HÖGSKOLAN
I GÄVLE

AKADEMIN FÖR TEKNIK OCH MILJÖ
Avdelningen för industriell utveckling, IT och samhällsbyggnad

Kraschrapporteringssystem för förbättrad återkoppling från användare

Mikael Ivarsson

2017

Examensarbete, Grundnivå (högskoleexamen), 15 hp
Datavetenskap
Dataingenjörsprogrammet

Handledare: Torsten Jonsson
Examinator: Carina Pettersson

Abstrakt

Detta arbete undersöker hur krascher i mjukvara kan rapporteras till ett utvecklingsföretag på ett enkelt men effektivt vis. Mjukvaruföretaget Monitor i Hudiksvall utvecklar affärssystem, och som all mjukvara under utveckling finns där buggar som emellanåt kan få programmet att krascha. Vid händelse av en krasch vill företaget gärna få reda vad som gick fel så det kan lösas. Det befintliga systemet analyserades och en prototyp utvecklades. En enkätundersökning som gjordes pekade mot att flera kunder drar sig för att rapportera när programmet fallerar på grund av att det är en tidskrävande process och för flertalet känns det inte meningsfullt. Att då ge möjligheten att automatisera hela processen kan hjälpa företaget öka antalet rapporter och med tiden få en tydlig bild över var de största problemområdena ligger. Istället för att låta användaren själv skicka e-post med en rapport togs en lösning fram som skickar via HTTP. Denna teknik tillåter att rapporter enklare kan sorteras och användas för att föra statistik över de största problemområdena i programmet. I det gamla systemet öppnas ett nytt supportärende i och med varje rapport som skickas in, men den utvecklade prototypen föreslår en lösning som låter kraschinformation rapporteras med ett enkelt knapptryck eller, om så kunden önskar, öppna ett supportärende. Lösningsförslaget som presenteras i arbetet skulle på sikt kunna minska på antalet timmar som läggs ner på att manuellt analysera buggrapporter.

Nyckelord: kraschrapportering, återkoppling, C#

Innehåll

1 Introduktion

Mjukvaruutveckling är en ständigt fortgående process. Att skriva och släppa en fungerande mjukvara är en process i sig, men ingenting är evigt. Mjukvara behöver förr eller senare underhållas och utvecklarna behöver användarnas hjälp. Dock är inte all hjälp alltid bra hjälp då användare inte alltid bidrar med den informationen utvecklarna behöver.[1]

Att hitta och åtgärda buggar är det utvecklare lägger allra mest tid på när det kommer till mjukvaruutveckling. B Boehm och V R Basili [2]menar på att 70 procent av den totala kostnaden att utveckla mjukvara går åt till att underhålla densamma. För att effektivisera tid, och kostnader, krävs en tydlig beskrivning när något går fel. En redogörelse för vad som hände och av vilken orsak. Användaren av programvaran kan rapportera till företaget som utvecklar den att någonting gick snett som fick programmet att krascha. Detta görs ofta i form av ett inbyggt system som samlar in relevant data och skickar in till utvecklarna. Många gånger ombeds användaren fylla i ytterligare information, så som till exempel vad vederbörande gjorde som ledde fram till kraschen.

När en kraschrapport väl kommit in ska den sedan analyseras och bestämmas var någonstans den ska hamna, vilken prioritet den ska ha och vilken utvecklare som ska få ta hand om att lösa problemet. Att sortera bland buggrapporter, sätta prioritet och tilldela utvecklare är en process som, bland programmerare, kallas triage. [3] Uttrycket härstammar från sjukvården där triage är processen då patienter sorteras och prioriteras efter hur pass allvarligt läget är.[4]Triage är vanligast på akutmottagningar, och i händelse av katastrofer, där många patienter måste sorteras snabbt. Liknelsen till en databas med ett stort antal buggrapporter är således inte så långsökt.

Större system består av en ofantlig mängd källkodsfiler, och att hitta vilken fil det är som fallerar är inte alltid det enklaste.[5] Detta kan sätta krav på felhanteringssystemet att samla in korrekt, relevant, information för att förenkla sökandet.

1.1 Monitor ERP System AB

Arbetet utfördes hos Monitor ERP System AB [6] i Hudiksvall. Det är ett mjukvaruföretag som utvecklar affärssystem åt små och medelstora, tillverkande, företag. Affärssystemet innehåller allt som ett företag kan tänkas behöva, från lagerhantering till tidsrapportering. Monitor arbetar med två större versioner av affärssystemet. En äldre version - Monitor G4 (kort för Generation 4) – och en ny generation - G5. I G4 finns en kraschrapporteringsfunktion implementerad, men en liknande finns ännu inte i G5.

1.2 Syfte

Syftet med detta arbete var att utreda ett sätt att kunna samla in bättre och mer värdefull återkoppling från användare angående fel och brister i systemet. I det ingick att undersöka huruvida kunder drar sig för att skicka in felrapporter av olika anledningar, samt hur man kan få in mycket - och användbar - information så enkelt som möjligt för användaren. Målet med arbetet var att få fram ett förslag på hur ett rapporteringssystem till Monitor kan utformas för att höja kvaliteten på kraschrapportering samt göra det enklare att prioritera buggar.

1.3 Problemformulering

Arbetet fokuserade kring en huvudsaklig frågeställning: Hur kan en kraschrapporteringsfunktion se ut som är optimerad både för användare och utvecklare?

Med optimerad menas mindre ”jobb” för användaren, samtidigt som det inkluderas mycket värdefull information till utvecklare. Ska man överväga möjligheten att endast ha automatisk rapportering, och således förlora användarens input? Kanske en kombination? Hur kan ett sådant system se ut till en programvara som Monitor?

1.4 Teoretisk bakgrund

Återkoppling från slutanvändare är det som håller mjukvara vid liv,[7] och Monitor är inte ett undantag till detta. Men att utvecklare får in ett stort antal felrapporter behöver inte betyda att de alltid är till någon hjälp.[8] Om det finns buggar som drabbar många men bara rapporteras av få finns en risk att de inte blir prioriterade att åtgärda.

S. Breu et al. [8] gjorde en studie om buggrapportering på Eclipse och Mozilla-projekten. De analyserade buggrapporter och kom fram till att majoriteten av rapporter som kom in saknade relevant information för att utvecklarna skulle kunna återskapa problemet. Några exempel på utebliven information var vilket operativsystem som kördes på maskinen, information om vad användaren gjorde vid kraschtillfället eller att det saknades en stacktrace för att se var problemet låg. Detta hindrade utvecklare att snabbt kunna lokalisera och lösa problemen.

N. Bettenburg et al. [9] bekräftade föregående författares studie. En överväldigande majoritet av bristerna i buggrapporter var just undermålig och utebliven information som krävdes för att återskapa och hitta problemen.

Steven Davies och Marc Roper [10] har också undersökt open source-projekt, Eclipse, Mozilla och Apache, för att ta reda på vad buggrapporterna som kommer in oftast innehöll. Rent logiskt innehöll de flesta en kort beskrivning av vad som hände i programmet, vilket kan vara hjälpsamt om ytterligare information tillhandahålls, vilket inte alltid var fallet. Buggrapporter som endast talar om hur felet såg ut är väldigt svåra att återskapa, och således svåra att lösa. Om rapporten, å andra sidan, skickades tillsammans med en bifogad stacktrace, en kort text som berättar vilka steg som ledde upp till felet samt en beskrivning av vad det förväntade beteendet var, blev ärendet plötsligt mer troligt att bli åtgärdat.

I ett stort projekt, som till exempel Mozilla Firefox, är det desto viktigare att buggrapporter är välformulerade. P. Hooimeijer och W. Weimer [11]tar upp ett exempel om en bugg i webbläsaren som uppmärksammades mellan 2003-2006. När en fil som laddades ner var större än två gigabyte kunde det ibland visas att filen hade en negativ storlek. Den första rapporten som kom in var inte tillräckligt informativ för att utvecklare skulle kunna återskapa problemet, och buggen förblev kvar i systemet fram till 2006 efter att över 140 rapporter hade kommit in som beskrev buggen - som till slut kunde fixas.

När antalet rapporter blir allt större ökar således risken för dubletter – användare som rapporterar om samma problem. Detta kan göra att antalet rapporter är mycket större än det behöver vara. Större projekt får ofta in fler rapporter än vad som hinner hanteras av människor. Då kan något behövas som grupperar dubletter och på så vis minskar antalet markant. F .Thung et al. [12] diskuterar hur DupFinder hjälper till med just detta. DupFinder är ett verktyg som går igenom registrerade buggrapporter och listar potentiella dubletter som snabbt kan kontrolleras och grupperas.

1.5 Avgränsningar

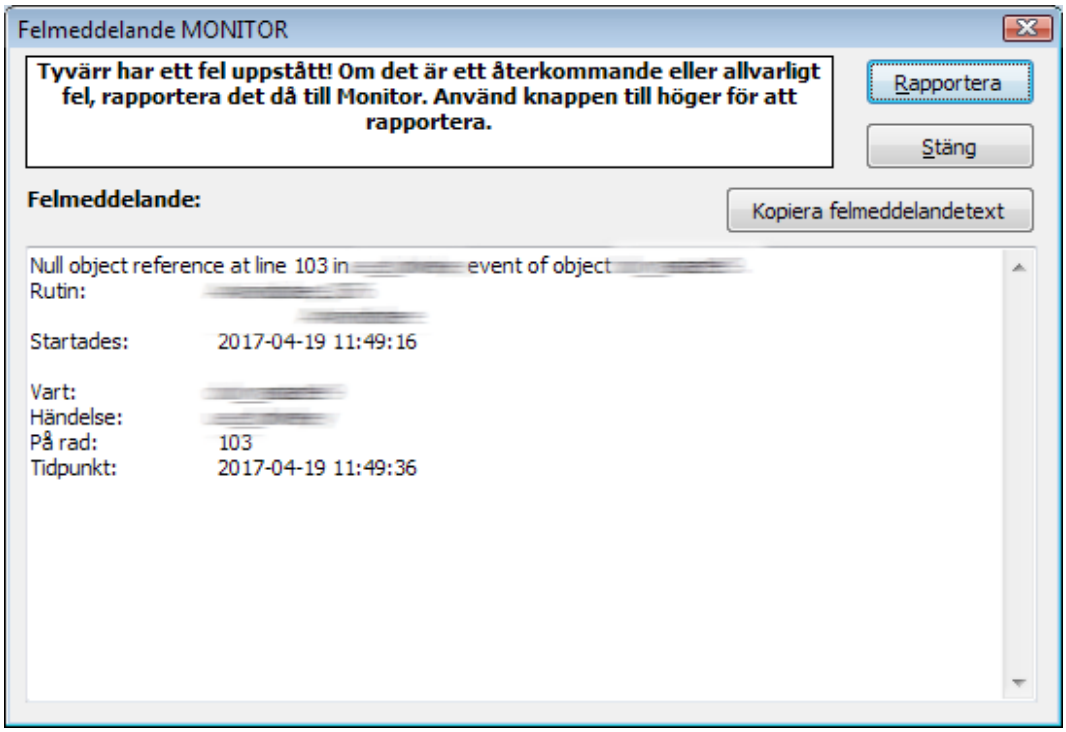
Arbetet avgränsas till att ta fram en prototyp som visar på hur buggrapporteringsapplikationen skulle kunna se ut – ett konceptförslag med begränsad funktionalitet - och argumentera för varför det kan fungera bättre än det som finns i Monitor G4. I ett sådant system finns flertalet säkerhetsaspekter att överväga, men dessa togs ej i beaktande i utvecklandet av prototypen. Ett exempel på en sådan aspekt hade varit att upprätta en säker anslutning mellan klient och extern databas med HTTPS samt kryptera sänt data.

Prototypen som togs fram var självständig och inte integrerad i Monitor på något vis. Därför skrevs ett litet exempelprogram, som platshållare för Monitor, som designades med flera brister som får programmet att krascha för att visa på hur kraschrapporteringen skulle kunna fungera i Monitor.

2 Befintliga kraschrapporteringsfunktioner i Monitor

2.1.1 Kraschrapportering i G4

I Monitor G4 finns en guide som startar när programmet kraschar. Det låter användaren fylla på med information som är väsentlig för att utvecklarna ska kunna utreda och fixa orsaken till kraschen. Vid händelse av att Monitor kraschar möts användaren av en dialogruta, Figur 1, som visar en sammanfattning av vad i programmet som orsakade kraschen. Därefter får användaren gå igenom några steg och fylla i, till exempel, vad som hände precis innan kraschen och om användaren upplevt kraschen tidigare. I det sista steget kan användaren sedan välja att skicka in den genererade kraschrapporten via e-post till en supportbrevlåda. Alla dialogrutor i de olika stegen visas i Bilaga A.

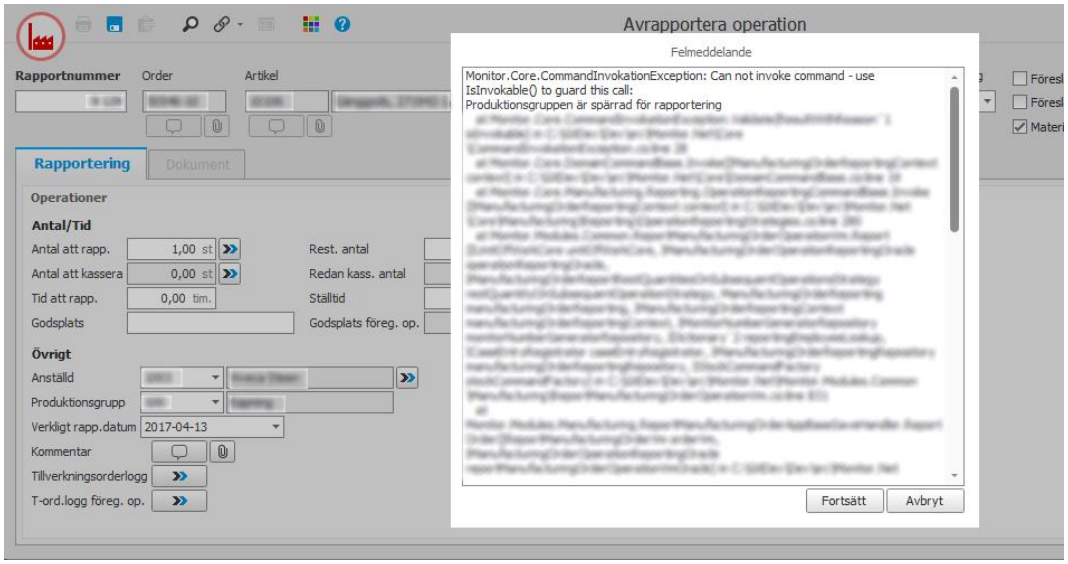


Figur 1- Felmeddelande som dyker upp vid händelse av krasch i Monitor G4

Monitor G4 är skrivet i programmeringsspråket PowerScript i verktyget PowerBuilder. Där finns ett event som heter `SystemError` som ger all data som behövs för att felsöka, så som radnummer, rutinnamn etc.

2.1.2 Kraschrapportering i G5

I Monitor G5 dyker endast en diagonalruta upp, som den i Figur 2, med ett felmeddelande som kan verka oerhört kryptisk. Härifrån får vederbörande själv ta en skärmbild eller kopiera felmeddelandet och manuellt maila in till Monitor som tar hand om ärendet därifrån. Möjliga lösningar har undersökts som skulle kunna ersätta detta och på så vis förbättra kvaliteten på återkopplingen.



Figur 2 - Felmeddelande som visas i Monitor G5

I programmeringsspråket C#, som G5 är skrivet i, kan det vara lite svårare att få ut den tekniska informationen som behövs till debugging. För att kunna få information om exakta filnamn och radnummer där ett exception kastats måste en PDB-fil (Program DataBase) skickas med vid bygget av programmet. Denna fil innehåller mycket information om själva programmet, och mycket av denna information kan vara känslig, då den visar fullständiga filsökvägar, och kan användas för att enklare hacka mjukvaran. Det är därför inte rekommenderat att filen skickas med i releaseversionen av mjukvaran, utan endast används vid felsökning. PDB-filen används när kraschen återskapas hos företaget varpå man då kan få reda på exakt var någonstans det gick fel – filnamn och radnummer.

3 Metod

Arbetet utfördes på plats hos Monitor i Hudiksvall. I och med detta fanns utvecklare nära tillhands under hela arbetet. Tanken var att bygga in en prototyp direkt i Monitor, men Monitor och dess kodbas var ett alldeles för stort system att sätta sig in i. Istället bestämdes att en självständig prototyp, ett konceptförslag, skulle utvecklas.

Det första som behövde göras var att undersöka hur rapporteringsfunktionen såg ut i Monitor och identifiera de aspekter som kunde förbättras. Krascher tvingades fram för att kunna visa hur det ser ut för användaren när affärssystemet falerar. Samtliga dialogfönster som visades dokumenterades och analyserades. Samma sak gjordes sedan på G5, som inte hade en komplett, inbyggd, funktion för kraschrapportering. När Monitor G5 drabbades av en krasch visades endast en diagonalruta med ett kryptiskt felmeddelande, vilket användaren själv fick kopiera och manuellt maila in till företaget.

3.1 Enkätundersökning

För att ta reda lite mer om brister i det gamla systemet gjordes en enkätundersökning med ett antal användare av Monitor G4. Syftet med enkäten var att få en bild av huruvida kunder drar sig för att rapportera och vilka orsaker som kan ligga bakom. Den skickades till en liten grupp på ca 10 personer, då den inte var tänkt att generera absolut statistik, utan endast ge ett litet hum om åt vilket håll det lutade.

3.2 Prototyp

En prototyp skulle byggas utifrån svaren från enkätundersökningen, men också från egna analyser och slutsatser dragna från de befintliga rapporteringsfunktionerna i Monitor G4 och G5. Denna prototyp var tänkt att bli som ett konceptförslag för hur en framtida buggrapporteringsfunktion skulle kunna se ut. Den skulle bestå av en applikation som sparar en kraschrapport till databas och en annan som hämtar och presenterar data. Prototypen byggdes för att visa på hur funktionaliteten skulle kunna se ut.

3.3 Verktyg

Monitor G5 utvecklas i programmeringsspråket C# i Visual Studio 2015. Det föll sig då naturligt att utveckling av en prototyp skedde i samma miljö. Buggrapporteringprototypen utvecklades med Windows Forms, ett ramverk inbyggt i Visual Studio.

C# är ett objektorienterat programmeringsspråk. Det är således väldigt likt – till exempel – Java, som varit det huvudsakliga programmeringsspråket under utbildningen på Högskolan i Gävle.

Till databashantering valdes MySql via Xampp.[13] Xampp är en programvara som levereras med både databashanterare och Apache, så det gick att sätta upp en virtuell server att arbeta mot under utvecklingen av programvaran.

För att kommunicera med den externa databasen skulle en liten PHP-applikation skrivas som lades på servern och som hade i uppgift att populera databasen. Denna backend-lösning utvecklades i IDE:n PhpStorm av JetBrains s.r.o. [14]

Användarinformation som användes till att simulera inloggade användare för att automatiskt fylla i användarinformation lagrades i en Microsoft Access-databas direkt i Visual Studio.

Till enkätundersökningen användes Google Forms. [15] Detta verktyg är mångsidigt, och resultat visas i realtid allteftersom svar kommer in.

4 Genomförande

4.1 Enkätundersökning

Enkätundersökningen skickades per e-post till användare av Monitor G4. En maillista med tio kunder tillhandahölls, och en länk till Google Forms-enkäten skickades till dessa. De som fick enkäten skickade till sig ombads att dela den med kollegor på sina arbetsplatser, då endast en person per företag kontaktades.

E-postmeddelandena som skickades ut till användarna innehöll en kort beskrivning av arbetet som utfördes. Det förklarades att det var ett examensarbete, samt lite kort om varför undersökningen vad nödvändig för fortsättningen av detsamma (Bilaga B).

Enkäten bestod av frågor med flersvarsalternativ - förutom en ”Annat” där de kunde få svara med egna ord om inget av alternativen stämde. Frågorna som ställdes hade att göra med huruvida kraschrapporteringsfunktionen i Monitor G4 använts samt varför eller varför inte. När skickades en rapport in och när hoppades det över? Hur pass utförliga är de när de väl skickar in en rapport? Detta för att få fram några anledningar till uteblivna rapporter, som sedan kan försöka åtgärdas i prototypen. Hela enkäten finns i sin helhet i Bilaga B.

4.2 Monitor-simulator

Eftersom denna prototyp blev helt separerad från Monitor behövdes ett exempelprogram som då skulle kunna motsvara Monitor-klienten. Detta program bestod av ett fönster som i sin tur kunde öppna mindre, inre, fönster, motsvarande Monitors rutiner. Dessa små rutiner byggdes så att de kastade olika exceptions som inte togs omhand av programmet, som därmed triggade rapporteringsprogrammet. Användaren gavs då möjligheten att antingen skicka in ren statistisk och anonym data eller öppna ett supportärende.

4.3 Fånga upp en krasch

Programmet kraschade när ett exception kastats som inte hanterades av koden, till exempel i ett `try/catch`-block. I `Main()`-metoden, den metod som körs när programmet exekveras, hade ett par metoder implementerats som fångade upp när ”Unhandled exceptions” och ”Thread exceptions” kastats. Detta är alltså exceptions som hindrar programmet från att fortsätta exekveras. Då anropades en metod som i sin tur körde rapporteringsprogrammet, med exception-objektet som inparameter.

4.4 Samla in tekniska data

När rapporteringsdialogen startat samlades all relevant teknisk data in från exception-objektet. Därifrån kunde programmet extrahera metodnamn, stacktrace och namnet på det exception som kastades. En statisk klass implementerades som höll koll på vilket fönster som var aktivt vid kraschtillfället, så det kunde dokumenteras. Att veta vilket fönster, eller rutin, som var aktivt när felet inträffade kan hjälpa utvecklare i utredningen med hur felet kan hittas och åtgärdas. Det sista som lades in i Exception-objektet var information om tid och datum. Detta var all data som skickas till databasen om användaren valt det automatiska alternativet. Om ett supportärende öppnats behövs ytterligare information.

4.5 Samla in användardata

Till prototypen utvecklades en enkel inloggningsfunktion. Innan huvudprogrammet kunde användas måste en registrerad användare logga in, eftersom det skulle simulera en licensierad programvara. Till programmet var kopplat en Microsoft Access-databas, i vilken användarinformation lagrades som sedan automatiskt fyllde rapportformuläret. Användaren gavs sedan möjligheten att med egna ord beskriva vad det var som ledde upp till felet, så att felet skulle kunna återskapas för effektiv debugging.

4.6 Spara rapport i extern databas

När all väsentlig information samlats in var nästa steg att skicka den till en extern databas. När användaren väljer att skicka rapporten kapslas all data in i ett `NameValueCollections`-objekt. Detta objekt laddades sedan upp till en virtuell server som med hjälp av en liten PHP-applikation överför all data och lagrar i databasen.

4.7 Säkerhetsaspekter

Som nämnt under Avgränsningar fanns här vissa säkerhetsaspekter att ta hänsyn till, till exempel användarintegritet. Då prototypen blev ett konceptförslag har inga säkerhetsåtgärder vidtagits, så all information som skickas sänds okrypterat. I ett skarpt system bör användaren också först ges möjlighet att bevilja att Monitor samlar in information om användaren, och även tala om vad det är för information som samlas in, innan någonting skickas till Monitor. Sådana saker beaktades inte i utvecklingen av prototypen.

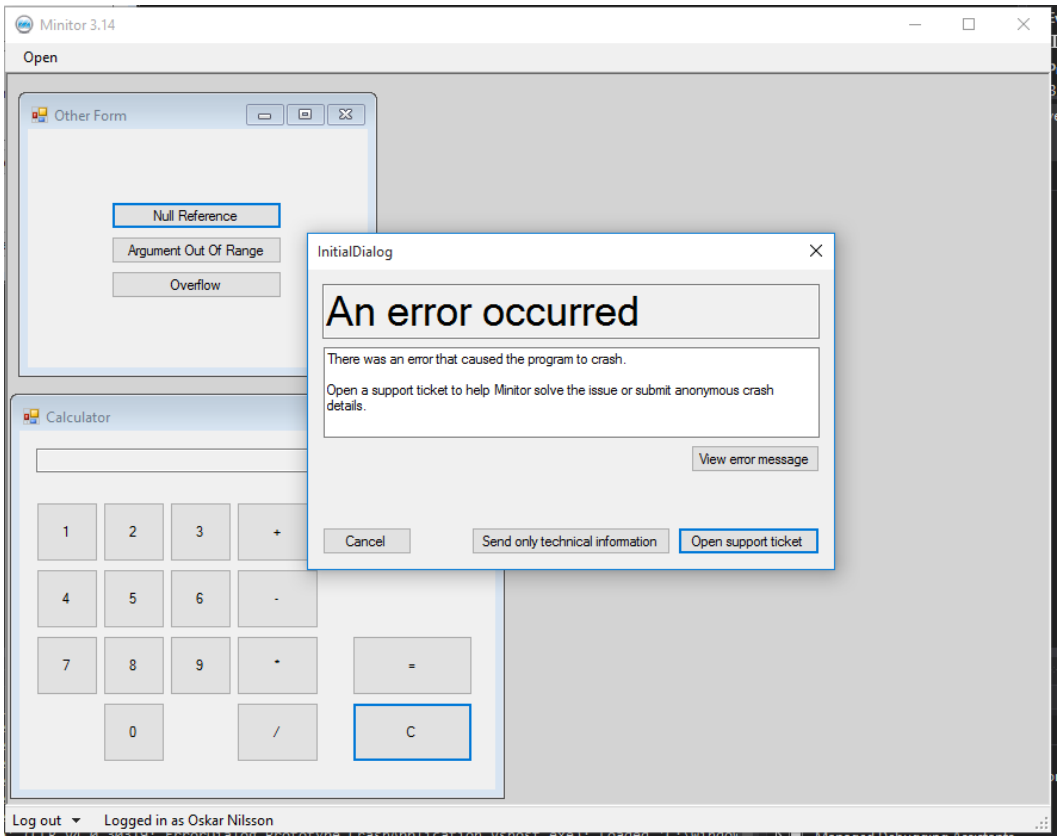
4.8 Användargränssnitt

4.8.1 Logga in i Minitor

Det utvecklade testprogrammet som agerade platshållare för Monitor döptes till Minitor. För att kunna använda Minitor behöver en användare logga in med befintlig användare eller registrera en ny. När en användare loggat in kommer rapportformuläret automatiskt fyllas på med användarinformationen vid öppnandet av ett supportärende.

4.8.2 Krasch

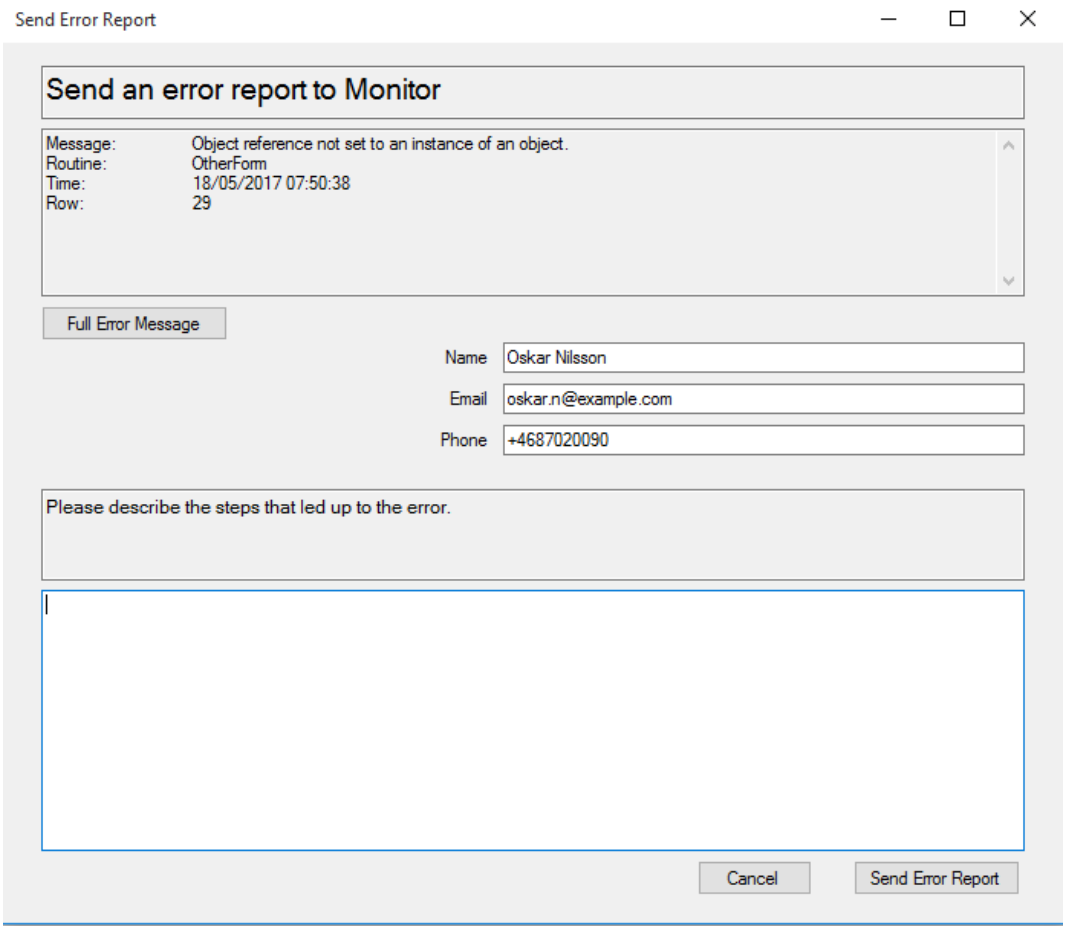
Efter inloggning kunde Minitor börja användas. Figur 3 visar hur prototypprogrammet kunde se ut precis efter en krasch. Två rutiner är öppnade, och en av dem har orsakat en krasch som aktiverar kraschdialogen. Användaren ges ett val att antingen skicka in anonym information eller öppna ett supportärende. Om anonym, teknisk, information valdes skickades data iväg och programmet avslutades sedan.



Figur 3 – Testprogrammet Minitor har kraschat och aktiverat kraschrapporteringsprototypen.

4.8.3 Öppna supportärende

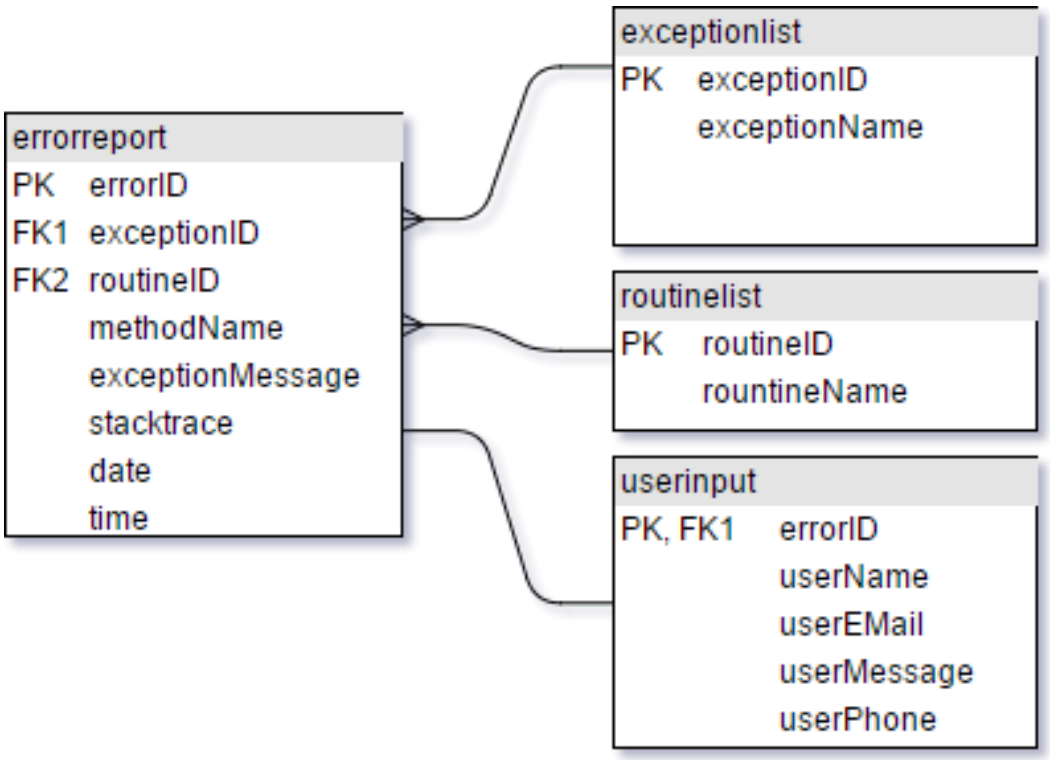
Vid valet att öppna ett supportärende öppnades ett ny dialogfönster. Figur 4 visar hur textrutorna automatiskt fyllts i med användarinformation samt en kort sammanfattning av felmeddelandet. Användaren ombads sedan att redogöra för hur felet inträffade innan rapporten skickades in.



Figur 4 - Dialogen för att öppna ett supportärende

4.9 Databas för lagring av kraschrapporter

En databas i MySql byggdes för att lagra de kraschrapporter som skickades in. I detta projekt sändes de till en virtuell server via HTTP. En liten PHP-applikation tog emot förfrågan och plockade ut den skickade informationen och la in allt i databasen. Databasen var byggd att vara så enkel som möjligt till denna prototyp. I den kunde all väsentlig information om kraschen sparas, och hämtas senare för att analyseras. I databasen skulle det även kunna tänkas att spara annan teknisk information så som vilken version av Monitor som kördes, vilket operativsystem det kördes på etc, men det lämnades därhän då det inte var relevant i prototypsyfte.



Figur 5- UML-diagram över databasen

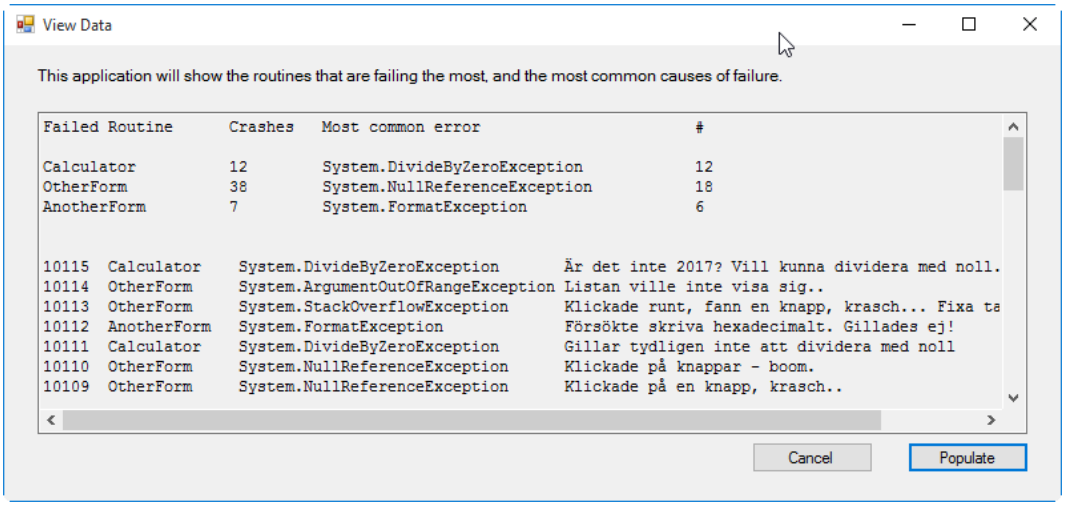
Databasen byggdes med fyra tabeller (Figur 5) för att få den så normaliserad som möjligt. Eftersom ett exception kan kastas på fler ställen och av olika anledningar lades endast ett ID till ett exception i ”errorreport”-tabellen, som då blev främmande nyckel, detta för att minska redundans. Dock visas inte alltid samma exception-meddelande varje gång ett exception kastas, så det meddelandet fick hamna i en egen kolumn i ”errorreport”-tabellen. Det medför dock en del dubbellagring, men i prototypsyfte valdes det att behållas.

Samma gällde med ”routinelist”. Då det finns ett begränsat antal rutiner läggs även de i en egen tabell med ett ID, som en främmande nyckel i ”errorreport”-tabellen. I både ”exceptionlist”- och ”routinelist”-tabellerna läggs nya poster till allteftersom om de inte redan finns i tabellen och tilldelas ett unikt ID.

”Userinput”-tabellen var tänkt för just det, användarinput. Medan ”errorreport” rymmer den tekniska informationen, läggs användarinformationen i denna. Varje gång en rapport skickas in skapas ett unikt ID i ”errorreport”. Detta ID blev främmande nyckel i ”userinput” för att kunna koppla samman de båda tabellerna.

4.9.1 Lokalisera problemområden

Till prototypen byggdes ännu en liten applikation som hämtar ut data från databasen. I prototypsyfte gjordes den relativt primitiv. Den hämtar endast exceptionnamn och användarens meddelande, samt vilken rutin som kraschade. I fönstret listas hur många gånger varje rutin kraschat samt vilken orsak som var den vanligaste för varje rutin. Efter tester som gjorts på prototypen visas ett exempel på hur statistiken presenteras i Figur 6.



Figur 6 – Statistik som visar vilka fel som är vanligast i rutinerna.

I prototypen finns tre rutiner, och de har kraschat av olika anledningar. Som exemplet visar har OtherForm-rutinen kraschat 38 gånger, och den vanligaste anledningen var att ett `NullReferenceException` kastats 18 gånger. Det som listas inunder är samtliga krascher som fanns registrerade på databasen.

5 Resultat

5.1 Enkätundersökning

Enkätundersökningen besvarades av 17 kunder som använder sig av Monitor G4. Utifrån resultatet kan det urskiljas en trend som lutar åt viljan att automatisera rapporteringsprocessen. Flera av respondenterna sa att de ofta drar sig för att rapportera. Några exempel på anledningar var:

- Användaren tycker inte det är meningsfullt.
- Kraschen har inte ansetts tillräckligt allvarlig för att rapporteras.

Flera av deltagarna önskade att processen automatiserades så att de inte behövde slösa tid på att öppna ett supportärende varje gång något går fel i programmet. Resultatet finns sammanställt i Bilaga C, och en mer djupgående analys av enkätundersökningen gjordes i Diskussionskapitlet.

5.2 Prototyp

5.2.1 Monitor-simulator

Detta arbete resulterade i en fungerande prototyp som kraschar och aktiverar en kraschrapporteringsfunktion. Programmet startade med att man får logga in på ett litet exempelprogram som utvecklades till att simulera hur Monitor är uppbyggt. Det bestod av ett huvudfönster som är själva huvudprogrammet. I detta fönster kunde sedan flera mindre fönster - så kallade rutiner - öppnas för att utföra olika uppgifter. Rutinerna i prototypen var programmerade som så att de var benägna att krascha i tid och otid, varpå en dialog för kraschrapportering öppnades.

Ett exempel på en rutin var en liten kalkylator. Denna applikation var programmerad på det viset att det inte fanns någon kontroll som tittade om användaren försökte dividera med noll. Om detta skulle ske kastades ett exception som inte hanterades av programmet utan orsakar då programmet att krascha.

5.2.2 Kraschrapporteringsguide

Det första som visades var ett fönster som låter användaren själv välja om ett supportärende skulle öppnas eller om bara anonym, teknisk, information ska skicka in.

Statistisk information som skickades in bestod enbart av tekniskt information, så som vilken rutin som var drabbad, metoden som orsakade kraschen samt en stacktrace. Denna information kunde sedan lagras och användas för att i längden se var någonstans i programmet de flesta felen har en tendens att förekomma. Detta alternativ kunde väljas om det är ett återkommande fel eller om användaren helt enkelt inte ville öppna ett nytt supportärende; informationen kunde ändå skickas in och användas av Monitor anonymt.

Om användaren istället valde att öppna ett supportärende ombads användaren redogöra för stegen som ledde upp till kraschen. Användarinformation kunde skickas med för att kunna ges återkoppling och kunna bli kontaktad om några oklarheter uppstod. Användaren kunde även öppna en dialogruta som presenterade all data som skulle skickas in i rapporten, ifall vederbörande ville veta säkert vad det var för information som skickades.

5.2.3 Felstatistik

Till prototypen byggdes ett litet ”statistikprogram” som listade de vanligaste anledningarna till krascher. Denna applikation blev mer ett ”proof of concept” då den knappt visade sig vara till någon större nytta i debugging-avseende, men i prototypsyfte visade den hur data kunde hämtas och listas på ett informativt vis. I en skarp release av ett sådant system skulle mer information kunna hämtas ut, däribland metodnamn, för att ge en tydligare bild. En annan del av applikationen skulle sedan kunna implementeras som listar varenda enstaka rapport och eventuellt sorterar dem efter rutiner eller andra sökvillkor. Återigen, i prototypsyfte begränsades funktionaliteten endast till att lista de vanligast kastade exceptions i respektive rutin.

6 Diskussion

6.1 Enkätundersökningen

Svaren från enkätundersökningen visar att flera önskade att rapporteringprocessen kunde automatiseras. Om denna möjlighet ges ökar förhoppningsvis chansen för att mer återkoppling kommer in till företaget, som då får mer data att utgå från när det kommer till buggprioritering. Ett exempel: Ett supportärenden öppnas som handlar om en krasch i en specifik rutin. Om andra användare drabbas av en liknande krasch i samma rutin, och inte vill öppna ett nytt ärende, kan de istället meddela företaget att något hände med endast ett knapptryck. På så vis kan ärendet få en högre prioritet i och med att andra användare troligtvis upplevt samma bugg.

Från enkätundersökningen finns också en antydan till att rapporter främst skickas in när problemet ses som tillräckligt allvarligt. Det kan således tyckas att det blir onödigt att öppna ett supportärende om det endast är ett litet fel som orsakar en krasch, eller något tillfälligt som kanske beror på den egna hårdvaran. Då kan det antas att om användarna ges möjligheten att meddela företaget, med endast ett knapptryck, att någonting hänt blir fler benägna att vilja rapportera. I nuläget hade företaget aldrig fått reda på att denna bugg existerade om ingen rapporterat det.

En anledning till att antalet deltagare i enkätundersökningen inte var så stort berodde på att det endast skulle bekräftas att det fanns kunder som eventuellt hoppar över att rapportera, samt för att hitta några anledningar till varför. Som nämnt tidigare i rapporten var det aldrig ett mål att få in data som, till exempel, tillät djupare analyser som kunde visa på hur stor andel av kundkretsen som gör si eller så. En generell undersökning kunde förvisso genomförts, som enbart handlade om kraschrapportering överlag och inte specifikt för Monitor, och således enklare kunnat få fler respondenter. Men arbetet handlade dock om just Monitors rapporteringsfunktion och då kändes det bäst att genomföra en undersökning med dem.

6.2 Automatisk och semiautomatisk rapportering

Ett av målen var att göra kraschrapporteringen så enkel som möjligt, men ändå ge användare möjlighet att med egna ord berätta hur problemet skulle kunna återskapas för att sedan rättas till. Utan denna information finns en stor risk att alltför många buggar förblir olösta eller, helt enkelt, dolda. Med helautomatisk rapportering finns större chans att antalet rapporter ökar, men på bekostnad av utebliven information. Då ingen användarinput tillhandahålls med en sådan lösning uteblir hjälpen till att återskapa kraschen, vilket knappast hjälper utvecklarna. Därför valdes en metod som ger användaren valet att antingen skicka in anonym, teknisk data eller öppna ett supportärende. Tanken är då att under tiden som antalet rapporter växer kommer mönster börja formas där man kan identifiera de största problemområdena i programmet.

Den överhängande risken med denna lösning som måste tas i beaktande är då att alla användare väljer det enkla alternativet, och kompletta kraschrapporter utebliver näst intill helt. Detta kanske kan förebyggas genom att slumpmässigt fråga en extra gång om inte användaren vill skicka en fullständig rapport till företaget. Alternativt, om programmet är välutvecklat kanske det kan identifiera att det är första gången programmet drabbats av ett specifikt fel och mer eller mindre tvingar användaren att skicka in fullständig rapport.

6.3 Skicka rapport över webb istället för e-post

Att skicka data över webb istället för e-post har många för- och nackdelar. En uppenbar fördel är att det är väldigt enkelt för användaren att bara behöva trycka på en knapp för att skicka iväg. Men detta på bekostnad av säkerhet och eventuellt användarintegritet. Detta område om IT-säkerhet är enormt - och således utanför arbetes omfattning - och valdes därför att inte tas i beaktande.

Detta tillvägagångssätt anses vara en stor förbättring jämfört med att skicka via e-post. På detta vis sparas allt direkt i databas, och kan kategoriseras därefter, utan att en människa manuellt behöver göra det. Om detta implementeras på ett lämpligt vis i ett system kan oerhört mycket tid och arbetskraft sparas.

Även utifrån resultatet från enkätundersökningen går det att argumentera för denna teknik. Om processen automatiseras och skickas direkt till en server på detta vis undviker man det extra steget med e-post, och således ännu mer tid sparas på rapporteringen. Användare behöver inte ställa in SMTP-inställningar och behöver inte krångla med e-post-klienter.

6.4 Utebliven information

När en applikation utvecklad i C# byggs och släpps till slutanvändare exkluderas en PDB-fil. Denna ProgramDataBase-fil skapas när programmet byggs i Visual Studio i debugging-syfte. Den innehåller information som kan visa exakt var någonstans felet skedde, vilken källkodsfil samt rad. Dock kan informationen i filen även användas för mer illvilliga ändamål, och av säkerhetsskäl inkluderas inte den filen när applikationen släpps. När en rapport kommer in finns således bara den informationen som är på ytan, till exempel metodnamn och rutinnamn. Under arbetet gjorde många försök att ta sig runt detta, men förgäves. Det blir sedan upp till utvecklare eller testare att försöka återskapa felet - då med denna PDB-fil - som då kan visa exakt var felet ligger. Det optimala hade varit att hitta något relativt enkelt sätt att kunna inkludera radnummer och eventuellt filnamn utan att exponera för mycket. Det hänger mycket på att användarna tydligt kan återge stegen för att återskapa kraschen. I felmeddelandet i Figur 4 visas vilken rad kraschen skedde på, men denna information visas alltså endast om PDB-filen finns tillgänglig. För kunden skulle där stå ”Row: 0” istället. Alternativt tas den raden bort helt, då utvecklare kan få radnumret ur stacktrace vid återskapandet av kraschen.

Annan information som kan vara till hjälp, som inte inkluderades i prototypen, var till exempel information om systemet som Monitor körs på. Vilket operativsystem körs, vilken version av Monitor körs samt annan relevant teknisk information. Arbetet hade dock blivit alldeles för omfattande om all denna data skulle behövas samlas in.

6.5 Avgränsningar och vidareutveckling

6.5.1 Ytterligare användarinput

Något som heller inte kommer med i prototypen är huruvida användaren upplevt det rapporterade felet tidigare, eller om det är första gången, och detta kan då leda till att många rapporter blir dubletter. Det kan dock tänkas att om den tekniska informationen som samlas in analyseras på lämpligt sätt kan det antas att fel kan vara sammanhängande och på så vis grupperas som ”samma”. Tanken med konceptet är att det ska kunna utvecklas vidare och lägga på sådan funktionalitet som minskar antalet timmar en människa ska behöva lägga ner på bugg-triage.

6.5.2 Säkerhet

Då kraschrapporter kan innehålla viss känslig, personlig, information önskas således en säker transport av datat. Därför vore en HTTPS-anslutning optimal för att vara säker på att anslutningen är säker. Detta hindrar även vem som helst som har tillgång till internet att skicka spam till servern, då autentiering måste ske innan data kan skickas.

6.5.3 Statistik

Dessa buggrapporter med all data sparas till en databas, vilket gör att de kan kategoriseras, sorteras, listas etc. Vidareutveckling av detta koncept kan bestå i mer utförliga klienter som kan presentera felstatistik på ett relevant vis. Prototypen visar endast ytliga fel, men det kan, som sagt, utvecklas till något mer komplicerat system i framtiden.

6.5.4 Spara lokala loggar

Något som lämnades därhän var möjligheten att lokalt logga kraschrapporter om internetuppkopplingen ligger nere. Detta är något som redan finns implementerat i G4, vilket tillåter utvecklare att, till exempel, via en fjärranslutning öppna och läsa i en loggfil.

7 Slutsatser

När arbetet började var utgångspunkten rapporteringsfunktionen i Monitor G4 och dess brister. Den överhängande svagheten som finns i systemet, vad gäller utredning och prioritering av buggrapporter, är att för varje rapport öppnas ett nytt supportärende. Detta kräver att en människa måste analysera och tolka varenda rapport som kommer in och tilldela den till rätt utvecklare. Vad prototypen föreslår är en mer kvantitativ lösning. Supportärenden kan fortfarande öppnas, men med ett enkelt knapptryck kan företaget meddelas anonymt när något går fel i programmet, vilket loggas och kan analyseras.

En lösning lik den som åstadkoms med prototypen kommer, enligt författaren, kunna öka kvaliteten på återkopplingen Monitor får in. En fullständig implementation av konceptet kan vara komplicerad och tidskrävande men kan med tiden visa sig vara värt besväret.

Referenser

[1] T. D. Sasso, A. Mocci and M. Lanza, "What makes a satisficing bug report?" in *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS),* 2016, pp. 164-174.

[2] B. Boehm and V. R. Basili, "Top 10 list [software development]," *Computer,* vol. 34, pp. 135-137, 2001.

[3] H. Hu *et al*, "Effective bug triage based on historical bug-fix information," in *2014 IEEE 25th International Symposium on Software Reliability Engineering,* 2014, pp. 122-132.

[4] M. Christ *et al*, "Modern Triage in the Emergency Department," *Dtsch. Arztebl Int.,* vol. 107, pp. 892-898, Dec, 2010.

[5] J. Zhou, H. Zhang and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *2012 34th International Conference on Software Engineering (ICSE),* 2012, pp. 14-24.

[6] Monitor ERP System AB. (). *MONITOR Affärssystem - Monitor ERP System AB*. Available: https://www.monitor.se/.[Accessed: 12 april 2017]

[7] M. Castro, M. Costa and J. Martin, "Better Bug Reporting with Better Privacy," *SIGPLAN Not.,* vol. 43, pp. 319-328, mar, 2008.

[8] S. Breu *et al*, "Information needs in bug reports: Improving cooperation between developers and users," in *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work,* Savannah, Georgia, USA, 2010, pp. 301-310.

[9] N. Bettenburg *et al*, "What makes a good bug report?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering,* Atlanta, Georgia, 2008, pp. 308-318.

[10] S. Davies and M. Roper, "What's in a bug report?" in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement,* Torino, Italy, 2014, pp. 26:1-26:10.

[11] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering,* Atlanta, Georgia, USA, 2007, pp. 34-43.

[12] F. Thung, P. S. Kochhar and D. Lo, "DupFinder: Integrated tool support for duplicate bug report detection," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering,* Vasteras, Sweden, 2014, pp. 871-874.

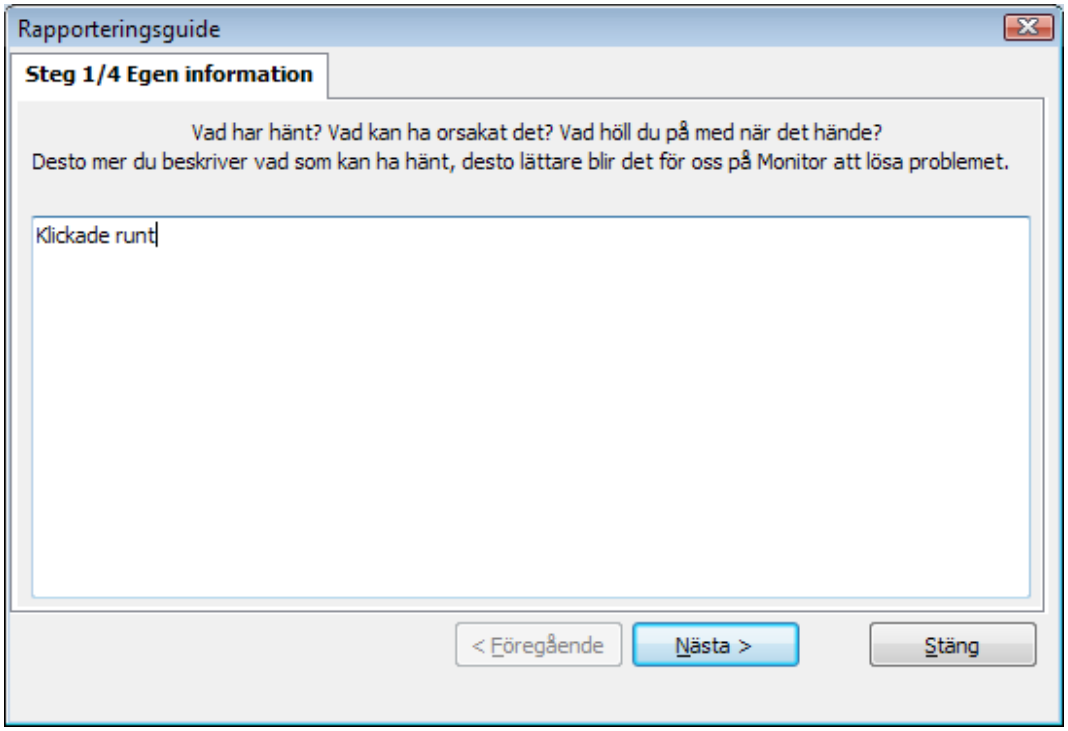
[13] Apache Friends. (). *XAMPP*. Available: https://www.apachefriends.org/index.html.[Accessed: 28 april 2017]

[14] JetBrains s.r.o. (). *PhpStorm*. Available: https://www.jetbrains.com/phpstorm/.[Accessed: 1 maj 2017]

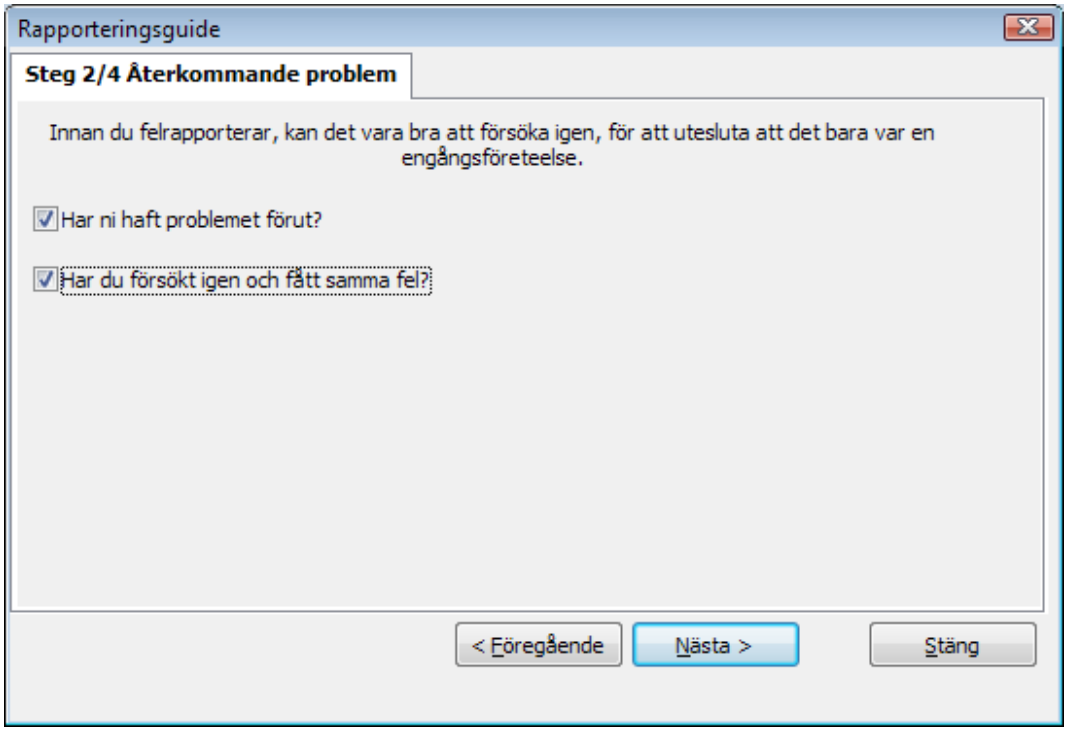
[15] Google Inc. (). *Google Forms*. Available: https://docs.google.com/forms. [Accessed: 17 april 2017]

Bilaga A

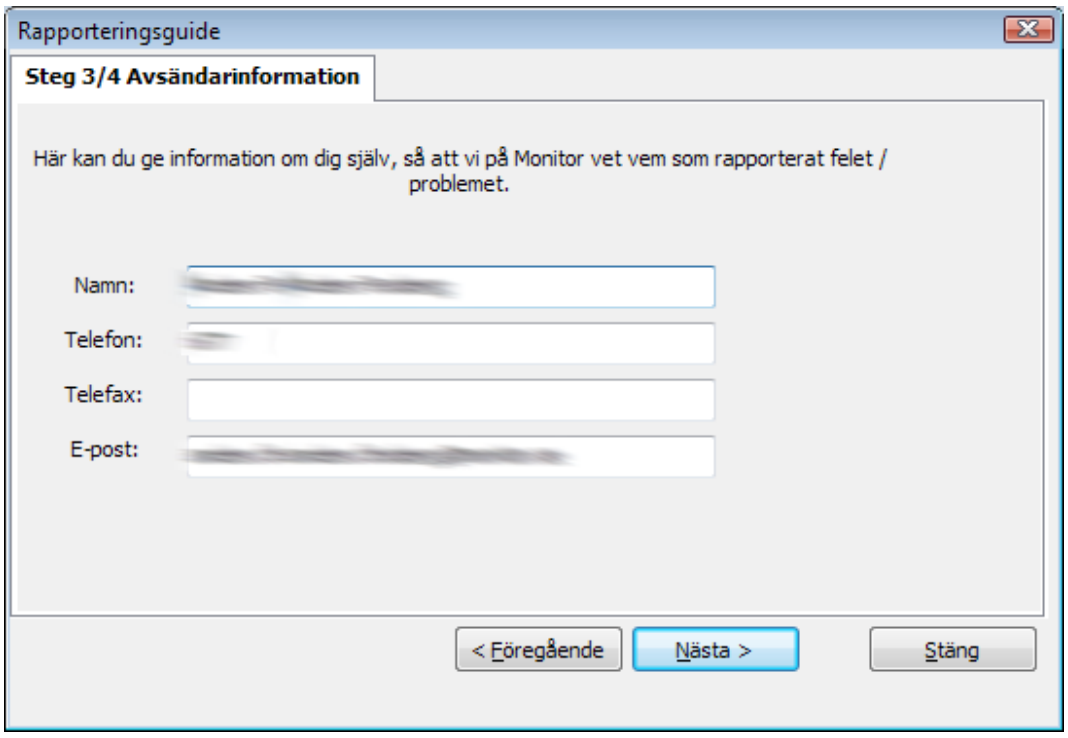
Rapporteringsguiden för att öppna ett supportärende steg för steg i Monitor G4.



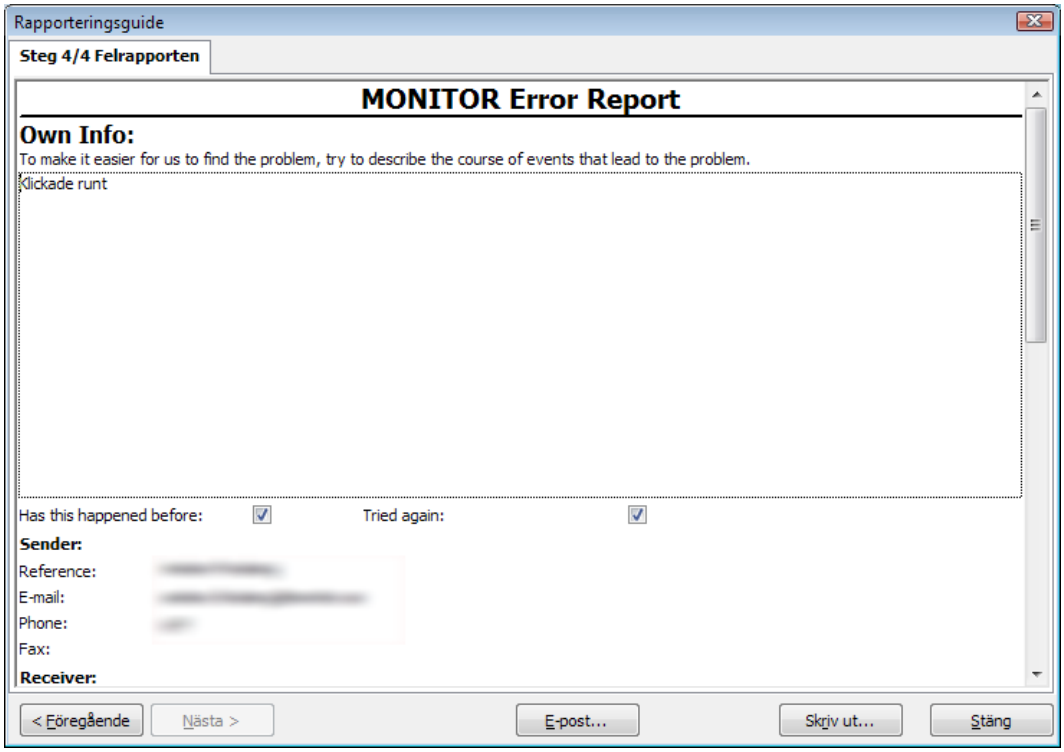
Figur 7- Användaren får med egna ord redogöra för stegen som ledde upp till kraschen.



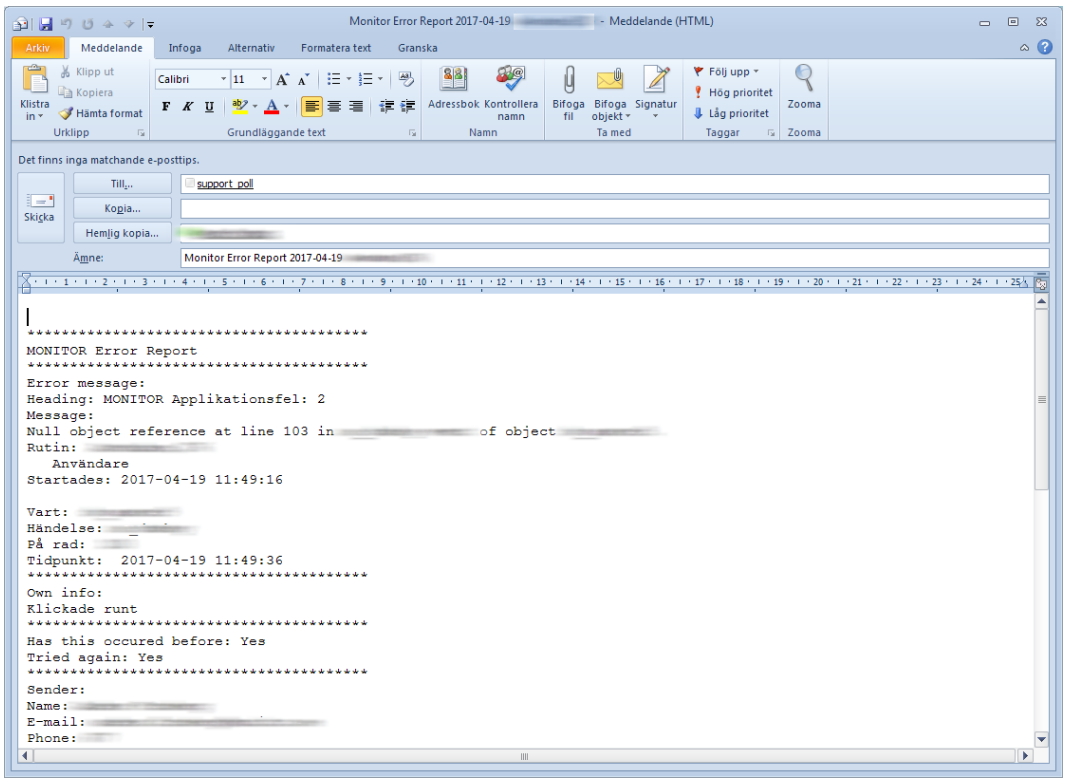
Figur 8 - Har problemet visats förut, och har användaren själv försökt återskapa det?



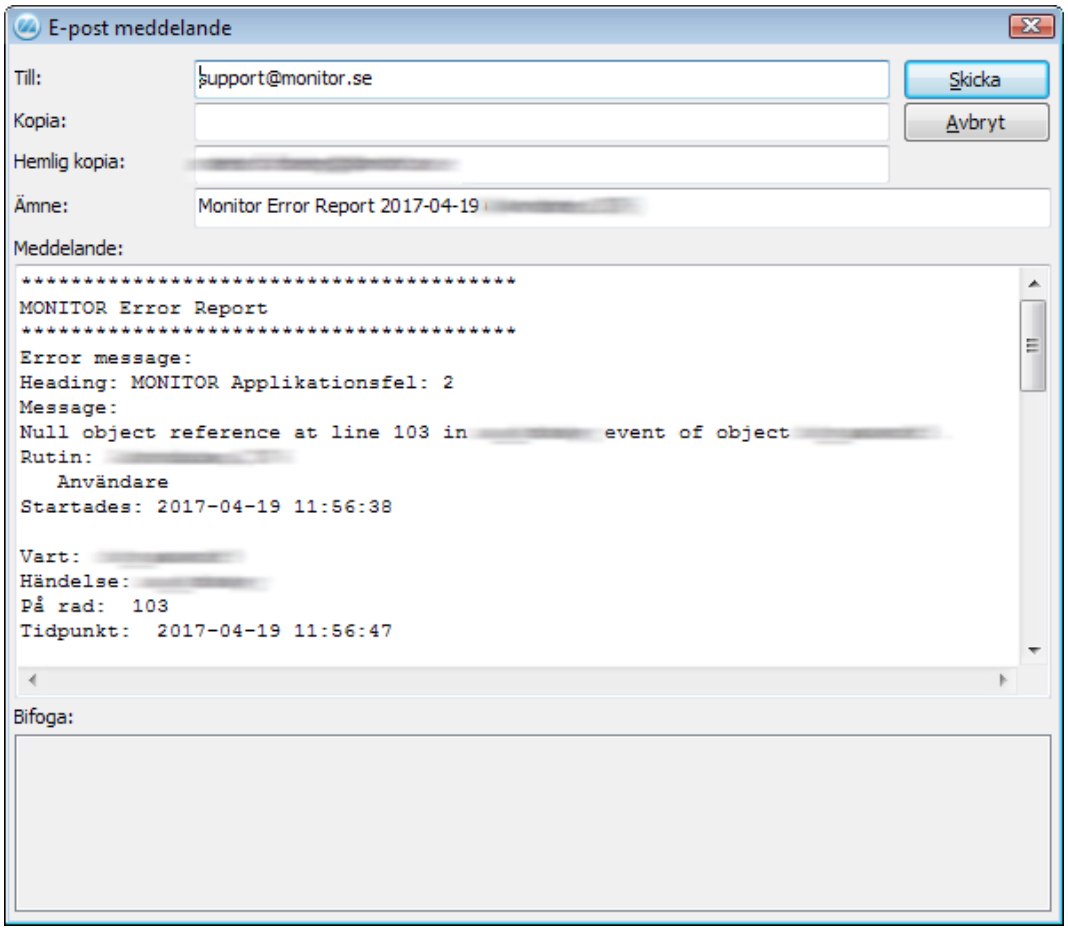
Figur 9 - Kontaktinformation ifall Monitor behöver få kontakt med rapporteraren.



Figur 10 - Sammanfattning av rapporten som sedan skickas via e-post till Monitor.



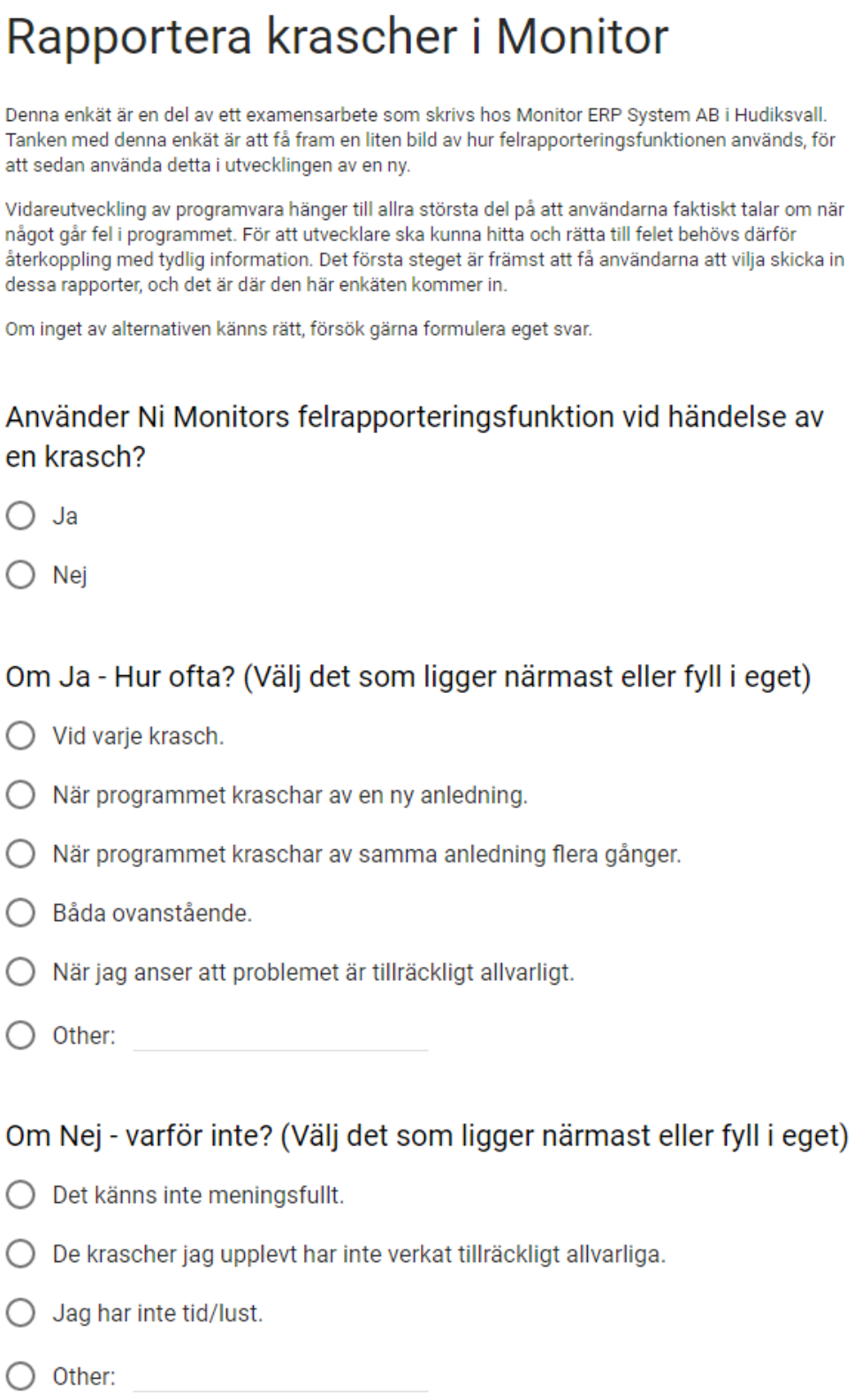
Figur 11 - Rapporten skickas via e-post-klient.



Figur 12 - Rapport skickas via förinställda SMTP-inställningar.

Bilaga B

Enkätundersökningen som skickades till G4-kunder



Rapportera krascher i Monitor

Denna enkät är en del av ett examensarbete som skrivs hos Monitor ERP System AB i Hudiksvall. Tanken med denna enkät är att få fram en liten bild av hur felrapporteringsfunktionen används, för att sedan använda detta i utvecklingen av en ny.

Vidareutveckling av programvara hänger till allra största del på att användarna faktiskt talar om när något går fel i programmet. För att utvecklare ska kunna hitta och rätta till felet behövs därför återkoppling med tydlig information. Det första steget är främst att få användarna att vilja skicka in dessa rapporter, och det är där den här enkäten kommer in.

Om inget av alternativen känns rätt, försök gärna formulera eget svar.

Använder Ni Monitors felrapporteringsfunktion vid händelse av en krasch?

- ○ Ja
- ○ Nej

Om Ja - Hur ofta? (Välj det som ligger närmast eller fyll i eget)

- ○ Vid varje krasch.
- ○ När programmet kraschar av en ny anledning.
- ○ När programmet kraschar av samma anledning flera gånger.
- ○ Båda ovanstående.
- ○ När jag anser att problemet är tillräckligt allvarligt.
- ○ Other: _____

Om Nej - varför inte? (Välj det som ligger närmast eller fyll i eget)

- ○ Det känns inte meningsfullt.
- ○ De krascher jag upplevt har inte verkat tillräckligt allvarliga.
- ○ Jag har inte tid/lust.
- ○ Other: _____

Figur 13 - Första delen av enkäten

Om Nej - Vad skulle få Er att göra det? (Välj det som ligger närmast eller fyll i eget)

- ◯ Om processen automatiserades så jag slipper slösa tid på det.
- ◯ Om jag fick direkt återkoppling när felet jag rapporterar rättats till, så känns det mer meningsfullt.
- ◯ Other: _____

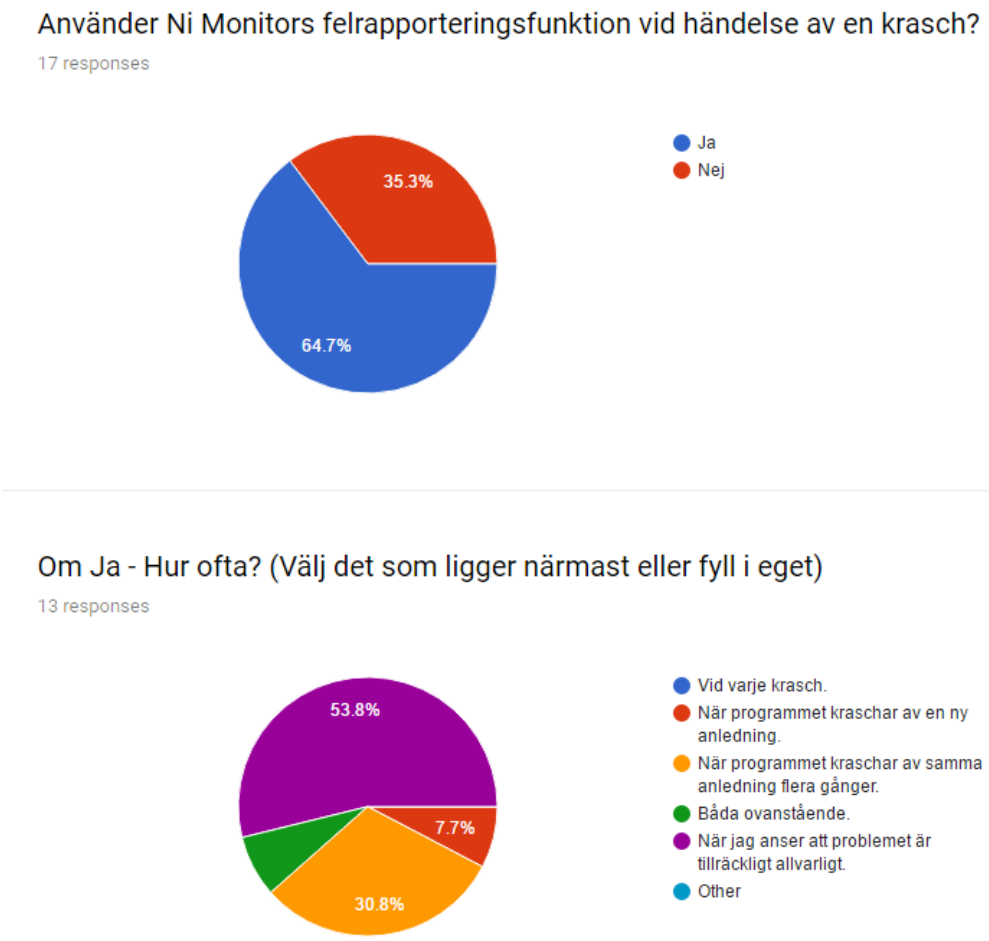
Om Ja - Hur utförlig brukar Ni vara? (Välj det som ligger närmast eller fyll i eget)

- ◯ Så utförlig och tydlig jag kan (Förklarande text steg för steg, skickar in skärmdumpar, etc)
- ◯ Jag kör snabbversionen (Klickar bara i relevanta checkboxar och skickar)
- ◯ Other: _____

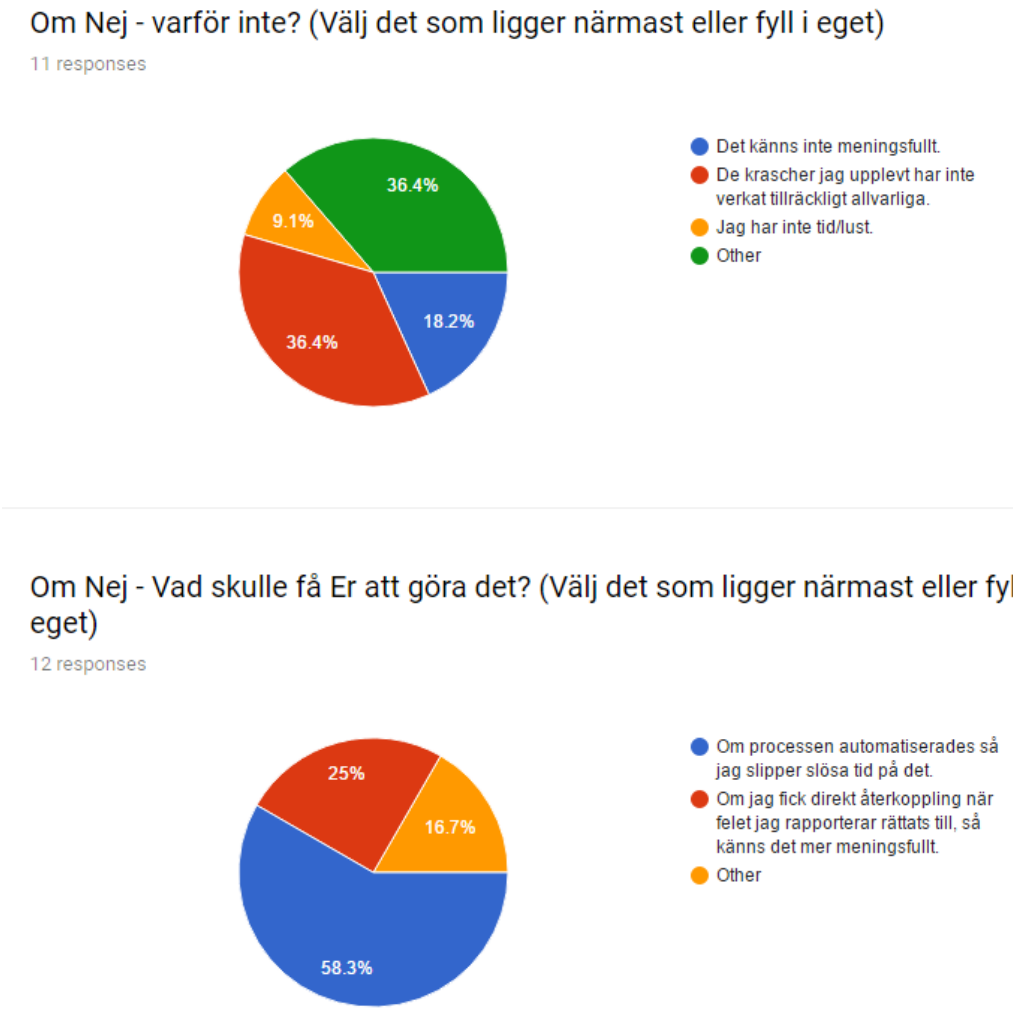
Figur 14 - Andra delen av enkäten

Bilaga C

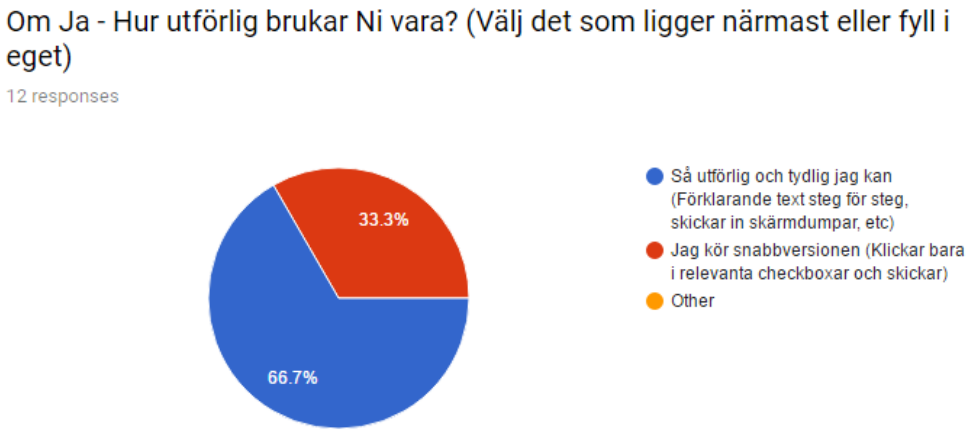
I denna bilaga presenteras det sammanställda resultatet på enkätundersökningen.



Figur 15 - Första delen av enkätresultatet



Figur 16 - Andra delen av enkätresultatet



Figur 17 - Sista delen av enkätresultatet