna VCS.

7.3.3 Teori

Nedan följer en teoretisk bakgrund till versionshantering.

7.3.3.1 Vad är VCS?

Ett VCS är ett system för att hantera ändringar i dokument, filer, program och i stort sett all data som kan lagras på ett datorsystem. Versionshantering tillämpas i många applikationer, exempelvis ordbehandlare och wiki-system, men den tillämpning som den här rapporten fokuserar på är hantering av programkod vid programvaruutveckling i team.

VCS är idag en standard för programvaruutveckling, något som projekt med många utvecklare inte kan klara sig utan, och är också kritiskt för ett projekts framgång.[15]

Olika VCS fungerar på olika sätt, och använder olika tekniker internt. Trots detta så delar de alla den grundläggande principen att revisioner av data sparas, ofta med tid, en kommentar och namnet på utvecklaren som utförde ändringen. En revision är inte en separat kopia av ursprunget, utan en ”ögonblicksbild” av de ändringar som utfördes.

7.3.3.2 Varianter

Många olika VCS existerar, och deras användning och utbredning kan bero på flera olika faktorer, som operativsystem, typ av projekt, ålder på projektet (och därmed versionshanteringen) eller helt enkelt personlig preferens. De två mest använda systemen idag är Git och SVN.[16]

7.3.4 Metod

Den metod som användes för att besvara frågeställningen delades upp i två delar.

7.3.4.1 Observation

Under projektet och framför allt under de första två iterationerna, hade de övriga projektmedlemmarna många frågor. Detta kom naturligt av det faktum att majoriteten inte hade någon tidigare erfarenhet av VCS och Git. Det uppstod även problem emellanåt, där orsaken oftast kunde spåras till felaktig användning av Git. Dessa frågor och problem antecknades och fördes in i en tabell, för att på så sätt få en översikt över de vanligaste problemen. De problem som var specifika för Git filtrerades bort, då rapportens frågeställning är generell.

7.3.4.2 Analys

Den insamlade datan kunde sedan analyseras. Detta gjordes genom att gruppera alla identifierade problem enligt följande; alla problem med samma rotorsak placerades i samma grupp. Med denna indelning kunde sedan de vanligaste problemen identifieras.

7.3.5 Resultat

Nedanstående tabell illustrerar de identifierade (ej analyserade men filtrerade) problemen, insamlade under hela projektet.

Problem	Förklaring	Frekvens
Kodkonflikter	Kod som synkroniseras mot en central server ”krockar” med kod från en annan utvecklare	8
Kod kompilerar lokalt men inte för andra	Kod som fungerar lokalt fungerar inte för andra utvecklare efter synkronisering mot server	6
Kod har märkligt och odefinierat beteende efter synkronisering	Efter en synkronisering mot server beter sig kod annorlunda än hos andra utvecklare	4
Filer utan ändringar synkroniseras	Filer som till synes är helt oförändrade av utvecklaren markeras som ändrade och synkroniseras	3

7.3.6 Diskussion

7.3.6.1 Metod

Metoden kan ifrågasättas på flera grunder.

Till att börja med är urvalet för litet för att kunna dra en slutsats som är representativ för all utveckling med VCS. Detta är dock inte målet med denna rapport, utan snarare att destillera erfarenheter till framtida utvecklingsprojekt. Därför kan metoden ändå generera användbara resultat.

Ett annat problem är att urvalsgruppen till stor del består av noviser inom versionshantering, vilket vinklar resultatet något. I en mer erfaren grupp är det möjligt att problemen aldrig hade uppstått, alternativt att problemen kunde lösas enkelt och därför inte hade rapporterats. Dock kan detta också ses som en positiv egenskap, om frågan som ställs istället är: vilka problem kan uppstå då någon skall lära sig versionshantering?

Trots metodfelen är bedömningen att den data som har samlats in ändå kan användas för att svara på frågeställningen.

7.3.6.2 Resultat

7.3.6.2.1 Kodkonflikter

Kodkonflikter var den absolut vanligaste typen av problem i projektet, något som uppstår då samma kod ändras samtidigt. Det kan finnas flera orsaker till detta problem, och en uppenbar anledning är kommunikationsproblem mellan utvecklare; man arbetar med samma kod samtidigt. Det är enkelt åtgärdat genom tydlig fördelning av arbetsuppgifter, och kommunikation i de fall där någon behöver ändra i kod som tillhör någon annans arbete.

En annan, mindre uppenbar anledning är storleken på synkroniseringarna. Om en utvecklare skriver mycket kod innan en synkronisering ökar risken att den modifierade koden hamnar i konflikt med annan kod. Även här är problemet lätt att åtgärda, nämligen att minska storleken på varje enskild synkronisering mot det centrala repositoriet. Det är också en bra strategi av anledningen att ändringar kan bli enklare att förstå, och revisionerna blir mer granulära.

7.3.6.2.2 Kod kompilerar lokalt men inte för andra

Problemet orsakas i majoriteten av fallen att kod som har skapats lokalt, i nya filer, inte följer med synkroniseringen. Resultatet blir att den lokala kopian är korrekt, medan den centrala kopian saknar viss kod. Genom att försäkra sig om att de filer utvecklaren själv har skapat och eventuella autogenererade filer från utvecklingsverktyg finns med i versionshanteringsklientens synkroniseringslista blir detta ett icke-problem.

7.3.6.2.3 Kod har märkligt och odefinierat beteende efter synkronisering

Det här är ett problem där orsaken kan vara svår att identifiera. I det här projektet var dock orsaken uteslutande att filer genererade av utvecklingsverktygen inte hanterade den ihopslagning av filer som VCS använder. Ett exempel är projektfiler för Visual Studio. Dessa håller bland annat information om projektstrukturen, och om den ändras samtidigt av utvecklare kan projektfilen hamna i ett odefinierat tillstånd. Resultatet blir kompileringsfel eller odefinierat beteende.

Lösningen är att lägga till undantag för de filer som inte klarar av ihopslagning, så att dessa inte synkroniseras. I de fall när detta inte är möjligt kan den enda lösningen vara att klona projektet till en ny mapp, och utgå från den i fortsättningen.

7.3.6.2.4 Filer utan ändringar synkroniseras

Detta är också ett problem ofta orsakat av utvecklingsverktyg och integrerade utvecklingsmiljöer som skapar eller uppdaterar projektspecifika filer, och autogenererar kod. Det är i sig inget problem men kan dock vara en orsak till förvirring.

7.3.7 Slutsatser

Baserat på resultatet är slutsatsen att många av de upplevda problemen kan förhindras med följande råd:

- Utvecklare bör ha tydligt definierade arbetsområden, för att undvika arbete med samma kod.
- Utvecklare bör kommunicera med varandra kontinuerligt.
- Förståelse för utvecklingsverktyget och hur det hanterar projektfiler och autogenererad kod behövs.
- Versionshanteringssystemets undantagssystem bör användas för alla icke-nödvändiga filer.

7.4 Mattias Lantz Cronqvist – Testansvarig

7.4.1 Sammanfattning

Det är viktigt att automatiskt kunna testa gränssnitt inom mjukvaruutveckling. Det finns flera olika metoder man kan ta hjälp av och designmönster som kan underlätta för testningsarbetet. Rapporten kommer ta upp ett enkelt men tidskrävande sätt att automatisera testarbetet. Metoden går ut på tre olika steg man ska följa för att garantera ett vältestat användargränssnitt (GUI). Rapporten avgränsar sig till C# och WPF.

7.4.2 Inledning

Att automatiskt testa ett GUI kan vara svårt, det kan involvera många olika tillstånd och händelser. Anledningen till detta är att indata ofta är interaktiv medan utdatan antingen kan vara grafisk eller eventbaserad. Indata kan ofta innehålla väldigt många permutationer som behöver testas och när fel väl uppstår är det inte ens säkert att användaren kan se det på skärmen. [17]

7.4.2.1 Motivering

Inom systemutveckling är det viktigt att på ett smart sett automatiskt testa GUI. Hur man gör detta är dock inte alltid helt trivialt. Denna rapport kommer att gå igenom tre steg man kan göra det på med fokus på C# och WPF.

7.4.2.2 Syfte

Syftet med denna rapport är att få en klarare bild av en metod för att automatiskt testa ett GUI i WPF samt olika designmönster för att underlätta testningen.

7.4.2.3 Frågeställning

Denna rapport kommer besvara följande frågor. Hur kan man automatiskt testa ett GUI på ett effektivt sett? Hur kan man underlätta för GUI-testning vid planering av projekt?

7.4.2.4 Avgränsningar

Frågeställningen kommer att begränsas till GUI inom C# och WPF och besvaras utifrån testledares synpunkt i projekt inom mjukvaruutveckling. Etiska aspekter kommer inte att diskuteras eftersom dessa inte är relevanta för frågeställningen.

7.4.3 Teori

Här beskrivs teorin bakom tre steg för att automatiskt testa GUI. Det första steget är att spela in mus-/tagentbordsgester med någon extern programvara där man går igenom programmets alla tillstånd.

Här är det viktigt att alla tillstånd tas med annars kan det leda till att felaktigheter i systemet inte hittas. Nästa steg är att ta skärmdumpar på systemets alla tillstånd och spara dessa i en databas. Alla tillstånd som spelats in med mus eller tangentbord behöver det finnas skärmdumpar på. Det sista steget är att spela upp gesterna från det första steget och jämföra alla tillstånd med skärmdumparna i databasen. Här kan till exempel pixlars värden jämföras för att hitta skillnader i körningarna. Om en skillnad finns kan det tyda på att något gått fel.

Det är väldigt viktigt att man bygger systemet så att GUI-testning ska vara enkelt. Det kan man göra genom att använda designmönstret MVVM där all logik flyttas från vyn och in i vymodellen. Då kan man med enhetstester hela tiden hålla koll på vad som händer i bakgrunden och eventuella komplikationer.

7.4.4 Metod

För att besvara frågeställningen så har information från vetenskapliga artiklar tagits. Artiklarna hittades främst via Google Scholar.

7.4.5 Resultat

Att automatiskt testa ett GUI är svårt. Att sätta upp en bra testmiljö och följa de presenterade stegen tar mycket tid. Om man senare gör om gränssnittet gör det att mycket av arbetet går förlorat och måste göras om. I rapporten tas 3 steg fram som tillsammans kan ge en automatiserad testmiljö för GUI i WPF.

7.4.6 Diskussion

Här diskuteras resultatet och metoden.

7.4.6.1 Metod

Metoden som beskrevs består av 3 steg där alla steg är minst lika viktiga. Om man bara använder sig av enhetstestning och antar att gränssnittet ser ut på ett visst sätt kan man inte garantera att det faktiskt gör det. På samma sätt garanterar inte steg 1 och 2 att bara för att det ser rätt ut på skärmen behöver programvaran fungera korrekt. Ett exempel på det här kan vara en skicka knapp, du kan få ett meddelande om att något skickats iväg utan problem men du kan inte verifiera att systemet faktiskt gjort det.

7.4.6.2 Resultat

Om man har ett gränssnitt som man inte riktigt vet hur det kommer se ut i slutet kan det vara onödigt och slöseri med tid att sätta upp en testmiljö. Men så fort man vet att endast mindre ändringar av gränssnittet kan komma hända kan det vara värt att sätta upp en testmiljö. Oftast har man redan steg 3 klart om man kör någon form av testdriven utveckling, till exempel TDD, så det kan räcka med de två första stegen.

Men om man inte har någon utav automatisk testning kan det fort gå över styr med olika tillstånd systemet kan vara i, “for just three windows in a screen with five modes. there are potentially 1050 possible combinations to test”. Man kan enkelt ha mycket större system än så och som man kan förstå blir det orimligt att testa allting manuellt, utan man måste i många lägen förlita sig på att saker fungerar. Har man då istället investerat lite tid i att bygga upp en bra och stabil testmiljö kan det spara tid i längen. [18]

7.4.7 Slutsats
Att automatisera testning av ett GUI kräver en tidsinvestering men kan tillslut leda till att tid sparas. I Digimergo projektet bestämdes det för sent hur användargränssnittet skulle se ut och därför ansågs det inte vara värt att använda denna presenterade metod.

7.5 Anders Söderström – Specialist inom användbarhet

7.5.1 Sammanfattning
Ett gränssnitt för DSE skulle utformas för att ge användaren ett så enkelt sätt att navigera som möjligt. I denna del behandlas arbetsgången bakom hur användargränssnittet togs fram. Resultatet baserades på vilken lösning som slutförde alla kraven med minst antal interaktionssteg för användaren.

7.5.2 Inledning
För att ett datorsystem ska gå att använda krävs ett gränssnitt mellan datorn och användaren. Hur dessa gränssnitt ser ut och fungerar finns det många lösningar på. I denna del kommer två olika gränssnittslösningar jämföras för att se vad som passar bäst för DSE.

7.5.2.1 Motivering
Valet av gränssnitt ligger ofta till grund till hur hela systemet ska användas och vad som ska gå att göra.

7.5.2.2 Syfte
Vilka konsekvenser valet av gränssnitt har för användaren.

7.5.2.3 Frågeställning
Frågeställningen för denna del är följande:

- Vilken gränssnittslösning är bäst utformad för Digimergo?
- Hur vet man att valet verkligen är det bästa?

7.5.2.4 Avgränsningar
Två olika gränssnittslösningar kommer jämföras, Multiple Document Interface (MDI) och Tabbed Document Interface (TDI).

7.5.3 Teori
Här beskrivs teorin bakom de två gränssnittslösningarna

7.5.3.1 Multiple Document Interface
Detta förslag fokuserarar på att dela upp innehåll till flera separata fönster. Detta för att kunna se en eller flera arbetsytor samtidigt. Med en arbetsyta menas en yta där flera objekt går att placera ut på. Det ger användaren mer kontroll över utseendet och gör det möjligt att placera arbetsytorna mer fritt på skärmen istället för att låsas till endast en synlig arbetsyta åt gången.

7.5.3.2 Tabbed Document Interface
I detta gränssnittsförslag jämfört med MDI ligger fokus mer på att hålla ihop hela programmet i ett fönster. Detta ger ett mer överskådligt arbetssätt när endast en arbetsyta är synlig åt gången. [19]

7.5.4 Metod

Metoden för att ta fram gränssnittet för DSE-mjukvaran byggde på förslagsskisser i form av olika prototyper där enkelhet för användaren låg i fokus. För att inte låsas till en viss layout skapades tio stycken helt olika skisser på hur systemet kunde tänkas designas. Med helt olika menas att navigeringen i varje skiss skulle skilja sig från resterande skisser. Alla skisser utgick från att användare på ett enkelt och smidigt sätt ska kunna skapa och flytta runt objekt på olika arbetsytor. Detta var då informationen som tagits ut från det tidigare systemet över vilka grundläggande funktioner DSE måste ha stöd för. Två olika gränssnitt togs fram där det ena fokuserade mer på att hantera arbetsytor i separata fönster likt MDI, medan de andra lösningarna fokuserade mer på hanteringen av hela scenariot i ett fönster likt TDI.

Sedan valdes det bästa förslaget ut som gick vidare till nästa steg där mer funktionalitet lades till. Varje skiss betygsattes då efter hur många interaktionssteg enkla uppgifter skulle ta att genomföra. En enkel uppgift kan exempelvis vara att lägga till ett nytt objekt i en arbetsyta.

Vilken prototyp som sedan gick vidare till testning baserades på vilken gränssnittslösning som utförde uppgifterna med minst antal interaktionssteg. Användartestet utfördes på åtta olika personer. Testerna bestod av tre olika uppgifter som skulle utföras. Innan varje test lästes uppgiften upp och sedan svarade användaren hur svår uppgiften uppskattades att vara på en skala mellan 1 och 7. Uppgiften utfördes sedan av användaren som under testet beskrev tankesättet bakom varje beslut. Efter varje slutfört test fick användaren på nytt uppskatta svårigheten på en skala mellan 1 och 7.

7.5.5 Resultat

Resultatet från användartesterna listas nedan:

Uppgift 1: “Skapa en ny olycksplats på Ringargatan”
Medelbetyg innan övning sattes till 3 av 7. Efter slutförd övning uppskattades svårigheten till 2 av 7.

Uppgift 2: “Lägg till en patient på olycksplatsen, som ska synas när branden är släckt.”
Medelbetyg innan övning sattes till 6 av 7. Efter slutförd övning uppskattades svårigheten till 4 av 7.

Uppgift 3: “Skapa en ny händelse som aktiveras efter 20 minuter”
Medelbetyg innan övning sattes till 5 av 7. Efter slutförd övning uppskattades svårigheten till 1 av 7.

7.5.6 Diskussion

Här diskuteras metoden och resultatet från användarstudien.

7.5.6.1 Metod

Metoden som användes för att skapa gränssnittet bestod av idéskisser som sedan utvecklades vidare till en prototyp. I detta fall skapades prototypen innan dom flesta kraven var sammanställda vilket skapade lite problem om prototypen som togs fram verkligen blev den bästa om alla kraven hade varit klar innan start. Att bestämma prototyp efter antal interaktionssteg kan ge en felaktig bild över vad som är mest användarvänligt. Då den inte tar hänsyn till hur svårt användaren upplever att navigeringen i gränssnittet är.

7.5.6.2 Resultat
Resultatet från användarstudien var till stor hjälp för att eliminera tveksamma designbeslut. När det kommer till vilken lösning som är bäst att använda är användarstudien inte tillräcklig för att ta ett sådant beslut. Där fick antal interaktionssteg ligga till grund för valet.

7.5.7 Slutsats
Slutsatsen av det hela är att valet av gränssnitt alltid går att förbättra. Därför är den bästa lösningen på gränssnittet där all funktionalitet som systemet kravställs på är möjligt att utföra.

7.6 Fredrik Präntare – Utvecklare

7.6.1 Sammanfattning
Forskningsområdet designmönster är relativt nytt. Därmed kan det vara intressant att utforska något av alla de designmönster som finns. I den här delrapporten diskuteras för- och nackdelar med det relativt outforskade designmönstret “Model View ViewModel” (MVVM) utifrån dess implementation i ett riktigt programvaruprojekt. På grund av vissa begränsningar är studien endast översiktlig. Studien kommer fram till att MVVM har fördelar när det gäller abstraktion och databindning medan programkomplexiteten som följer är ett mindre och hanterbart problem.

7.6.2 Inledning
I programvaruutvecklingsprojekt görs ofta ett flertal beslut gällande kodarkitektur, designmönster och programmeringsmetodik. I denna delrapport förklaras för- och nackdelarna med ett av de viktigaste designmönstren som använts i Digimergo: MVVM.

I de system som projektgruppen utvecklade användes ett flertal olika designmönster. Det övergripande designmönstret som användes i de flesta delsystemen var MVVM.

7.6.2.1 Motivering
Designmönster inom programvaruutveckling är en katalogisering av vanliga programvaruutvecklingsproblem samt deras vanliga lösningar. Denna katalogisering härstammar från teori inom arkitektur och samhällsplanering. I “Notes on the synthesis of form” från 1964 beskriver Christopher Alexander hur designmetodik och designmönster uppstår som ett nytt forskningsområde, där han, emot sin egen vilja, beskrivs som en ledande person. [20] Utifrån Christopher Alexanders arbeten på designmönster började sedermera Kent Beck och Ward Cunningham applicera designmönster på programvaruutveckling. [21]

Eftersom designmönster är ett relativt nytt och outforskat forskningsområde finns det fortfarande utrymme för nya jämförelser av designmönster. Det kan vara intressant att utforska vilka för- och nackdelar olika designmönster har, för att man då kan använda rätt designmönster vid rätt tillfälle. MVVM är i sig ett väldigt intressant designmönster då det är nytt, relativt outforskat och samtidigt har stort stöd i en av de större utvecklingsmiljöerna: Microsoft Visual Studio.

7.6.2.2 Syfte
Syftet med denna delrapport är att översiktligt utforska för- och nackdelar med det centrala designmönstret MVVM som använts i Digimergo.

7.6.2.3 Frågeställning

Den centrala frågeställningen för denna delrapport, baserat på syftet, är: Vilka för- och nackdelar har det centrala designmönstret MVVM jämfört med "Model View Controller" (MVC) i ett programvaruutvecklingsprojekt som liknar Digimergo?

7.6.2.4 Avgränsningar

I denna delrapport kommer MVVM endast översiktligt jämföras med ett annat liknande designmönster: Model View Controller (MVC).

Arbetet saknar koppling till samhälleliga och etiska aspekter.

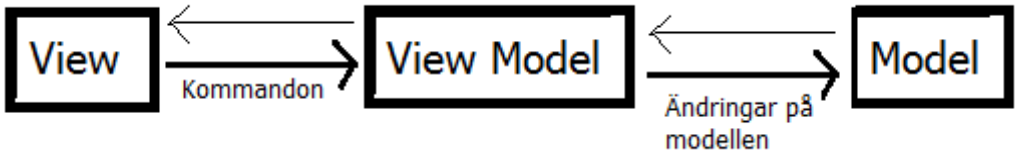
7.6.3 Bakgrund

Denna delrapport är en del av ett större kandidatprojekt. I detta kandidatprojekt fick en projektgrupp från civilingenjörsprogrammet inom datateknik uppgiften att vidareutveckla systemet Digimergo. Digimergo är en digitalisering av katastrofövningssystemet Emergo Train System. Flera delsystem som använder sig av MVVM utvecklades.

7.6.4 Teori

Avsnittet beskriver begreppen MVVM och MVC.

7.6.4.1 Model View ViewModel



Figur 2: Visualisering av det generella designmönstret "Model View ViewModel".

"Model View ViewModel" (MVVM) används för att abstrahera bort det visuella lagret (view) från den bakomliggande logiken (model) med hjälp av objektorienterad programmering. Detta görs genom att modellen inte får någon tillgång till det visuella lagret eller vice versa. Istället konstruerar man ett mellanlager (view model) som hanterar all kommunikation mellan modellen och det visuella lagret. För en illustration av MVVM se Figur 2.

Kombinationen MVVM och C# låter utvecklaren skriva all grafikrelaterad kod i XAML istället för direkt i det visuella lagret. Med hjälp av XAML och C# kan man sedan databinda den XAML-skrivna koden till mellanlagret.

7.6.4.2 Model View Controller

"Model View Controller" (MVC) är likt MVVM ett designmönster som isolerar den bakomliggande logiken (model) från vyn (view). Istället för ett mellanlager som tillåter databindning och XAML används istället ett mellanlager (controller) som delvis hanterar input från användaren. Mellanlagret meddelar sedan modellen om modellen behöver uppdateras. Modellen underrättar själv vyn när vyn behöver uppdateras.

7.6.5 Metod
För att besvara frågeställningen jämförs MVVM med MVC genom att diskutera hur Digimergo kunnat se ut implementerat i MVC istället för MVVM.

7.6.6 Resultat
I MVVM representerar mellanlagret en vy. Detta betyder i praktiken att mellanlagret är mer sammankopplat med vyn än modellen. I MVC är mellanlagret närmare modellen eftersom mellanlagret direkt eller indirekt kan ändra på modellen. I Digimergo har MVVM implementerats så att mellanlagret består av små egna moduler som tar hand om databindning mellan vyn och modellen. Detta gör att vyn blir oberoende från modellen och vice versa.

Digimergo utvecklades i C# med hjälp av Microsoft Visual Studio. Microsoft Visual Studio stödjer MVVM, C# och XAML. C# har stort stöd för databindning och passar således väldigt bra för system implementerade med MVVM.

7.6.7 Diskussion
Avsnittet diskuterar metoden och resultatet.

7.6.7.1 Metod
Att jämföra hur Digimergo sett ut med MVVM istället för MVC valdes som metod eftersom delrapporten har en begränsning på tre sidor. Begränsningen leder till att MVVM endast kan jämföras med ett begränsat antal liknande designmönster. Begränsningen leder också till att jämförelsen blir översiktlig och svepande. Om studien haft mindre begränsningar, hade jämförelsen kunnat göras grundligare. Resultatet blir av samma anledning något bristande. Det är heller inte säkert att studien har hög reliabilitet då en stor del av studien diskuteras utifrån systemet Digimergo. Eftersom MVC och MVVM är två väldefinierade designmönster, som även definieras i den här studien, är validiteten i studien hög. Källorna som används är alla skrivna av personer högt ansedda inom forskningsområdet designmönster. Dessa källor bör anses ha hög trovärdighet.

7.6.7.2 Resultat
MVVM har många likheter med designmönstret MVC. Till skillnad från MVC har MVVM fördelar när det kommer till databindning och passar således väldigt bra i kombination med C# och XAML. Eftersom Digimergo vidareutvecklades på en systembas skriven i C# kom det sig naturligt att implementera de nya delsystemen med hjälp av MVVM. Hade istället Digimergo utvecklats i ett annat utvecklingsspråk hade ett annat designmönster möjligtvis använts.

En möjlig nackdel med MVVM jämfört med MVC är att MVVM, i C#, vanligen implementeras med ett extralager skrivet i XAML. I MVVM och Digimergo har även varje mellanlager en egen vy medan man i MVC oftast bara använder en vy. Ett extralager och flera vyer leder till flera komponenter för samma funktion. I vissa fall kan det öka ett systems abstraktion och möjlighet till vidareutveckling medan det i andra fall kan leda till onödig programkomplexitet. I fallet Digimergo är ökad abstraktion viktigt då Digimergo är ett system under ständig utveckling med flera utvecklare. Programkomplexiteten är inget större problem då den för projektgruppen visat sig vara hanterbar.

7.6.8 Slutsats
I ett projekt som liknar Digimergo har MVVM fördelar när det gäller abstraktion och databindning. Programkomplexiteten som följer av MVVM är ett mindre och hanterbart problem.

7.7 Oscar Nöjdh – Analysansvarig

7.7.1 Sammanfattning
I Digimergo-projektet fanns det två parter vars förväntningar skulle infrias av projektets resultat. Prioriteringarna från dessa skulle balanseras i en kravspecifikation som gjorde båda parter nöjda. Den här delrapporten diskuterar hur man gör detta.

7.7.2 Inledning
Att analysera användarnas behov kan vara kritiskt för ett projekt. Om man arbetar med flera parter som hoppas att vinna något på projektet blir det extra viktigt att analysen utförs på ett bra sätt.

7.7.2.1 Motivering
Denna delrapport behandlar balansering av kravprioritering i ett mjukvaruprojekt med flera parter. När man arbetar mot flera parter i ett projekt är det mycket viktigt att kravanalysen utförs på ett rikitg sätt eftersom fler måste bli nöjda med den.

7.7.2.2 Syfte
Syftet med denna delrapport är att diskutera metoder för att ta fram en kravspecifikation anpassad till flera parter.

7.7.2.3 Frågeställning
De frågor som kommer att besvaras i denna delrapport är:

- Hur tas en kravspecifikation fram utifrån användarens behov?
- Hur balanserar man kundens behov mot användarens behov?

7.7.2.4 Avgränsningar
Denna delrapport kommer inte behandla ett stort antal olika metoder för att ta fram kravspecifikationer utan kommer istället utgå från för- och nackdelar med metoden använd i Digimergo-projektet. Etiska aspekter kommer inte att diskuteras eftersom dessa inte är relevanta för ämnet.

7.7.3 Bakgrund
I detta projekt utvecklades Digimergo, en digitalisering av katastrofövningssystemet Emergo Train System. Projektets resultat skulle motsvara kundens förväntningar samtidigt som dessa kunde infrias inom tidsbudgeten. Därför togs en kravspecifikation fram där kunden och projektgruppen var överens om vad som skulle göras.

7.7.4 Teori
Kravanalys är det arbete som utförs för att ta fram användarens behov. Brister i kravanalysen kan leda till ökade kostnader, missade tidsplaner och lägre kvalitet på den resulterande mjukvaran.[22] Kravanlys kan utföras på olika sätt. En central del är intervjuer med användaren. Det kan vara en ren

intervju med förbestämda frågor men det är även vanligt att man observerar användaren i situationer där mjukvaran som ska utvecklas kunde ha använts. Att arbeta med prototyper är ett annat sätt att observera vad användaren behöver. Man kan då observera interaktion med "programmet" redan innan det har konstruerats. Man kan få ut mycket användar återkoppling med en prototyp men att konstruera prototyper kan vara tidskrävande. Tiden och kostnaderna som krävs för att producera en prototyp är oftast små jämfört med värdet på informations vinsten man får ut av den.[23]

När man utvecklar en produkt åt en kund som sedan ska distribuera den vidare får man ta hänsyn till två parter när man analyserar kraven. Kundens och användarens prioriteringar kan skilja sig i saker som funktionalitet och pris. Att konstruera produkten med bara användaren i åtanke kan resultera i en annan produkt än vad kunden efterfrågade.

7.7.5 Metod

Kunden hade redan vid projektets start arbetat fram en kravspecifikation. Detta hade gjorts eftersom mer inflytande ville ges till Katastrofmedicinskt Centrum (KMC) under kravspecifikationens skrivande. Jobbet som analysansvarig blev därför inte att analysera vad det var kunden ville ha och skriva en egen kravspecifikation. Istället analyserades och prioriterades kraven som kunden redan dokumenterat. Analysprocessen började med flera intervjuer med kunderna. På dessa möten gavs en klarare bild av vad våra kunder hade för förväntningar på produkten och var deras prioriteringar låg. Information om det existerande systemet införskaffades också vid dessa möten. Eftersom kunden hade höga förväntningar på projektet hölls i början fyra möten och regelbunden mailkontakt för att diskutera fram en specifikation som båda parter var nöjda med. Representanter från projektgruppen skuggade även riktiga övningar på KMC för att ta reda på användarnas behov. Efter att ha skuggat övningar på KMC sågs kravspecifikationen över igen. Uppgiften var då att flytta fokus från kundens prioriteringar till instruktörernas. Kunden bekräftade att ett sådant fokus var att föredra. Erfarenheterna på KMC ledde inte till några nya krav utan gav en klarare bild av hur kraven skulle prioriteras. En prototyp konstruerades och testades med testanvändare liknande de slutgiltiga användarna. Strax efter halvtid i projektet visades produkten upp För KMC för verifiering.

7.7.6 Resultat

Analysarbetet resulterade i en kravspecifikation som projektgruppen ansåg genomförbar och som motsvarade kundens förväntningar på resultatet. Kravspecifikationen reviderades ett par gånger under första halvan av projektet för att sedan förbli oförändrad under den andra halvan. Kundens och användarnas prioriteringar visade sig vara de samma. Demon för KMC i mitten av projektet verifierade att produkten motsvarade kundens förväntningar.

7.7.7 Diskussion

Kravspecifikationens fokus hamnade i det här projektet på användarens behov. Kundens målsättning var att uppfylla användarens behov till så stor grad som möjligt. Man kan därför säga att både kundens och användarens målsättning uppfylldes. Kunden hade i början av projektet för höga förhoppningar om vad som skulle rymmas i den projektets begränsade tidsbudget. Detta löstes genom att nedprioritera krav kunden tyckte kunde vänta. Användarna var genom hela projektet väldigt intresserade av vad som skedde i projektet och kom hela tiden med användbar input. Eftersom Digimergo bygger på ett system som används idag vet användarna precis vad det är dom är

ute efter. Hade användarintervjuerna analyserats utan kundens grundspecifikation hade viktiga funktioner missats. Flera funktioner i det nuvarande systemet används sällan och dessa hade varit svåra att få med i en kravspecifikation. Prototypen borde ha testats på KMC med de riktiga användarna. Detta var målet från början men då gemensamma tider var svåra att hitta och projektet inte fick stanna upp blev dessa tester inte av. Hade dessa tester blivit av hade detta antagligen gett mycket nyttig återkoppling. Eftersom ETS används till så många olika sorters övningar är det svårt att hitta alla funktioner användaren vill ha bara genom observation. En prototyp ger dessutom kunden en chans att komma med åsikter om produkten innan det är för sent att göra större ändringar.

7.7.8 Slutsats

Om kundens och användarnas förväntningar stämmer överens gör detta kravanalysen lättare. Detta ger dessutom fler källor till kravanalysen. En prototyp är bra att konstruera för att kunna visa användarna vad man tänkt sig och få återkoppling innan projektet har gått så långt att ändringar är svåra att göra.

7.8 Carl Einarson – Arkitekt

7.8.1 Sammanfattning

I ett mjukvaruprojekt lägger man oftast ner mycket tid på arkitekturen men hur nödvändigt är det i ett kort projekt? I den här rapporten diskuteras för- och nackdelar med att ha en arkitekt i ett projekt som bara tar 6 till 10 månader. Information ifrån vetenskapliga artiklar har sammanställts för att ta reda på vad en arkitekt gör och hur viktigt det är i ett kortare projekt.

7.8.2 Inledning

Arkitekt är en vanlig roll i mjukvaruprojekt men exakt vad gör en arkitekt i projektet och hur viktigt är det för projektet? Denna rapport kommer att fokusera på vad arkitektens arbete faktiskt innebär samt vad arkitekten bör fokusera på för att vara till stor nytta för projektet. Den fråga som kommer besvaras är hur det kan hjälpa till att ha en arkitekt i ett utvecklingsprojekt som ska pågå i 6-10 månader.

7.8.2.1 Motivering

En arkitekts arbete i ett mjukvaruprojekt går ut på att definiera mjukvaruarkitekturen, vilket innebär att fatta beslut angående begränsningar i implementationen och designen av systemet. Detta är för att det ska vara självklart för projektmedlemmarna vilka begränsningar som ska finnas i systemet. Hur detta ska göras på ett bra sätt kan vara svårt och kräver att arkitekten vet mycket om ämnet för att göra rimliga avgränsningar. Detta gör att det kan gå åt mycket av projekttiden på att göra arkitekturen vilket kan kännas onödigt.

7.8.2.2 Syfte

Syftet med rapporten är att översiktligt gå igenom en arkitekts uppgifter i mjukvaruprojekt, att undersöka vad som är bra egenskaper hos en arkitekt och om det är nödvändigt med en arkitekt i ett projekt som bara ska hålla på i 6 till 10 månader.

7.8.2.3 Frågeställning
Frågeställningen för denna del är följande:

- Behövs en arkitekt i ett litet projekt (6-10 månader långt)?

7.8.2.4 Avgränsningar
Rapporten är avgränsad till att endast fokusera på de centrala aspekterna av en arkitekts arbete. Etiska aspekter kommer inte att diskuteras eftersom dessa inte är relevanta för ämnet.

7.8.3 Teori
Detta stycke kommer att beskriva vad en arkitekts arbete huvudsakligen går ut på och vilka egenskaper som kan vara bra hos en arkitekt.

7.8.3.1 Arkitektens arbete
Arkitektens främsta uppgifter är att ta beslut om begränsningar i implementationen och designen av systemet. Detta görs lämpligtvis med ett dokument kallat “Architecture Notebook” där alla projektmedlemmar kan gå in och läsa om begränsningarna och motivationen bakom dem. Arkitekten hjälper även till med att skriva krav och att designa systemet [24].

7.8.3.2 Viktiga egenskaper för en arkitekt
Några av de viktigaste egenskaperna hos en arkitekt är att vara erfaren i problemlösning och mjukvaruutveckling och att kunna kommunicera sina beslut på ett bra sätt för projektgruppen så att de får reda på de beslut som arkitekten tagit angående systemet.

För att ta beslut så måste arkitekten kunna göra kvalificerade uppskattningar om hur systemet kommer att se ut och snabbt kunna ta beslut om problem som kan uppstå i projektet.

Arkitekten bör fokusera på en minimalistisk syn på utvecklingen för att få det bästa resultatet[25].

7.8.4 Metod
För att svara på frågeställningen så har information ifrån vetenskapliga artiklar sammanställts och utvärderats. Informationen har främst hittats via Google Scholar.

7.8.5 Resultat
Det kommer att löna sig att ha en arkitekt även i ett mindre projekt eftersom det kommer att ge en bra grund på hur man ska börja även om det tar längre tid att starta upp projektet. Då kommer projektmedlemmarna att veta vad och hur det ska göras vilket gör att produkten kommer att utvecklas korrekt.

7.8.6 Diskussion

7.8.6.1 Metod
Metoden som används har i detta fall varit väldigt lämplig eftersom det finns mycket dokumentation om hur en arkitekts arbete går till och varför det är viktigt att ha en arkitekt inom mjukvaruutveckling.

Det hade även kunnat vara bra att studera projekt som har utvecklats med och utan arkitekt och jämföra tidsåtgång. Det är dock svårt att hitta information om detta och resultatet av projekten påverkas av mer än vad arkitekten vilket gör det svårt att göra en bedömning av arkitektens arbete.

7.8.6.2 Resultat

Utan en arkitekt är det lätt att projektet tar alldeles för lång tid eller utvecklas fel då det inte avgränsats på ett bra sätt och onödig funktionalitet kan ha lagts till. En minimalistisk syn på vad produkten ska innehålla ger det bästa resultatet [25]. Detta saknas då man inte har en arkitekt i projektgruppen eftersom man då inte avgränsar systemet.

Med en arkitekt så kommer det att kräva en del arbete i början av projektet för att hitta avgränsningar och designval för projektet vilket gör att själva programmeringen inte kommer att börja direkt men projektgruppen kommer att veta vilka begränsningar som finns och vad som ska fokuseras på i projektet och hur det ska utvecklas[24]. Detta gör att det är värt att ha med en arkitekt även om uppstarten kommer att ta längre tid.

7.8.7 Slutsats

I ett mindre projekt på 6 till 10 månader så är det värt att ha med en arkitekt eftersom alla projektmedlemmar då kommer att veta hur systemet ska byggas upp och vad det finns för avgränsningar. Det minskar även risken för att produkten utvecklas fel och tar längre tid än nödvändigt.

Om projektgruppen redan vet om avgränsningarna och hur systemet bör byggas upp så kommer inte en arkitekt fylla någon viktig funktion för gruppen.

8 Referenser

[1]Forselius, P. (2008). Software development program characteristics. Available: http://citeseerx. ist. psu. edu/viewdoc/summary.

[2]Chitu Okoli, Suzanne D. Pawlowski, Information & Management (Vol. 42), Amsterdam, Netherlands: Elsevier Science, 2004, Issue 1, p15-29. 15p.

[3] Royce, Winston, Managing the Development of Large Software Systems: 1970, WESCON technical papers volume 14, Western electronic show and convention

[4]Molyneaux, Tom & Brumley, John, “The wiki for project work” in The use of wikis as a management tool to facilitate group, Melbourne, Australia: 2007, Page 3-4.

[5] V. Sistenich, “International Emergency Medicine: How To Train For It”, Emergancy Medicine Australasia, vol. 24, no. 4, pp. 435-441, Augusti, 2012.

[6] K. Varshney et al., “Disaster Triage Tags: Is One Better Than Another?”, Emergency Medicine Australasia, vol. 24, no. 2, pp. 187-193, April 2012.

[7] Vice. Ryan, “MVVM survival guide for enterprise architectures in Silverlight and WPF”, Birmingham, Mumbai: Packt Publishing, 2012

[8] IEEE Standard for Software and System Test Documentation, IEEE Standard 829, 2008

[9] An experimental application of the Delphi method to the use of experts
Authors: Dalkey, Norman1, Helmer, Olaf1

[10] The Delfi Method
Lindstone, H., and Turoff, M. (ed.), The Delfi Method, Addison Wesley Publishing Co., 1975.

[11] Sudhakara, Goparaju Purna et al., ”MEASURING PRODUCTIVITY OF SOFTWARE DEVELOPMENT TEAMS,” in Serbian Journal of Management. Vol. 7 Issue 1, p65-75. 11p. DOI: 10.5937/sjm1201065S, 2012.

[12] Joseph Blackburn et al., “Brooks’ Law Revisited: Improving Software Productivity by Managing Complexity”, Maj 2006.

[13] http://www.linkedin.com/jobs2/view/12947906?trk=jobs_search_public_seo_page

[14] http://www.linkedin.com/jobs2/view/13366899?trk=jobs_search_public_seo_page

[15] CollabNet. (2007, May 15). Rapid Subversion Adoption Validates Enterprise Readiness and Challenges Traditional Software Configuration Management Leaders [Online]. Available: http://www.businesswire.com/news/home/20070515005055/en/Rapid-Subversion-Adoption-Validates-Enterprise-Readiness-Challenges#.U3FlnShpfgk

[16] The Eclipse Foundation. (2013, June). Eclipse Community Survey Report [Online]. Available: http://www.slideshare.net/IanSkerrett/eclipse-survey-2013-report-final

[17] White, L.J et al., ”User-Based Testing of GUI Sequences and Their Interactions”, Ohio: Los Alamitos, 2001

[18] White, L.J, “Regression Testing of GUI Event Interactions”, Ohio: Los Alamitos, 1996

[19]Human Computer Interaction by Rajendra Ranjan Kumar (1 Dec 2005)
2.6.3 “Tabbed Document interface (TDI)” sida 57

[20] Christopher Alexander, “Notes on the synthesis of form”, Harvard, preface, 1964.

[21] Reid Smith, “Panel on design methodology”, OOPSLA ’87, 1987.

[22] H. Jain et al., “An assessment model for requirements identification in component-based software development”, ACM SIGMIS, vol. 34, no. 4, pp. 48-63, hösten 2003.

[23] T. Z. Warfel, “Prototyping: A Practitioners Guide”, Rosfenfeld Media, November 2009.

[24] Kruchten, Philippe. The Software Architect. In: Software architecture. Springer US, 1999. P. 565-583.

[25] Hoftstader, Joseph. We don’t need no architects. The architecture journal, 2008, 2.

9 Appendix A - Affärskoncept

9.1 Affärsidé

Verksamheten har fokus på i huvudsak 2 områden, dels konsulttjänster med inriktning på digitalisering av ”analoga” system, dels utveckling och försäljning av programvara till Digimergo. Vad gäller konsulttjänster finns idag ett stort behov av konsulter med specifik kompetens, ett behov som vi kan tillfredsställa. Potentiella kunder finns inom alla sektorer av affärslivet där gamla, omoderna system behöver uppgraderas. Kundnyttan ligger i att så bra och kostnadseffektivt som möjligt uppgradera sina system.

Försäljning av licenser till programvara för Digimergo är företagets andra fokus. Digimergo bygger på Emergo Train System, ett system för katastrofövningar som är väl använt i Sverige och till viss del i många andra länder. Digimergo är den naturliga utvecklingen av Emergo Train System, och vår ambition är att våra produkter ska vara ett nödvändigt komplement för alla användare.

9.2 Affärsmodell

Affärsmodellen baseras dels på intäkter från konsultverksamheten, och dels på årligt återkommande intäkter för licenser på programvara. Intäkter från konsultverksamhet kan antas växa någorlunda linjärt med antalet anställda, medan licensförsäljningen kommer att expandera i jämn takt med Digimergos spridning.

9.3 Mål

Kortsiktigt är målet att etablera försäljningen av licenser inom majoriteten av Sveriges 20 landsting, för att på så sätt bygga upp en inkomstström och etablera produkterna. På sikt ska licensförsäljning påbörjas i andra länder. Det långsiktiga målet är att de produkter vi erbjuder ska bli en standard inom Digimergo och katastrofövningar.

Konsultdelen av verksamheten skall etableras så snabbt som möjligt, i den mån det är möjligt ekonomiskt. Tanken är att licensförsäljningen ska möjliggöra ekonomiskt tillväxt, och därmed nyanställningar. Med mer personal kan företaget börja utföra flera och större uppdrag.

9.4 Vision

Vi ska bli världsledande på tillbehör och programvara till Digimergo, och bedriva en väl etablerad konsultverksamhet i Sverige.

Ledningsgrupp

9.5 Ledning

- Anders Söderström – VD, specialist inom användbarhet
- Fredrik Präntare – PR, marknadsansvarig och kvalitetssamordnare
- Martin Bodin – Ekonomi, dokumentansvarig

9.6 Nyckelmedarbetare

Den ursprungliga gruppen består av ledningen och fem medlemmar som kallas in som konsulter. Dessa är:

- Jon Dybeck – Projektledare
- Marcus Jonsson – Utvecklingsledare
- Mattias Lantz Cronqvist – Testledare
- Oscar Nöjdh – Analysansvarig
- Carl Einarson – Arkitekt

Alla gruppmedlemmar har även roll som utvecklare.

Produkter

9.7 Produktsortiment

9.7.1 Konsultuppdrag

Konsulttjänster enligt överenskommelse, inom programvaruutveckling, systemintegration och modernisering av gamla system.

9.7.2 Licensförsäljning

Licens att använda vår programvara säljs till varje enskild instans som använder Digimergo. Beroende på önskad storlek på övningarna finns två olika licenser:

- Licens liten:
  - Maximalt 5 klienter kan hanteras simultant
  - Begränsad service
- Licens stor:
  - Obegränsat antal klienter
  - Obegränsad service

9.8 FoU

Den Forskning och Utveckling (FoU) som skall bedrivas är inledningsvis begränsad till området Digimergo och vidareutveckling av de verktyg vi erbjuder.

Marknadsplan

9.9 Marknaden

Marknaden för våra tjänster och produkter är stor, med förväntad ökning i framtiden.

9.10 Kunder

Vad gäller licensförsäljning är det huvudsakliga kundsegmentet landsting och myndigheter, d.v.s. de instanser som står för utbildning av sin räddningspersonal.

Konsultverksamheten riktar in sig på ett annat, bredare område, och kunderna kan därför variera, med allt från små till stora företag inom både statlig och privat sektor.

9.11 Konkurrenter

I dagsläget är vi ensamma om de produkter vi erbjuder till Digimergo, och kommer fortsätta vara det under den överskådliga framtiden. Vi vill utnyttja det till vår fördel och skapa så mycket värde som möjligt innan de första konkurrenterna träder fram.

Konsultverksamheten möts av större konkurrens, då många konsultföretag är verksamma inom programvaruutveckling. Vi räknar dock med att vår specialisering (digitalisering) medför att den konkurrens som finns inte påverkar oss negativt, då vi har vår nisch.

9.12 Marknadsstrategier

9.12.1 Prissättning

9.12.1.1 Konsulttjänster

Pris för uppdrag bestäms efter konsultation.

- Kostnad per timme för en utvecklare: 650 kr
- Minsta kostnad för projekt: 26000 kr (motsvarar en utvecklare anställd en normal arbetsvecka)

9.12.1.2 Licenser

Priset beror på licenstyp:

- Licens liten: 9000kr per månad eller 100000kr för ett år.
- Licens stor: 15000kr per månad eller 160000kr för ett år.

Om betalning per månad sker är bindningstiden ett år. Vid betalning för ett helt år direkt blir kostnaden mindre än vid månadsbetalning.

9.12.1.3 Försäljning och distribution

Försäljning och distribution sker direkt från våra lokaler och via de utvecklare som är tillgängliga för tillfället.

9.12.1.4 Service

Service ges enligt avtal för konsultuppdrag och enligt licenstyp för licensförsäljning. Vid konsultuppdrag kostar service extra.

9.12.1.5 Marknadsföring

Marknadsföringen skall främst ske genom att nöjda kunder sprider ett gott rykte om företaget och våra produkter. Vi ska även kontakta potentiella kunder direkt gällande licensförsäljning, och bedriva småskalig webbannonsering gällande konsulttjänster.

10 Ekonomisk översikt

10.1 Ekonomisk situation – nuläget

I dagsläget har vi få inkomster. De resurser vi har är vår färdigutvecklade programvara till Digimergo och de lokaler vi utnyttjar i sammarbete med KMC. Vi räknar också den specifika kompetens som våra utvecklare besitter som en resurs.

10.2 Finansiering

Under företagets första tid ska fokus ligga på att sälja så många licenser som möjligt. Intäkter från dessa ska betala anställdas lön och möjliggöra expandering på sikt.

10.3 Kapitalbehov

Med utomstående finansiering kan företaget snabbare komma ut på marknaden, varför vi ämnar söka detta. Det bör dock nämnas att företaget klarar sig utan finansiering. En utomstående finansiering motsvarande 1-2 utvecklares årslöner skulle kunna snabba upp företagets utveckling avsevärt.

10.4 Ekonomisk utveckling – prognos

Under det första året räknar vi med att sälja licenser till minst 18 av Sveriges 20 landsting, där hälften är ”små” och hälften är ”stora” licenser. Med detta i beaktning blir minimiintäkter för första året 2 340 000 kr, vilket är tillräckligt för att täcka utgifter för 3 utvecklare anställda på heltid, utrustning samt kostnader för att hyra in resterande teammedlemmar som konsulter i den mån det behövs och finns pengar. Vi räknar därför med att redan efter första året göra vinst, och därefter fokusera på att öka försäljning av licenser och konsulttjänster och därmed gå med vinst.

11 Appendix B – Erfarenheter

Under projektets gång så har trac använts för erfarenhetsfångst. Varje gång någon har tidsrapporterat så har denna även skrivit in vad som har gjorts och om eventuella problem har uppstått. I slutet av projektet gick vi igenom dessa kommentarer för att sammanställa och få en bra översikt över vad vi har gjort och lärt oss. Vi har även tillsammans i slutet av projektet diskuterat vilka erfarenheter vi har fått under utvecklingen i detta projekt.

11.1 Gruppens tekniska erfarenheter

Här följer gruppens tekniska erfarenheter.

11.1.1 C#

Som krav hade vi att systemet skulle vara byggt i C# och WPF. Ingen i gruppen hade tidigare erfarenheter av dem, därför räknade vi med en utbildningsdel i tidsplanen där det var tänkt att vi skulle utbilda oss inom ämnet. När vi väl började programmera satt vi i par för att känna oss mer bekväma i den nya miljön. Detta höjde också kvaliteten på koden eftersom alla inte blivit utbildade i exakt samma moment.

11.1.2 Git

För att organisera projektets källkod användes git och ett privat Github-arkiv. Redan vid projektstart bestämdes det att vi skulle använda någon form av versionshantering. Detta eftersom det är väldigt viktigt att veta vem som gjort en ändring och när, vilket nästan är omöjligt utan ett versionshanteringsystem. Det var även ett krav på projektet enligt kursen.

Git valdes för att det gamla systemet redan fanns tillgängligt på Github och för att kunden ville att vi skulle använda git, men också för att det har flera tekniska fördelar över till exempel svn som kräver en central server för att fungera. Detta leder ibland till att utvecklare väntar med att synkronisera sina ändringar för att inte behöva vänta på att svn skall utföra sina operationer mot den centrala servern. Överlag har vårt användande av git fungerat bra, även om det ibland har uppstått problem med interaktionen mellan Git och Visual Studio. Därför är det viktigt att alltid ha minst en projektmedlem som är bekant med git för att hantera dessa problem.

Det vi skulle kunna förbättra i relation till Git har främst med utbildning och organisation av branches. Först borde projektgrupper se till att alla medlemmar blir utbildade i Git i början av projektet så att de är bekanta med systemet när utvecklingsarbetet börjar.

För att få ut så mycket som möjligt av Git är det också en bra ide att ha flera branches istället för en enda som vi hade i majoriteten av projektet. Detta ledde till att det blev svårt att veta vad som ändrades vid någon given tidpunkt; en bättre organisation hade varit att dela upp arbetet på flera branches enligt någon av de många standarder som finns. På så sätt kan de olika utvecklingsgrupperna arbeta utan att skapa problem för varandra.

11.1.3 SQL

Vårt projekt fokuserade på att implementera nya applikationer till ett redan existerande system. I detta system fanns det en serverapplikation som använde sig av en SQL-databas för delar av sin datalagring. Vi tror att tanken bakom detta var att den data som skulle lagras är välstrukturerad och databaser är mycket bra på att lagra välstrukturerad data. Dessvärre saknar den data som Digimergo använder sig av relationer och den förändras endast väldigt sällan. Då försvinner många av fördelarna med databaser över enklare lösningar, som att lagra datan i en vanlig fil med något standardiserat format. Detta blir särskilt tydligt när man tar den ökade komplexiteten med att använda en SQL-databas. Exempelvis måste SQL-databasen startas separat innan applikationen vilket leder till många fler moment vid exempelvis testning. Källkoden blir även den mer komplex eftersom mjukvaran måste ansluta mot databasen och konvertera mellan olika typer av data när applikationen kommunicerar med databasen. Vår slutsats var att en SQL-databas helt enkelt var onödigt komplex för uppgiften, istället ersatte vi databasen med en XML-fil innehållande samma information. Fördelen med denna lösning är att det är enkelt att felsöka, installera och underhålla. Vårt råd är att inte direkt vända sig till databaser om uppgiften som skall lösas inte faktiskt behöver till exempel relationer eller säkra transaktioner på data.

11.1.4 Visual Studio

Visual Studio är det IDE som har använts under utvecklingen. Överlag har erfarenheterna varit positiva, bortsett från små problem som kan ha varit orsakade av användningen av git. Det positiva med VS har framför allt varit bra autokorrigering i form av IntelliSense, och bra verktyg för att refaktorera kod.

De negativa egenskaperna är inte överdrivet många. Ett problem som vissa medlemmar i projektgruppen har upplevt är bristen på kortkommandon och möjligheten att konfigurera nya, något som ofta fungerar bra i utvecklingsmiljöer på andra operativsystem än Windows.

Överlag är VS en mycket kompetent editor och dess bästa funktion IntelliSense har en positiv effekt på utvecklingshastighet och till viss del kodkvalitet.

11.1.5 WCF

I projektet har användningen av WCF (Windows Communication Foundation) varit ett återkommande inslag i utvecklingen. Då det ursprungliga systemet använde sig av WCF för kommunikation togs beslutet att även i detta projekt fortsätta med samma teknik. Innan en ordentlig grund och förståelse för projektets delar hade anskaffats verkade komplexiteten hos WCF och dess användning i systemet

vara hög. Efter bekantning med tekniken ändrades dock den uppfattningen, och valet av WCF för kommunikation mellan systemets olika delar ansågs som bra.

Den kanske största fördelen med användningen av WCF för nätverkskommunikation är dess mångsidighet. En WCF-tjänst kan konfigureras till att agera på många olika sätt, och om någon funktionalitet saknas är det i många fall enkelt att åtgärda genom ärvning och överlagring. Kort sagt kan en väl konfigurerad och väl implementerad WCF-tjänst användas i många olika typer av scenarion. En fördel med WCF gentemot andra tekniker är också möjligheten att separera kod i logiska avgränsningar, och automatisk kodgenerering med Visual Studio gör det enkelt att uppdatera klientsidans kod när serversidans kod har ändrats.

Just denna möjlighet till konfiguration och är också WCFs största problem. Att sätta sig in i WCF och förstå och alla dess funktioner kan vara en överväldigande och långsam uppgift. I projektet upplevdes även problem med att obskyra och svårtolkade felmeddelanden skapades av WCF, något som kunde ta många utvecklingstimmar att åtgärda. I jämförelse med andra tekniker och sätt att kommunicera över nätverk, exempelvis en lösning med sockets och XML/JSON i ett REST-api, är WCF svår att implementera ordentligt till en början, och det är därför svårt att motivera användningen i ett tidskritiskt projekt om utvecklarna saknar erfarenhet av tekniken.

11.2 Gruppens processerfarenheter

Här följer gruppens processerfarenheter.

11.2.1 Trac

Trac är det system som har använts i projektet för tidsrapportering och tickethantering. Våra erfarenheter av Trac är blandade. Det är onekligen ett kompetent system, och möjligheten till att använda plugins gör det väldigt flexibelt. Det är dock väldigt avskalat i sin grundläggande form, och användning av plugins är i princip nödvändigt för att använda Trac. Erfarenheten i projektet var att systemet först måste konfigureras med plugins, vilket visserligen är en ganska enkel process, men många plugins fungerar inte som förväntat (har buggar eller dåliga/inga anvisningar) vilket leder till slösad tid och frustration.

Med ovan problem nämnda bör de positiva egenskaperna nämnas, nämligen att ett fungerande Trac-system med rätt plugins för projektet kan spara mycket tid, ge bättre översikt över aktuellt arbete och åskådliggöra eventuella problem med arbetad tid. En lärdom från projektet är att det är viktigt att alla utvecklare tidigt lär sig använda Trac-systemet rätt, för att undvika senare problem och för att kunna använda den statistik som vissa plugins gör tillgänglig.

11.2.2 Delfimetoden

När vi skulle börja med tidsplanen hade vi flera olika idéer på hur detta kunde göras. Vår handledare tipsade oss tidigt om Delfimetoden. Vi beslöt oss för att undersöka denna metod och till slut kände vi att det var en tillräckligt väletablerad metod för att användas. Metoden visade sig fungera mycket bra men på grund av komplexiteten i projektet underskattades tiden för vissa aktiviteter.

En sak vi märkte när vi skulle använda metoden var att det var svårt att uppskatta hur mycket tid vissa aktiviteter skulle ta då många av oss inte hade gjort någonting liknande förut. Då blev det så att de personer som kände sig osäkra oftast höll med de som kände sig lite mer säkra på sin tidsuppskattning. Detta kan ha varit en anledning till att vissa aktiviteter underskattades.

11.2.3 Måndagsmöten

Vi tyckte att det fungerade bra med att ha ett återkommande möte varje måndag vid lunch för att hela tiden hålla sig uppdaterad om vad som hände i de andra delarna av projektet. "Fikapinnesystemet" fungerade bra eftersom vi tror att det ledde till att alla försökte komma i tid till mötena.

11.2.4 Handledarmöten

Vi tyckte det fungerade bra med återkommande handledarmöten. Det gav oss hela tiden en uppdaterad bild av vad som förväntades av oss. Dessa möten fungerade som en länk mellan projektet och kursen.

11.2.5 Användbarhetsstudier

Vi har lärt oss att användbarhetsstudier kan vara förvånansvärt användbara. När vi gjorde en prototyp av systemet tänkte vi inte alltid på samma saker som användarna. Därför tror vi att helhetsbilden av systemet blev tydligare och ökade i värde för kund.