EXAMENSARBETE INOM TEKNIKOMRÅDET
INFORMATIONSTEKNIK
OCH HUVUDOMRÅDET
DATALOGI OCH DATATEKNIK,
AVANCERAD NIVÅ, 30 HP
STOCKHOLM, SVERIGE 2017

Kodmått som indikation på brister i underhållbarhet

JONAS MODLING

KTH
SKOLAN FÖR DATAVETENSKAP OCH KOMMUNIKATION

Kodmått som indikation på brister i underhållbarhet

JONAS MODLING

Masterprogram, datalogi (TCSCM)
Datum: 12 juni 2017
Handledare: Stefan Nilsson
Handledare: Magnus Larsson
Examinator: Henrik Artman
Engelsk titel: Code Metrics As Indication Of Flaws In Maintainability
Skolan för datavetenskap och kommunikation

Sammanfattning

Det här examensarbetet undersöker sambandet mellan kodmåtten cyklomatisk komplexitet, antalet metoder i en klass och djup i arvshierarkin och de faktiska anledningarna till att en kodbas blev grundligt refaktorerad. Man har länge försökt använda kodmått som indikationer på hur mycket ansträngning som krävs för att förändra en kodbas när nya krav framkommer. I fallstudien tittar vi på en kodbas som refaktorerats på grund av den dåliga underhållbarheten för att utreda till vilken grad kodmåtten antyder de problem som låg till grund för refaktoreringen. Målet är att ta reda på om man kan använda kodmått som indikator på vad som bör refaktoreras när man som konsult kliver in i ett projekt med kvalitetsproblem.

Metoden som används för att identifiera problemen i kodbasen är kvalitativa intervjuer, och mätvärdena hämtas ur verktyget NDepend. För att öka reproducerbarheten och validiteten i analysen kretsar intervjuerna kring definitionen av underhållbarhet ur ISO/IEC 25010 [1].

Slutsatsen är att de mått som ger väldigt höga värden på vissa klasser eller metoder ofta indikerar objekt som är involverade i större problem i kodbasen. I den här fallstudien indikerade dock måtten sällan objekt av ”rätt anledning”, alltså av den anledning som anges i litteraturen. Därför bör man inte använda kodmåttens värde som direkt indikation på objekt som ska refaktoreras, utan snarare välja att titta närmare på de objekt som indikerats och göra en professionell bedömning. Resultaten går inte helt i linje med tidigare forskning, där vanliga samband mellan kodmått och underhållbarhetsproblem sammanställts. Detta antas bero på överanvändning av lös koppling i koden som studerats.

Abstract

In this thesis, the correlation of the code metrics Cyclomatic Complexity, Number of Methods in Class and Depth of Inheritance Tree with the real reasons a codebase got heavily refactored is investigated. Code metrics have long been used in a attempts to indicate the amount of effort needed to change a codebase when new requirements emerge. In the case study we look at a codebase that has been refactored due to the low maintainability, to see to which degree the code metrics imply the problems on which the decision to refactor was based. The goal is to find out whether code metrics can be used as indicators of what should be refactored when you as a consultant join a project with existing quality problems.

The method used to identify the major problems in the codebase is qualitative interviews and the code metrics values are measured with the tool NDepend. To improve the reproducability and validity of the analysis, the interviews are built around the definition of maintainability presented in ISO/IEC 25010 [1].

The conclusion is that metrics showing very high values for certain classes or methods often indicate objects involved in major problems in the codebase. In this case study though, the measures seldom indicated objects for the ”right reason”, by which we mean the reason given in the literature. Because of this, the code metric values should not be used as a direct indication of what should be refactored but rather as a direction in which to look while doing a professional assessment. The results are not entirely in line with previous research, where common correlations between code metrics values and maintainability problems have been compiled. This is assumed to be due to the high degree of loose coupling in the studied code.

Innehåll

Kapitel 1

Introduktion

Målet med det här examensarbetet är att utreda sambandet mellan tre utvalda kodmått och orsaker till en grundlig refaktorering av en kodbas.

Syftet är att kunna använda kodmått i säljprocessen hos konsultbolaget Attentec AB, som ett argument för att förbättra existerande kod, refaktorera den, innan man vidareutvecklar den.

1.1 Bakgrund

Kodmått har länge använts i mjukvaruutvecklingens periferi, men har potential att vara riktigt användbar. Forskning har fokuserat på att använda kodmått till att uppskatta hur stor investering som krävs för att underhålla kod. Här kommer kodmåtten användas i en ny kontext.

Attentec AB är ett konsultbolag med fokus på modern mjukvaruutveckling, vars säljande punkt är att de kan utveckla mer skräddarsydd mjukvara än andra. Fokus ligger på kunden och det är viktigt att *rätt* kvalitet levereras, av kostnadsskäl. Kunder ska inte behöva betala för mer än de behöver, och kod ska inte innehålla mer funktionalitet än som är nödvändigt.

I flera fall har Attentec vidareutvecklat befintlig mjukvara, ibland för att utöka funktionaliteten i mjukvaran och ibland för att höja kvaliteten. Attentecs kunder är i allmänhet inte intresserade av refaktoreringsprojekt, eftersom det är svårt för en ägare av ett system att se värde i utveckling när ingen ny funktionalitet implementeras. Det medför svårigheter i säljprocessen när kunden måste övertalas att betala för refaktorering av sitt projekt innan nyutveckling kan ske. Ett fall av den här typen behandlas i det här examensarbetet.

Analysen av en kodbas innan man accepterar ett uppdrag har hittills gjorts genom att en utvecklare manuellt går igenom koden och med hjälp av erfarenhet berättar om vilka delar av koden som ser problematiska ut. Kodmåtten är tänkta som ett verktyg som objektivt stödjer de här misstankarna, vilket anses kunna vara en stark säljande punkt för refaktorering.

1.2 Målspecifikation

Målet är att utreda sambandet mellan de klasser och funktioner med högst mätvärden från de i litteraturen kända måtten cyklomatisk komplexitet, antal metoder i en klass och djup i arvshierarkin och de verkliga orsakerna till att en kodbas refaktorerades.

Den här rapporten är en typ av validering av tidigare forskning, samt ett alternativt sätt att angripa problemet att mäta underhållbarhet: kvalitativa intervjuer. Tidigare forskning har fokuserat på samband mellan hur mycket en kodbas förändras och kodmåttens värden, men här befinner vi oss i den unika situationen att man vet att kodbasen som studeras har refaktorerats och kan därför göra jämförelsen mellan kodmåttens värden och de faktiska orsakerna till refaktoreringen. Resultatet av den här undersökningen kan användas som datapunkt i bedömningen om man vill applicera de tre kodmått som här utvärderats i sitt eget projekt, samt till att skapa en grundläggande bild av hur väl kodmått kan användas som direkt input till bedömningar av underhållbarhet.

1.3 Fallstudiens kodbas

Fallstudien undersöker ett program för hantering av expeditionsdata som när Attentec tog på sig uppdraget att vidareutveckla det redan hade utvecklats i 8-9 år. I systemet kan man spela in tidsstämplad data från olika sensorer (ljud, video, position, orientering, osv.) och spara det. Informationen kan sedan i programmets grafiska gränssnitt visualiseras och spelas upp i olika hastigheter (eller baklänges) och man kan sätta ut punkter på tidsaxeln för att markera intressanta händelser.

Programmet är implementerat enligt arkitekturen Microsoft Composite UI Application Block [2] och är uppdelat i ett 30-tal fullständigt separata moduler. Koden är ungefär 13000 rader kod skrivet i C# och uppdelat i cirka 600 klasser. På grund av de problem Attentecs konsulter hade att vidareutveckla och förbättra programmet genomfördes en omfattande refaktorering av koden. Den här studien baserar sig på koden som den såg ut precis innan refaktoreringen påbörjades.

1.4 De utvalda kodmåtten

Kodmåtten som använts i den här studien är cyklomatisk komplexitet, antal metoder i en klass och djup i arvshierarkin. Nedan följer en kort introduktion till de tre.

Det cyklomatiska komplexitetstalet [3] mäter antal möjliga vägar exekveringen av programmet kan ta. Forskningen publicerades 1976 och måttet beräknas genom att man räknar antalet alternativa vägar genom kontrollflödesgrafen, vilket i princip ger antalet if-, for- och while-satser i koden.

Antal metoder i en klass [4] är ett mått på ett objekts komplexitet, härlett ur Bunges definition av komplexitet: antalet komponenter något består av. Med det här tankesättet har en klass med många variabler och metoder en hög komplexitet. Måttet definierades av Chidamber and Kemerer [4] år 1994.

Djup i arvshierarkin [4] relateras till Bunges uppfattning av influensområde. Det avser alltså att mäta till vilken utsträckning en klass påverkas av andra klasser, i det här fallet sina föräldraklasser. Man mäter hur långt ifrån roten i arvshierarkin en klass befinner sig.

1.5 Översikt

Kapitel 1 introducerar läsaren till studiens mål, bakgrund och den kodbas som kommer undersökas.

Kapitel 2 presenterar relevant teori kring kodmått, underhållbarhet och kvalitativa intervjuer.

Kapitel 3 presenterar metoden som använts under arbetets genomförande.

Kapitel 4 presenterar mätvärden från undersökningen av kodbasen, de intervjufrågor som tagits fram samt de teman som funnits i intervjun.

Kapitel 5 presenterar tolkningar av de uppmätta värdena från kapitel 4.

Kapitel 6 innehåller diskussioner kring vilka problem som fanns i kodbasen och hur de pekades ut av kodmåtten. Här diskuteras även metodvalet och alternativa metoder.

Kapitel 7 sammanfattar diskussionerna och presenterar slutsatser och möjlig framtida forskning. Här beskrivs också i vilken grad målet med examensarbetet uppfyllts.

I appendix bifogas definitioner som användes vid intervjuerna, de kategoriseringar som gjorts, samt citaten från intervjuerna som kategoriseringarna baserade sig på.

Kapitel 2

Teori

Det här kapitlet innehåller en introduktion till underhållbarhet hos kod, en översiktlig presentation om forskning inom kodmått, samt ett stycke om utformning och analys av kvalitativa intervjuer.

2.1 Underhållbarhet

Underhållbarhet kallas ett kodstyckes egenskap att vara lätt att underhålla. Den standarddefinition som ofta används i litteraturen är IEEE's definition i dokumentet *IEEE Standard Glossary of Software Engineering Terminology*, som introducerades 1983. Dokumentet har sedan uppdaterats två gånger, år 1990 och 2010. Definitionen från 2010 lyder:

> "*Maintainability: The ease with which a software system or component can be modified to change or add capabilities, correct faults or defects, improve performance or other attributes, or adapt to a changed environment.*" [5]

Vilket på svenska kan översättas till:

> "*Underhållbarhet: Enkelheten med vilken ett mjukvarusystem eller komponent kan modifieras för att ändra eller lägga till funktionalitet, rätta till fel eller defekter, öka prestanda eller andra attribut, eller anpassas till en föränderlig omgivning.*"

Underhållbarhet har delats upp i flera olika faktorer som influerar hur underhållbart ett mjukvarusystem är. ISO/IEC 25010 [1] är en standard som definierar programvarukvalitet. Kvalitén underhållbarhet är i ISO/IEC 25010 uppdelad i de här fem komponenterna:

1. moduläritet
2. återanvändbarhet
3. analyserbarhet
4. modifierbarhet
5. testbarhet

De huvudsakliga verktyg som forskningen har ställt till förfogande för utvecklare är riktlinjer för hur kod bör se ut.

2.2 Forskning på kodmått: 1976 - Idag

Ingen del av ISO/IEC 25010's beskrivning av underhållbarhet [1] är något man direkt kan mäta. Forskningen på området började med att mäta faktorer som upplevdes bidra till underhållbarhet, eller använde komplexitetsmått som prediktorer. Resultaten från forskningen testades utförligt och fanns i vissa fall tillförlitliga i att förutsäga underhållbarhet, men led av bristfällig vetenskaplighet. Senare försök genomfördes i samband med intåget av objektorienterad programmering och höll högre vetenskaplig kvalitet. Nedan följer en översikt över forskning inom kodmått och underhållbarhet i kronologisk ordning.

2.2.1 1970-talet: De första måtten

De första populära måtten på underhållbarhet, det cyklomatiska komplexitetstalet [3] och Halstead's alla mått [6], är egentligen mått på komplexitet. De togs fram i mitten på 1970-talet av McCabe [3] och Halstead [6] i samband med publiceringen av två rapporter. Båda verken har fått kritik för dålig vetenskaplig grund [7, 8, 9, 4], men de respektive måtten har testats utförligt och visat sig vara starka indikationer på underhållbarheten hos framförallt funktionsorienterad kod.

Det cyklomatiska komplexitetstalet [3] mäter antal möjliga vägar exekveringen av programmet kan ta. Måttet beräknas genom att man räknar antalet alternativa vägar genom kontrollflödesgrafen, vilket i princip ger antalet if-, for- och while-satser i koden.

Halstead [6] definierade 10 mått varav främst två, Halstead's Effort (E) och Halstead's Volume (V), har används frekvent i områden som utbildning, open-source-programvara och kommersiella verktyg för mjukvaruutveckling [9]. Verket är hårt kritiserat för att inte tydligt definiera vilken enhet man mäter i när man beräknar exempelvis Effort (E), som påstås mäta tidsåtgång för utveckling av kod [9].

2.2.2 1994: En svit av objektorienterade designmått

Chidamber and Kemerer [4] försökte år 1994 adressera behovet av mått på objektorienterad kod och introducerade de första empiriskt validerade formella kodmåtten på objektorienterad kod. De tog fram sex design-relaterade mått med teoretisk grund i Bunges ontologi [10]. Måtten har sedan frekvent refererats av andra forskare [7].

Exempelvis uttrycker Chidamber and Kemerer [4] koppling och sammanhållning som graden av extern påverkan hos delar av ett system och hur internt konsekvent delarna av systemet är. Mer precist definieras koppling som att två klasser är kopplade när en av klasserna påverkar en annan. Detta kan ske genom metodanrop eller genom arv när en barnklass använder en föräldraklass-metod. Sammanhållning mellan två metoder definieras (inspirerat av hur Bunge definierar likhet) som snittet av mängden instansvariabler som används av metoderna.

Måttet djup i arvshierarkin relateras till Bunges uppfattning av influensområde. Det avser alltså att mäta till vilken utsträckning en klass påverkas av andra klasser, i det här fallet sina föräldraklasser.

Ett annat mått som definieras är antal metoder i en klass, som ett mått på ett objekts komplexitet härlett ur Bunges definition av komplexitet: antalet komponenter något består av. Med det här tankesättet har en klass med många variabler och metoder en hög komplexitet.

Vidare diskuteras mått på antal barnklasser, sammanhållningen mellan funktioner i en klass, kopplingen mellan objekt i termer av metodanrop på olika sätt, hur stor del av koden som kan komma att exekveras som svar på ett metodanrop till klassen och ett mått på hur kod blir mer komplex genom interaktion mellan objekt.

Måtten i verket utvärderas med hjälp av sex (av totalt nio) analytiska egenskaper framtagna av Weyuker genom insamling av empirisk data. Kriterierna återges i verket [4].

2.2.3 1993: Underhållbarhetsmått i objektorienterad programmering

Li and Henry [11] genomförde 1993 statistiska undersökningar på mått från ett utkast av rapporten från Chidamber and Kemerer [4]. De försökte fastställa om måtten var tillförlitliga indikationer på ansträngningen som krävdes för att underhålla system genom att studera två system under tre års tid. Li and Henry [11] definierade förändring i en kod som antal rader som ändrats och noterar att förändringen i en kod och ansträngningen som krävs för att göra förändringen inte är samma sak, men påpekar att de två värdena korrelerar.

Resultatet visade starka indikationer på att de undersökta måtten var tillräckliga för att förutsäga mängden förändring i en kod över tid. Man observerade en tydlig förbättring i förutsägelserna när man använde de nya objektorienterade måtten, jämfört med att använda de då konventionella måtten.

2.2.4 2006: Historisk problematik med kodmått

Lincke and Löwe [7] ger en utförlig beskrivning av arbeten som gjorts på området fram till 2006. Gemensamt är att många bygger vidare på Chidamber and Kemerer [4] och att många saknar en metamodell för kodmått, vilket gör analys och enhetlighet svårt. Lincke and Löwe [7] upprättar en formell meta-modell för kodmått och formaliserar tidigare arbeten, så att definitionerna är entydiga. Detta för att ha en konsekvent bas för undersökningar och experiment i framtiden.

Lincke et al. [12] bekräftade även i en undersökning 2008 att mätningar och slutsatser av mätningar av tidigare mått skiljer sig markant beroende på verktyg. De fastställer flertydigheten i tidigare definitioner och avsaknaden av en *gold standard*, en standard att jämföra mätvärden med.

Ett exempel på avvikelse identifierad i [12] är vid mätning av CBO-värdet (*Coupling Between Objects* ur [4]) hos Java-kod. Vissa verktyg har valt att utelämna anrop till klassen `Object`, andra har valt att utelämna anrop till paketet `java.lang.*`, ytterligare andra mäter alla anrop. Alla tre kan tyckas vara rimliga val, de två tidigare riktar förmodligen in sig på att undvika att mäta standardbiblioteket för att användaren ska ha mer nytta av mätningen.

2.2.5 2012: Gränsvärden för moderna kodmått

År 2012 identifierade Ferreira et al. [13] behovet av att identifiera normalvärden för kodmått. Sex mått, huvudsakligen baserade på Chidamber and Kemerer [4] valdes ut och applicerades på över 26 000 klasser från öppen-källkods-projekt. Härifrån härleds vad vanliga värden för de respektive måtten är. I rapporten undersöks även projektet JHotDraw, ett projekt med erkänt ”bra kod”. JHotDraw har av flera andra forskningsgrupper

klassificerats som högkvalitativ [13]. Slutsatsen är att måtten definitivt har kapacitet att indikera fall av bristfällig design, men att falsklarm förekommer.

2.3 Kodmått i praktiken: ett exempel

Building Maintainable Software [14] är en bok skriven av Joost Visser vid Radboud University i Nijmegen. Hans forskning tillämpas av företaget Software Improvement Group (SIG), som ägnat sig åt konsultarbete inom programvarukvalitet sedan 2000-talets början. Boken sammanfattar några av de riktlinjer för underhållbar programvara som SIG har identifierat under åren.

Boken behandlar 10 principer för underhållbar kod och tillhandahåller gränsvärden som förenklar klassifikation av programvara efter en underhållbarhetsskala. Exempel på principer är *skriv korta kodenheter, duplicera inte logik, håll interfaces små, begränsa mängden moduler, balansera modulers storlek*. Rekommenderade riktvärden för exemplen är att kodenheter, metoder om man pratar Java, ska hålla sig i under 15 rader kod. Kod ska refaktoreras och delas upp för att minska duplicering. Metoder ska inte ha fler än 4 parametrar. Ditt program ska vara uppdelat i 6-12 moduler och modulerna ska vara jämförbara i storlek.

SIGs verktyg mäter hur väl ett mjukvarusystem följer SIGs riktlinjer och klassas på en 5-gradig skala (1 är dåligt, 5 är bra) efter hur underhållbara SIGs verktyg anser att de är. Ett exempel på vad som mäts är längd på metoder. För att klassas som 4/5 inom den här mätningen får max 7% av alla metoder vara längre än 60 rader kod, och längderna på metoderna i applikationen måste hålla sig under en viss fördelningskurva. Den här fördelningen av längder på metoder har SIG definierat och förbättrat under åren och finns för alla andra mätningar också.

2.4 Verktyg för kodmått för C# i dagsläget

Många verktyg har utvecklats framförallt för C++ och Java, tillsammans med ett fåtal för andra språk (däribland C#). VizzAnalyzer [15] är ett verktyg som utvecklats som del i ett forskningsprojekt och som tidigare verkar ha varit fritt att använda, men som i dagsläget är integrerat i företaget ARiSAs verktyg *Quality Monitor* och erbjuds som del i deras kommersiella tjänster.

NDepend [16] är ett verktyg inriktat på att profilera .NET-kod, vilket innebär att det även kan mäta C#-kod. Företaget bakom programmet har funnits sen 2004 och har rekommenderats av utvecklare och mjukvaruarkitekter på bl.a. Microsoft och HP.

Axivion Bauhaus Suite [17] erbjuds av företaget Axivion, som uppstod som följd av ett tyskt forskningssamarbete mellan två universitet under senare delen av 90-talet. Sviten är en helhetslösning för programvaruutvärdering, men tillhandahåller kodmått som en del i erbjudandet. C# ingår bland de stödda språken.

En mängd plugins till flera integrerade utvecklingsmiljöer är också tillgängliga, bland annat mäts ett fåtal mått av Microsoft Visual Studio 2015 for C# i standardutförande. De flesta av dessa plugins verkar utvecklas av individuella utvecklare på fritiden.

Kapitel 3

Metod

Målet med metoden är att utreda i vilken utsträckning den cyklomatiska komplexiteten, antalet metoder i en klass och var en klass befinner sig i arvshierarkin (*Cyclomatic Complexity (CC)*, *Number of Methods in Class (NMC)* och *Depth of Inheritance Tree (DIT)*) visar på samma problemområden som Attentecs konsulter har pekat ut i ett tidigare projekt.

En källa till information kring kvaliteten hos en kodbas är analysen som en erfaren utvecklare som arbetat med kodbasen kan göra. Den information som kan utvinnas skulle kunna vara att jobbiga delar av koden pekas ut, identifikation av bra eller dåliga konventioner som följs eller mer övergripande, allmänna strukturfel hos koden.

En annan informationskälla är de mätvärden man kan utvinna med hjälp av kodmått, samt analysen man kan göra av mätvärdena. Här kan man utvinna information om brott mot riktlinjer för god design, en uppskattning av komplexiteten hos koden och ett referensvärde för hur självständiga komponenterna i systemet är. Analysen görs utifrån måttens definitioner och tänkta följder. Exempelvis antas ett objekt med hög cyklomatisk komplexitet vara svår att testa på grund av att metoden förmodligen har för många syften [14]. Om utvecklarna pekat på problem att förstå koden eftersom den gör för många saker, eller pekat på kvalitetsproblem som har med testbarhet att göra kan alltså måttets värde anses sammanfalla med utvecklarens uttalande.

Metoden är utformad för att jämföra de här två informationskällorna. Ger de olika resultat? Mätningar genomförs på studiens kodbas och utvecklare intervjuas kring vilka problem de såg som ledde till refaktoreringen. Därefter analyseras resultatet för att hitta korrelation mellan mätvärden och problem.

3.1 Hur måtten valdes för undersökningen

De mått som använts i det här arbetet är cyklomatisk komplexitet, antal metoder i en klass och djup i arvshierarkin. De valdes ut genom en subjektiv bedömning av kodbasen, där det söktes efter teoretiskt välgrundade mått som hade stor sannolikhet att uppvisa tydligt avvikande värden för någon grupp klasser. Detta eftersom mått utan avvikande värden ansågs mindre lämpade att indikera vilka klasser som bidrog mest till problem i koden. Efter bedömningen var ett tiotal mått aktuella, varvid NDepends kompletta analys kördes. Ndepend använder sig av runt 200 mått för att göra sina bedömningar, tillsammans med andra faktorer. Flera av de mått som funnits lämpliga återfanns bland orsakerna till de problem som NDepend prioriterade som värst. Bland de som överensstämde valdes tre ut, för att man i intervjuerna skulle kunna undersöka måtten på djupet

med rimlig tidsåtgång.

3.2 Mätningar

Mätningarna i den här fallstudien baserar sig på verktyget NDepend [16]. NDepends mål och syfte är att ge en helheltsbild av bl.a. underhållbarheten hos en kodbas, men det är enkelt att uttyda vilka mätvärden som använts i bedömningen genom rapporten som genereras av verktyget. Det är svårare att uttyda exakt vad som mäts, ett fenomen som inte är helt okänt i litteraturen [12]. En utvärdering av mätvärdena tyder på att måttet NMC mäts exklusive konstruktorer och inklusive metoder från föräldraklasser. Måttet DIT verkar i NDepends rapport mätas inklusive all ramverkskod, så om en klass beror på en del i ett ramverk som har DIT = 8, så kommer den nya klassen ha DIT = 9.

3.3 Intervjuer

Två utvecklare som var delaktiga i refaktoreringen är fortfarande tillgängliga. Utvecklarna intervjuas individuellt under sammanlagt fem timmar för att bättre förstå de problem de hade med koden.

Intervjuerna utformas enligt en metod inspirerad av Lars Torsten Eriksson [18]: Först ska syfte och mål med undersökningen identifieras, därefter väljs informationsbehov, observationsenhet och sedan en metod för datainsamling. Intervjufrågor formuleras så att de är specifika och enkla att förstå och en förstudie ska sedan genomföras för att bekräfta att intervjufrågorna uppfyller ens informationsbehov.

Till en början definieras följande informationsbehov:

1. Vad är problematiskt i koden?
2. Vad är orsakerna till problemen?
3. Pekar måttens högsta mätvärden ut problem?
4. Vad är orsakerna till de problem måtten pekar ut?

Målet är sedan att genomföra intervjuerna och sammanställa de identifierade problemen så att det blir tydligt vad som indikerades av mått och vad som bedömts vara ett problem av utvecklarna, men som undgick måtten. Detta genom att formulera frågor kring var problem fanns i koden och vilka orsaker de här problemen hade. Frågorna formuleras med hänsyn till John Sawatsky's 10 dödssynder vid intervjuer, som återges i [18].

För att tillfredsställa informationsbehovet behövs två kategorier av frågor: En kategori med allmänna frågor för att förstå så många problem i koden som möjligt och en kategori med frågor för att förstå vilka problem som pekades ut av kodmåtten. Intervjun delas därför in i en allmän del, där frågor om koden i allmänhet täcks, och en mer specifik del, där frågor som anknyter till mått ställs på specifika klasser eller funktioner som gett särskilt stort utslag vid mätning.

För att kunna jämföra resultatet av den allmänna och specifika delen av intervjun hämtas inspiration från ISO/IEC 25010's uppdelning av underhållbarhet: moduläritet, återanvändbarhet, analyserbarhet, modifierbarhet och testbarhet, som introducerades i

stycket Underhållbarhet i Teori-kapitlet. Syftet med de allmänna frågorna blir således inte bara att identifiera problem med kodbasen, utan även att identifiera i vilka klasser, funktioner och konstruktioner någon av de 5 delarna av underhållbarhet inte råder. Syftet med de specifika frågorna blir att identifiera förekomsten och orsaken till de av måtten utpekade problemen, samt reda ut om det var någon av de 5 delarna av underhållbarhet som orsakade det avvikande mätvärdet. För att öka sannolikheten att svarsgivaren uppfattar frågorna korrekt presenteras ISO/IEC 25010's definitioner av de respektive delarna av underhållbarhet för de svarande. Dessa bifogas i appendix till rapporten.

För att öka sannolikheten att svaren som samlas in faktiskt är användbara genomförs en förstudie på en ledig utvecklare, som pratar om den kodbas han sist arbetade med. Resultaten genomgår sedan en komplett analys i syfte att identifiera dåliga frågeformuleringar och irrelevant information. Processen upprepas tills att intervjun resulterar svar som tillfredsställer informationsbehovet, varvid de riktiga intervjuerna kan påbörjas.

Den ena intervjun genomfördes i ett konferensrum med en laptop med koden framför oss. Den andra genomfördes via telefon. Följdfrågor av typen ”varför?”, ”jag förstår inte vad du menar med X?” ställs, men i övrigt uteslutande de frågor som presenteras nedan.

3.3.1 Intervjufrågor

De exakta definitioner av de olika delarna av underhållbarhet (se stycket Underhållbarhet i kapitlet Teori) som använts vid intervjuerna bifogas i appendix.

För att identifiera allmänna problem med koden ställs följande frågor. Ingen ytterligare information ges till den svarande, förutom definitionen av de olika delarna av underhållbarhet. Frågor som inte är relevanta ignoreras (med ej relevant menas att en svarande inte ska behöva peka ut probleminstanser i koden när den svarande redan berättat att det inte finns några problem av efterfrågad typ).

1. Vad var problematiskt med koden?
   (a) moduläritet
      i. Vilka problem för vidareutveckling ser du, som relaterar till modularitet? Beskriv dem!
      ii. Peka ut specifika klasser eller funktioner vars moduläritet bidrog starkt till problemen.
      iii. På vilket sätt blev problemen värre på grund av klassens/funktionens moduläritet?
   (b) återanvändbarhet
      i. Vilka problem för vidareutveckling ser du, som relaterar till återanvändbarhet? Beskriv dem!
      ii. Peka ut specifika klasser eller funktioner vars återanvändbarhet bidrog starkt till problemen.
      iii. På vilket sätt blev problemen värre på grund av klassens/funktionens återanvändbarhet?
   (c) analyserbarhet
      i. Vilka problem för vidareutveckling ser du, som relaterar till analyserbarhet? Beskriv dem!

   ii. Peka ut specifika klasser eller funktioner vars analyserbarhet bidrog starkt till problemen.
   iii. På vilket sätt blev problemen värre på grund av klassens/funktionens analyserbarhet?
(d) modifierbarhet
   i. Vilka problem för vidareutveckling ser du, som relaterar till modifierbarhet? Beskriv dem!
   ii. Peka ut specifika klasser eller funktioner vars modifierbarhet bidrog starkt till problemen.
   iii. På vilket sätt blev problemen värre på grund av klassens/funktionens modifierbarhet?
(e) testbarhet
   i. Vilka problem för vidareutveckling ser du, som relaterar till testbarhet? Beskriv dem!
   ii. Peka ut specifika klasser eller funktioner vars testbarhet bidrog starkt till problemen.
   iii. På vilket sätt blev problemen värre på grund av klassens/funktionens testbarhet?

Följande frågor ställs för att identifiera specifika problem som relaterar till kodmåtten. Alla frågor om cyklomatisk komplexitet ställs kring varje funktion som givit höga mätvärden för att identifiera om mätvärdet har ett samband med något av de generella problemen identifierade med de allmänna frågorna. Frågorna om antal metoder och djup i arvshierarkin ställs på samma sätt kring varje av respektive mått indikerad klass.

1. Cyklomatisk komplexitet
   (a) Vilka problem för vidareutveckling bidrog den här funktionens komplexitet till? Beskriv dem!
   (b) På vilket sätt blev problemen värre på grund av funktionens komplexitet?
   (c) Vad anser du om funktionens moduläritet, återanvändbarhet, analyserbarhet, modifierbarhet, testbarhet?
2. Antal metoder i en klass
   (a) Vilka problem för vidareutveckling bidrog den här klassen till genom att erbjuda så många funktioner? Beskriv dem!
   (b) På vilket sätt blev problemen värre på grund av att klassen erbjuder så många funktioner?
   (c) Vad anser du om klassens moduläritet, återanvändbarhet, analyserbarhet, modifierbarhet, testbarhet?
3. Djup i arvshierarkin
   (a) Vilka problem för vidareutveckling bidrog den här klassen till genom sin plats i arvshierarkin?
   (b) På vilket sätt blev problemen värre på grund av klassens plats i arvshierarkin?
   (c) Vad anser du om klassens moduläritet, återanvändbarhet, analyserbarhet, modifierbarhet, testbarhet?

3.4 Analysmetod

Analysmetoden är inspirerad av den grundprocess presenterad av Hedin [19]. Analysen påbörjas med att nyckelord extraheras separat ur de båda delarna av intervjuerna. Teman och eventuella underkategorier till teman identifieras genom gruppering av nyckelord.

Därefter genomförs en tillförlitlighetskontroll där två lediga konsulter får tilldela citaten varifrån nyckelorden är hämtade till de identifierade temana. Vid 80% överensstämmelse mellan de två tilldelningarna och mina egna bedöms temana som tillförlitliga, annars formuleras temana om och tillförlitlighetskontrollen upprepas.

Identifierade problem ur de båda delarna av intervjuerna presenteras sedan i en tabell. Även svar från den specifika delen presenteras i tabellformat. En diskussion behandlar sambandet mellan måttens utfall och utvecklarnas svar. Här används måttens definitioner och tänkta syften ur litteraturen. Även problem som identifierats i koden, men som inte kunde identifieras genom måtten diskuteras.

Sist utvärderas huruvida syftet och målet med arbetet kan anses uppfyllt och förslag på vidare forskning inom området.

Kapitel 4

Resultat

Här presenteras de högsta uppmätta värdena från kodbasen tillsammans med en kort beskrivning av klassen eller funktionen som orsakade mätvärdet, som bara är till för att sätta läsaren i kontext. Källa till beskrivningarna är en av de intervjuade utvecklarna.

Därefter presenteras de formulerade intervjufrågorna tillsammans med ett kort resonemang kring deras syfte och bidrag till frågeställningarna. Sist presenteras de teman och kategoriseringar som funnits genom analys av de genomförda intervjuerna.

4.1 Uppmätta värden

Här presenteras de klasser och funktioner i den undersökta kodbasen med högst mätvärden. I fall där många mätningar gett samma mätvärde (som t.ex. djup i arvshierarkin, där väldigt många klasser visade sig ha samma djup) har de klasser och funktioner som NDepend rankat som viktigast valts ut.

4.1.1 Cyklomatisk komplexitet

Nedan presenteras de funktioner i projektet där den uppmätta cyklomatiska komplexiteten (CC) var högst. Cyklomatisk komplexitet sammanfaller i litteraturen oftast med långa funktioner och funktioner med flera syften och uppgifter.

Klass	Metod	CC
PersistentLayerSettingsService	`Unserialize()`	27
PersistentVideoView	`StateChanged()`	27
PersistentVideoSnapshotGrabberService	`GrabSnapshot()`	26
VisibleRangeTrackLayer	`Render()`	23

PersistentLayerSettingsService är en klass för att spara inställningar för kartvyer. Klassen är del i en egenutvecklad kartmotor, som Attentec valde att ersätta med en komponent från en tredje part under refaktoreringen.

PersistentVideoView är ett SmartPart-objekt i Microsofts CAB-arkitektur för att visa en videoström på skärmen.

PersistentVideoSnapshotGrabberService är en hjälpklass som används i genereringen av en rapport om en expedition.

VisibleRangeTrackLayer är ett kartlager för att visa någon specifik form av avstånd. Klassen var del i den kartmotor som byttes ut under refaktoreringen.

4.1.2 Antal metoder i en klass

Här presenteras de klasser med flest antal tillgängliga funktioner (NMC) i projektet. Ett högt antal funktioner i en klass tyder på en hög koppling till framförallt föräldraklasser och barnklasser, samt tyder på att klassen har för många syften och uppgifter.

Klass	NMC
MapController	48
SystemWorkItem	43
Vector3	42
Algorithm	40

MapController är en klass som hanterar användarinteraktion med kartor i det grafiska gränssnittet till applikationen.

SystemWorkItem är basklassen för alla WorkItems i projektet. WorkItems är en del i Microsofts CAB-arkitektur [2], som tillämpades under utvecklingen.

Vector3 är en kapsling av en 3D-position med en hel del matematiska stödfunktioner.

Algorithm är en samling generiska stödalgoritmer, så som swap, hash, copy och fold.

4.1.3 Djup i arvshierarkin

Den här tabellen presenterar de klasser som befinner sig så långt ned i arvshierarkin att NDepend indikerade att det var ett problem. Projektet innehöll ytterligare klasser med ett DIT-värde på 4 (precis som de i tabellen) men NDepend pekade bara ut dessa som problem. Att en klass ligger långt ner i arvshierarkin antas hänga ihop med att klassen är svårförstådd, eftersom många av de uppgifter som utförs av klassen utförs i en föräldraklass, och att de erbjuder fler funktioner än andra klasser, vilket leder till de problem som beskrivs i stycket om antal metoder i en klass ovan.

Klass	DIT
Tuple	8
RovDataInputStream	4
SmarTrackInputStream	4

Tuple är en tupel med 8 variabler. Den ärver från en tupel med 7 variabler och lägger till ett värde. Tupeln med 7 variabler ärver i sin tur från en tupel med 6 variabler, osv.

RovDataInputStream är en klass för att kapsla in en dataström från en fjärrstyrd robot.

SmarTrackInputStream är en klass för att kapsla in en dataström från en SmarTrack-sensor.

4.2 Teman och kategoriseringar

Nedan följer de teman som behandlades under intervjuerna. De är separerade för att tydliggöra vilka teman som behandlades under den allmänna och specifika delen av intervjuerna. Problemen är numrerade, och följande delar av rapporten refererar tillbaka till temana med hjälp av de här numren.

#	Identifierat problem
1.	Problem att för mycket mönster och arkitektur användes
2.	Problem att förstå syftet med delar av koden
3.	Problem att förstå hur en komponent beror på andra komponenter
4.	Problem att se om koden användes alls

Följande problem i kodbasen identifierades på samma sätt genom tilldelning av citat till problem. Citaten kommer här från när klasserna och funktionerna med extrema mätvärden diskuterades mer ingående.

#	Identifierat problem
5.	Problem med att förstå vad koden gör
6.	Problem när metoder inte är uppdelade ordentligt
7.	Problem med att förstå tanken och syftet med koden
8.	Problem att se hur koden hänger ihop
9.	Problem med att avancerade eller onödiga konstruktioner används
10.	Problem med att saker inte ligger på rätt plats i koden

4.2.1 Bedömningar av de utpekade klasserna

Så här bedömdes de undersökta funktionernas och klassernas moduläritet, återanvändbarhet, analyserbarhet, modifierbarhet, testbarhet av utvecklarna. Beskrivningen här är en sammanfattning av vad som sades under respektive delfråga när klasserna och funktionerna behandlades under intervjuerna.

PersistentVideoView.StateChanged()		
moduläritet	-	Funktionen agerar bara lokalt, men bör delas upp
återanvändbarhet	-	Går ej att återanvända, men det är inget problem här
analyserbarhet	-	Lokal och analyserbar, men lång
modifierbarhet	-	Koden är modifierbar
testbarhet	-	Svårtestat, detta är vy-kod, vilket ofta är svårtestat

`PersistentVideoView.StateChanged()` beskrevs som en tydligt avgränsad del av koden, en funktion som ”faktiskt gör någonting istället för att bara skyffla runt information”.

PersistentVideoSnapshotGrabberService.GrabSnapshot()		
moduläritet	-	Gör något väldigt specifikt, det var inget problem
återanvändbarhet	-	Dålig återanvändbarhet, men det är ok
analyserbarhet	-	För lång för att vara överskådlig, men det är vanligt med DirectShow
modifierbarhet	-	Modifierbar, den fyller bara ett syfte och är lokal
testbarhet	-	Den känns testbar

`PersistentVideoSnapshotGrabberService.GrabSnapshot()` beskrevs som en extremt lång funktion med dålig läsbarhet på grund av den höga graden av felhantering inbäddad i koden.

VisibleRangeTrackLayer.Render()		
moduläritet	-	Inget behov av modularitet här
återanvändbarhet	-	Helt omöjlig att återanvändba
analyserbarhet	-	Det här går ju inte att förstå
modifierbarhet	-	Jag vågar inte ändra i den
testbarhet	-	Kanske går, men den är oerhört komplex och lång

`VisibleRangeTrackLayer.Render()` är en del i den in-houseutvecklade kartmotorn som byttes ut som del i refaktoreringen som genomfördes. Funktionen använder sig flitigt av delegat, tupler och dålig variabelbenämning, enligt en utvecklare.

MapController		
moduläritet	-	Klassen sitter invävd i något som sätts upp i runtime
återanvändbarhet	-	Oklart, eftersom det är svårt att förstå hur den används
analyserbarhet	-	Svårt att se hur klassen används av andra klasser, pga indirection
modifierbarhet	-	Sitter för tight ihop med andra klasser, svårt att identifiera concerns
testbarhet	-	Svårt, man måste sätta upp omgivningen som koden för att det ska gå

MapController är en klass som hanterar kartor i systemet. Den använder sig av en del hjälpklasser, vars syften och uppgifter tycks flyta in i varandra. Klassen raderades ur kodbasen vid refaktoreringen, eftersom den inte behövdes när kartmotorn bytts ut.

SystemWorkItem		
moduläritet	-	Irrelevant här
återanvändbarhet	-	Funkar bara för det här systemet, men det är menat så
analyserbarhet	-	Delvis bra
modifierbarhet	-	Modifierbarheten är bra
testbarhet	-	Svårt att testa WorkItems, de är för generella

SystemWorkItem är det mest abstrakta WorkItem-objektet i projektet. Klassen för med sig andra avancerade koncept från CAB-arkitekturen till kodbasen, tex. SmartParts, som senare förkastades för att funktionaliteten egentligen inte behövdes.

Vector3		
moduläritet	-	Modulärt
återanvändbarhet	-	Man måste titta på koden för att förstå, men bra funktionalitet
analyserbarhet	-	Man förstår ju vad den gör
modifierbarhet	-	Ja då
testbarhet	-	Går lätt att testa

Vector3 är en supportklass som representerar tredimensionella punkter. Inga problem har identifierats med klassen, men den kastades ut under refaktoreringen eftersom bara kartmotorn använde den.

Algorithm		
moduläritet	-	Statisk klass med hjälpfunktioner, en utility-klass
återanvändbarhet	-	Är till för återanvändbarhet
analyserbarhet	-	Klassen är ok att förstå
modifierbarhet	-	Lägga till funktionalitet, ok, men att ändra? nej
testbarhet	-	Skulle säkert gå bra att testa

Algorithm är ytterligare en supportklass som tillhandahåller både användbara funktioner som `fold()` och enligt en utvecklare ”helt onödig funktionalitet, som omdefinitionen av C#:s egen foreach-loop med delegater”.

Tuple		
moduläritet	-	Ok, generisk hjälpklass
återanvändbarhet	-	Bra
analyserbarhet	-	Klockren
modifierbarhet	-	Bra
testbarhet	-	Ja, det är bara att testa

Tuple är en välimplementerad klass som representerar en tupel. Problemet med Tuple var att den användes överallt i koden, istället för att använda riktiga klasser eller strukturer för att representera data. Det försämrade läsbarheten hos koden väldigt mycket.

RovDataInputStream		
moduläritet	-	Väldigt beroende av att sitta ihop på ett visst sätt
återanvändbarhet	-	Dålig, pga beroende av att sitta ihop på ett visst sätt
analyserbarhet	-	Arvshierarkin gör det svårt, förstår inte syftet med klassen
modifierbarhet	-	Ja, det borde gå att lägga till saker
testbarhet	-	Svårt pga. beroenden och oklara concerns

RovDataInputStream är en klass för att kapsla indata från en sensor på en Remote Operated Vehicle (ROV). Klassen är del i en, enligt en utvecklare, konstlad arvshierarki, där det är svårt att få överblick över vilken del av funktionaliteten som hör till vilken klass och varför.

Klassen PersistentLayerSettingsService, med sin funktion `Unserialize()`, pekades ut på grund av funktionens höga cyklomatiska komplexitetstal, men behandlades inte i intervjuerna. Klassen var en del i den kartmodul som bytts ut av andra skäl, så inga relevanta uttalanden kunde göras.

Klassen SmarTrackInputStream, som pekades ut som problematisk på grund av sin plats i arvshierarkin, behandlades inte i intervjuerna, eftersom koden identifierats som död av de ursprungliga utvecklarna. Modulen är en del i ett misslyckat försök att utöka applikationens funktionalitet.

Kapitel 5

Analys

I det här kapitlet presenteras diverse tolkningar av data från resultatdelen.

5.1 Orsaker till respektive problem

Genom att sammanfatta citaten som tilldelades respektive kategori kan följande orsaker till de olika problemen presenteras.

1. Problem att för mycket mönster och arkitektur användes

   Koden upplevdes som komplex och beskrevs i termer som ”overengineering” och ”jättemaskineri”. I allmänhet ifrågasattes avvägningar mellan generalitet och enkelhet.

2. Problem att förstå syftet med delar av koden

   Koden ansågs använda CAB-arkitekturens WorkItems på en opraktiskt granulär nivå. Detta medförde att det var svårt att se hur objekt arbetade med varandra, eftersom syftet med WorkItems är att delar av koden ska kunna operera helt utan kännedom om varandra.

3. Problem att förstå hur en komponent beror på andra komponenter

   Utvecklare beskrev problem att ändra i delar av koden, därför att andra helt orelaterade komponenter då kraschade och svårigheter med att förstå hur komponenter samspelade, på grund av den höga graden av lös koppling.

4. Problem att se om koden användes alls

   Omfattande användning av lös koppling gjorde det svårt att se vilken kod som anropade den kod man tittade på

5. Problem med att förstå vad koden gör

   Dåligt namngivna variabler i datastrukturer försämrade läsbarheten markant och det var ibland svårt att särskilja objekt med väldigt liknande funktionalitet. Syftesfördelningen mellan objekt var inte helt klar. Debuggning vid krascher upplevdes också som opraktisk, då det ofta krävdes att köra igång hela systemet och använda sig av så kallade breakpoints för att se när t.ex. en funktion anropades. Detta på grund av lös koppling.

6. Problem när metoder inte är uppdelade ordentligt

   Det fanns metoder som ägnade sig åt felhantering och loggning utöver sitt syfte, vilket bitvis gjorde koden svårläst. Vidare förekom ofta metoder som gjorde betydligt mer än bara en sak, vilket ledde till dålig namngivning analyserbarhet.

7. Problem med att förstå tanken och syftet med koden

   Valet av datastrukturer försämrade analyserbarheten hos koden genom att kontexten för dataobjekt försvann när objektet skickades vidare till andra delar av koden. Bristande dokumentation hindrade förståelsen för kod som inte var uppenbar, samtidigt som dålig namngivning hos metoder och variabler försvårade analysarbetet. Ansvarsfördelningen mellan objekt ansågs inkonsekvent, vilket gjorde koden oförutsägbar.

8. Problem att se hur koden hänger ihop

   Nära koppling mellan ett fåtal klasser i en modul gjorde varje individuell klass svårförstådd, eftersom klasserna inte hade klart definierade uppgifter. Att koden länkades ihop i runtime snarare än i compiletime gjorde det väldigt svårt att läsa sig till hur koden fungerade. Inte heller gick det på ett enkelt sätt att uttyda vilka klasser som använde sig av vilka andra.

9. Problem med att avancerade eller onödiga konstruktioner används

   Koden återimplementerade språket inbyggda funktioner och återimplementerade standardiserade mönster på felaktiga sätt, vilket försämrade läsbarheten. I vissa av de jobbigare sektionerna har arv används som enda koppling mellan klasser, vilket har lett till ologiska och svårförstådda arvshierarkier.

10. Problem med att saker inte ligger på rätt plats i koden

    Grunden till problemet tycks vara slarvig ansvarsfördelning mellan klasser.

5.2 Överlappande problem

Vissa av de klasser som mätts är del i de problem som identifierats i den allmänna delen av intervjuerna. Det är intressant att se vilka av de allmänna problemen som berördes när de specifika klasserna diskuterades.

Problemen som identifierats i den specifika delen av intervjuerna kategoriseras här in under lämpliga problem ur den allmänna delen av intervjuerna med hjälp av de identifierade orsakerna (se stycke 5.1). Siffrorna som markerar respektive problem är hämtade ur listan som presenterades under 4.2 Teman och kategoriseringar.

1. Problem att för mycket mönster och arkitektur användes

   7. Problem med att förstå tanken och syftet med koden
   8. Problem att se hur koden hänger ihop
   9. Problem med att avancerade eller onödiga konstruktioner används

2. Problem att förstå syftet med delar av koden
    5. Problem med att förstå vad koden gör
    6. Problem när metoder inte är uppdelade ordentligt
    7. Problem med att förstå tanken och syftet med koden
    8. Problem att se hur koden hänger ihop
    10. Problem med att saker inte ligger på rätt plats i koden
3. Problem att förstå hur en komponent beror på andra komponenter
    6. Problem när metoder inte är uppdelade ordentligt
    7. Problem med att förstå tanken och syftet med koden
    8. Problem att se hur koden hänger ihop
    9. Problem med att avancerade eller onödiga konstruktioner används
    10. Problem med att saker inte ligger på rätt plats i koden
4. Problem att se om koden användes alls
    5. Problem med att förstå vad koden gör
    8. Problem att se hur koden hänger ihop
    9. Problem med att avancerade eller onödiga konstruktioner används

Här kan konstateras att kodmåtten identifierade klasser där det fanns problem. Alla av måtten utpekade klasser berörde något av de problem som identifierats i den allmänna delen av intervjuerna. Tydligt är att problemet med att se hur koden hänger ihop är det mest utbredda i den här kodbasen, vilket till och med gjort det svårt att identifiera död kod som sådan.

Kapitel 6

Diskussion

I det här kapitlet diskuteras metodvalet och resultatet från intervjuerna. Alternativa metoder diskuteras tillsammans med ett resonemang kring följder av den valda metoden. Resultatdiskussionen behandlar orsaker och följder till resultatets utfall.

Examensarbetets resultat kan kortfattat beskrivas som att samma problem diskuterades oavsett om man i intervjuerna diskuterade kodbasen i allmänhet eller de av kodmåtten indikerade klasserna. Strukturen hos intervjuerna gav en tydlig bild av hur utvecklarna bedömde klasserna som pekats ut av kodmåtten. Hög cyklomatisk komplexitet förknippas med bättre underhållbarhet än väntat, högt antal metoder i en klass verkar inte betyda något i sig själv och djup i arvshierarkin var en dålig indikator i den här studien.

6.1 Diskussion kring metodval

Kvalitativa intervjuer valdes som undersökningsmetod över en enkät eller mer tekniska lösningar delvis för att undersökningen kräver djup förståelse för att kunna härleda varför måtten indikerar problem och delvis för att det inte fanns så många utvecklare att tillgå från ett och samma projekt hos Attentec. Förkunskapen om koden var inte särskilt stor, vilket hade försvårat utformningen av en kvantitativ studie, då det upplevs svårt att ställa rätt sorters frågor utan tillräcklig förkunskap. En alternativ metod hade kunnat vara att titta på olika system och jämföra hur väl kodmåtten identifierar problem i de olika systemen. En sådan metod hade möjligtvis kunna tillhandahålla ett mer robust förtroende för vissa mått, men hade troligtvis erbjudit mindre förståelse för vilka fenomen i verkligheten som relaterar till mätningarna och vilka som missats.

ISO/IEC 25010's uppdelning av underhållbarhet användes för att underlätta analysarbetet och ge struktur och relevans åt resultatet. Alternativa uppdelningar hade säkert varit möjliga och hade kunnat ge lika goda resultat. Syftet var att styra intervjuerna kring relevanta ämnen, vilket kunnat uppnås på andra sätt också.

Inspirationen för analysmetoden för resultatet av de kvalitativa intervjuerna är utvald baserat på identifierad okunskap och tillgänglighet. Den huvudsakliga källan är en sammanställning av flera andra verk och har rekommenderats till studenter vid flera olika högskolor i Sverige. En av begränsningarna till validiteten i det här examensarbetet var min egen brist på kompetens på området intervjuteknik. Det är mycket möjligt att relevant information på det här området inte har identifierats och sannolikt att relaterade detaljer missats. Ändå har den sökta informationen funnits och behandlats på ett sätt som bedöms reproducerbart. Resultatet är rimligt givet den studerade kodbasen och hämtar

tillförlitligt i den struktur och metod som använts.

Ändå bedöms den metod som användes som lämplig och tillräcklig i den utsträckningen att den gav struktur åt behandlingen av data och möjliggjorde en extrahering av information för diskussion, samtidigt som den utgick från en källa som refererats flitigt inom utbildningsväsendet.

Viktigt att tänka på är att de av utvecklarna angivna problemen här bara jämförts med de klasser och funktioner som gav högst utslag vid mätning. Detta beslut utgår från den intuitiva förståelsen att högre värden är sämre, vilket antyds i litteraturen, exempelvis i [14]. Därför kan man gissa att de *sämsta* värdena kommer matcha de *värsta* problemen, men det är naturligtvis möjligt att så inte är fallet.

Vidare har måtten betraktats i isolering, vilken möjligtvis inte är optimalt. Det är mycket väl möjligt att mått behöver användas i kombination för att förutsäga eller indikera något meningsfullt.

Viktigt är naturligtvis också att man här redan på förhand vet att kodbasen har refaktorerats från det tillstånd den befann sig i under undersökningen i det här arbetet. I andra kodbaser kanske utvecklare är mindre säkra i sina uttalanden om vad som är problematiskt i kodbasen, vilket kan leda till betydligt mindre användbara resultat.

En analys som inte gjorts är huruvida mindre problematiska klasser har samband med klasser som mått indikerar är underhållbara. Det hade kunnat vara intressant i den här kodbasen, eftersom de stora problemen inte tycktes vara de mest komplexa eller stora objekten.

6.2 Diskussion kring resultat

Tydligt är att de klasser och funktioner som har högst mätvärden från de utvalda kodmåtten är del i de stora problemen i kodbasen. Detta torde innebära att man i Attentecs ställe skulle kunna utgå från av kodmått utpekade klasser under analysen av en kodbas innan ett uppdrag och därifrån få en god uppfattning av kvaliteten på kodbasen. Måtten själva verkar däremot inte peka ut de väsentliga problemen i koden, utan pekade på objekt som bara ibland försämrade kodbasens underhållbarhet, och inte alltid av den anledning som anges i litteraturen. Det kan tänkas att anledningen till att problemen sammanfaller med komplexa objekt i huvudsak beror på att komplexa objekt gör många saker, alltså har de större sannolikhet att vara involverade i större problem.

Övergripande kan anmärkas att det är intressant att tidigare forskning antytt att måtten på ett tillförlitligt sätt förutspår underhållbarhet [11], men att den här studien tyder på att de undersökta måtten inte själva fångar vad utvecklarna anser var de största problemen. Det är möjligt att den här kodbasen är ett sådant falsklarm som t.ex. Ferreira et al. [13] stött på, där måtten indikerar dålig koddesign trots att koden anses bra eller vice versa.

En uppenbar felkälla här är att de mått som undersökts helt enkelt inte är tillräckliga för att identifiera just de problemen som förekom i den här kodbasen, varvid man kan fråga sig om det existerar en heltäckande mängd mått som fångar, eller åtminstone indikerar, alla sorters underhållbarhetsproblem.

En annan orsak till det avvikande resultatet kan vara att utvecklarna själva inte strikt anger problem som hänger ihop med underhållbarhet, utan kanske pekar ut de problem som har hindrat dem i tidigare projekt eller endast pekar ut de problem som de inte själva orsakat. Det senare bedöms i det här fallet ha liten påverkan, eftersom majoriteten av

koden i kodbasen inte skrivits av Attentecs konsulter.

6.2.1 Lös koppling och underhållbarhet

En intressant notering är att sk. *loose coupling* var ett problem i den här kodbasen. Hur kan man mäta det? Lös koppling ses normalt som något bra i en kodbas, men det verkar inte finnas så mycket konkret forskning på det. Det är intressant här, eftersom delegatfunktionerna i C# är en typ av lös koppling, men orsakade här betydligt fler problem än de löste.

Mått som kan användas för att mäta lös koppling är samma som mäter koppling. Problemet är att i normalfallet strävar man efter en låg koppling för att minska risken för att buggar propagerar genom systemet, men i det här fallet hade en högre koppling varit lämpligare för läsbarheten hos koden. Hur man ska mäta om lös koppling är lämplig eller inte ligger utanför den här rapporten, men det känns nästintill omöjligt. Det hade varit intressant med en studie som behandlar hur lös koppling relaterar till läsbarhet.

6.2.2 Cyklomatisk komplexitet

Måttet cyklomatisk komplexitet pekade ut funktioner i klasserna PersistentVideoView, PersistentVideoSnapshotGrabberService och VisibleRangeTrackLayer. I de två första fallen beskrev utvecklarna de utpekade funktionerna som långa, men underhållbara. Det var alltså här inget problem att funktionerna hade hög cyklomatisk komplexitet, utan det gjorde bara funktionen längre, vilket ansågs oöverskådligt. Testbarheten hos funktionerna ansågs acceptabel, vilket går emot exempelvis Visser et al. [14], som anser att en funktions cyklomatiska komplexitet inte bör överstiga 5, främst för att förenkla testningen av funktionen.

Man kan fråga sig om funktionerna med hög komplexitet ansågs underhållbara i det här systemet därför att den lösa kopplingen i allmänhet var ett så mycket större problem. Funktioner anropades ofta med delegat, vilket innebär att det den anropade funktionen utför är separerat från funktionen självt. Komplexa funktioner där man faktiskt kan utläsa vad de gör kanske då anses enkla att förstå även om de vanligtvis hade ansetts komplicerade.

6.2.3 Antal metoder i en klass

Måttet antal metoder i en klass (NMC) är menat att mäta komplexiteten hos ett objekt som antalet komponenter objektet är sammansatt av. Resonemanget bakom måttet är att antalet metoder är en indikation på ansträngningen som krävs att utveckla och underhålla klassen, att med ett större antal funktioner blir kopplingen med barnklasser större och klasser med fler funktioner är mer sannolikt applikationsspecifika [4]. Visser et al. [14] anser också att en klass med många funktioner ökar kopplingen mellan objekt i en kodbas. Klasserna MapController, SystemWorkItem, Vector3 och Algorithm hade högst mätvärden i den här kategorin.

MapController är mycket riktigt invävd i ett till synes oförståeligt nät av nära kopplade klasser och anses ej återanvändbar. En av huvudanledningarna till svårigheterna med klassen tycks vara att det är svårt att uttyda vilka syften de olika samverkande klasserna fyller. Detta stämmer bra överens med litteraturen.

SystemWorkItem är ett väldigt abstrakt objekt som återanvänds frekvent genom hela kodbasen. Den höga graden i vilken klassen återanvändes inom systemet tycks väga upp dess höga komplexitet, då en stor förståelse för den här klassen krävs oavsett vilken del av systemet man utvecklar.

Vector3 och Algorithm är däremot hjälpklasser. Vector3 är ett matematikbibliotek med ett väldigt avgränsat syfte och är därmed enkel att förstå. Det kan tänkas att den här klassens metoder endast använder sig av klassens variabler, och inte anropar andra klasser i särskilt hög utsträckning. Här kan man göra analysen att antalet metoder i en klass är en dålig indikation på hög koppling mellan objekt, åtminstone utan input från mått som mäter hur väl metoder samspelar med varandra. Algorithm är en statisk klass med hjälpfunktioner som är skapad för återanvändbarhet. Problemet med den här klassen är att den implementerar helt onödig funktionalitet. Det mättes inte alls av måtten, men det hade jag inte väntat mig heller. Att mäta huruvida syftet med ett kodsegment är relevant tycks mig väldigt svårt.

6.2.4 Djup i arvshierarkin

Klasserna Tuple och RovDataInputStream pekades ut som problematiska på grund av sin arvshierarki. Litteraturen pekar på att klasser som ligger långt ned i arvsträdet är svårare att förstå, därför att de ärver funktioner från fler klasser, och har fler funktioner än andra klasser [4]. Måttet går alltså delvis in i NMC, som behandlas ovan.

Tuple är en välimplementerad hjälpklass och led inte alls av de symptom som måttet anses peka ut. Man kan tycka att det är konstigt att implementera en Tuple-klass själv, men klassen Tuple hade när kodbasen började utvecklas inte införts i .NET. Det huvudsakliga problemet med den här klassen var minskad läsbarhet i hela projektet genom att man överallt använde en enda datastruktur med generiska namn för att hantera data i systemet, istället för specifika strukturer för varje syfte. Detta försämrade läsbarheten avsevärt, men det borde vara väldligt svårt att mäta något sådant med kodmått.

RovDataInputStream är faktiskt en klass som är jobbig att vidareutveckla på grund av sin arvshierarki, men det är tveksamt om det är djupet på hierarkin som är den stora faktorn. Man kan tänka sig att användandet av arv i det här fallet var felplacerat och att det är därför hierarkin blivit ”konstlad” och därmed kanske också djupare.

6.3 Samband mellan mått och underhållbarhet

Här sammanfattas hur de olika aspekterna av underhållbarhet [1] relaterar till de klasser som pekats ut som problematiska av kodmåtten.

6.3.1 Cyklomatisk komplexitet

Cyklomatisk komplexitet pekade inte ut några klasser med moduläritetsproblem. Återanvändbarheten hos funktioner med hög cyklomatisk komplexitet tycks vara genomgående låg. Huruvida koden går att analysera och modifiera verkar bero på andra faktorer än cyklomatisk komplexitet, eftersom det skiljer sig markant mellan funktionerna som undersökts. Det verkar råda tveksamhet till om funktionerna är testbara eller inte. Kort sagt tyder resultaten på att hög cyklomatisk komplexitet kan kopplas till dålig återanvändbarhet och dålig testbarhet, men inte nödvändigtvis dålig analyserbarhet.

6.3.2 Antal metoder i en klass

Klasser med högt antal metoder verkade inte mindre modulära än andra klasser, eftersom svaren kring moduläritet skiljde sig. Det går heller inte att göra något uttalande om återanvändbarheten hos kod med högt antal metoder i en klass. Analyserbarheten verkar här genomgående god, medan modifierbarheten och testsbarheten varierar. Möjligtvis beror det här på att funktionaliteten delats upp till tillräckligt små stycken för att det ska vara lätt att ta till sig. Många metoder i en klass hade i så fall kunnat indikera ökad analyserbarhet, förutsatt att metoderna är korta, jämfört med en klass med få, men långa metoder.

6.3.3 Djup i arvshierarkin

Av de två klasserna som klassades som problematiska på grund av sin djupa position i arvshierarkin kan sägas att klasserna verkade vara antingen bra på alla punkter, eller dålig på alla punkter. Här krävs alltså förmodligen att måttet används i kombination med något annat mått för att faktiskt kunna förutsäga något om underhållbarheten.

6.4 Etiska perspektiv

Det kan tänkas att illasinnade bolag eller utvecklare vill använda det här arbetet för att truga på en kund någon typ av onödigt inköp. Kanske har kunden inte tillräcklig kunskap för att själv göra en rimlighetsbedömning kring en föreslagen refaktorering, eller kan inte själv tolka kodmått. Om kodmått då presenteras som säljargument med den falska motiveringen att liknande situationer har lett till problem tidigare kan kunden ledas att tro att en utvecklare ser något i kodmåtten som kunden inte ser, även fast så inte är fallet, och beställa en fullständigt onödig refaktorering. För att minska risken för detta vill jag understryka att den här rapporten på inget sätt drar slutsatsen att mätvärden är att lita på. Här har vi visat att kodmått genom omsorgsfull tolkning kan visa på klasser som kan vara involverade i problem i en kodbas, om det finns några. Att refaktorera för att kodmåttens värden är höga är inte att rekommendera, gör hellre en professionell bedömning.

Ett annat möjligt scenario är att kodmått av Attentecs säljare överanvänds för att sälja fler konsulttimmar. Därmed försvinner Attentecs mål att sälja en så effektiv lösning som möjligt till sina kunder. Det är tänkbart att Attentec i lednings- och affärsplaneringssammanhang vill använda sig av den här typen av strategier, även om de utvecklare som utför själva bedömningarna och arbetet vill leverera en så skräddarsydd lösning som möjligt. Konsultbolag tjänar ju trots allt pengar på antal sålda konsulttimmar, och om det finns verktyg som gör att man kan tjäna mer pengar är det kanske frestande att använda dem.

6.4.1 Konfidentialitet

I det här arbetet döljs identiteten både hos systemet som undersöks, de utvecklare som skrivit koden och de konsulter som deltagit i undersökningen. Man kan fråga sig hur relevant det är att kritisera en kodbas som började utvecklas cirka 15 år innan studien gjordes (även om vi tittat på koden som den såg ut 10 år efter att den började utvecklas).

Hur som helst avser inte arbetet kritisera de fackmässiga kunskaper dessa personer besitter, eller sättet på vilket programvaran konstruerats. Jag har förstått att fenomenet med löskoppling var ett problem med flera kodbaser från 2003-2006. Syftet med att följa alla identiteter är helt enkelt att personerna och företagen i fråga inte ska påverkas negativt av ett arbete som analyserar vad som gått fel i ett projekt som tidigare genomförts.

Det material som samlats in, transkribering av intervjuerna, loggar och anteckningar förstörs när det här arbetet är färdigt.

Kapitel 7

Sammanfattning

Vi kan konstatera att de utvalda kodmåtten verkar peka på en bra grupp med klasser som startpunkt för manuell analys av underhållbarheten hos en kodbas. Alla stora problem som identifierats berördes under diskussioner kring de klasser som pekats ut. I det här systemet verkade dock i regel måtten inte ge utslag av den anledning som angetts i litteraturen, med vissa undantag.

Den cyklomatiska komplexiteten hos en klass tycktes i det här systemet snarare indikera en underhållbar klass än en klass med låg underhållbarhet, vilket troligtvis relaterar till den höga graden av lös koppling i kodbasen. Lös koppling var här så utbrett att samspelet mellan klasser i koden ibland var snudd på omöjlig att förstå.

Antal metoder i en klass kan vara en bra indikation på problematiska klasser, men man ska komma ihåg att bara för att värdet är högt betyder det inte att klassen är dålig ur underhållbarhetssynpunkt. Måttet kan tänkas vara betydligt mer användbart i kombination med ett mått på sammanhållning eller koppling. Klasser med många metoder, men hög sammanhållning mellan metoderna kan fortfarande anses underhållbara. I ett resonemang kring ISO/IEC 25010's uppdelning av underhållbarhet verkar måttet i det närmaste värdelöst.

Djup i arvshierarkin verkar inte vara ett tillräckligt mått för att förutsäga om en utpekad klass är ett problem eller inte.

7.1 Slutsats

Kodmått finns i den här studien inte tillförlitliga som direkt indikator för om en refaktorering är på sin plats eller ej. Man kan däremot utgå från de klasser som kodmått antyder har låg underhållbarhet när man gör en professionell bedömning av kodens underhållbarhet.

7.2 Framtida forskning

Framtida forskning skulle kunna bestå i att utvärdera kombinationer av mått som gör faktiska förutsägelser snarare än generella indikationer. På grund av att mjukvaruutveckling kan se så olika ut på olika håll bör vidare forskning fokusera på väldigt specifika områden, till exempel se om man lyckas hitta en kombination av mått som identifierar en skadlig klass med för många funktioner i 95% av fallen.

Forskning på delegatfunktioner som programmatisk teknik är också relevant. Hur bidrar den sortens lös koppling till kodbasen? Vilka kopplingar kan göras till kvalitet, underhållbarhet och säkerhet? Är det intressant att ta fram mått som mäter graden av delegeringar? Är det samma som måtten på loose coupling?

Litteraturförteckning

[1] *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models - ISO/IEC 25010:2011*. 2013.

[2] Microsoft composite ui application block, 3 Maj 2017. URL `https://msdn.microsoft.com/en-gb/library/ff648747.aspx`.

[3] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2 (4):308–320, Dec 1976. ISSN 0098-5589. doi: 10.1109/TSE.1976.233837.

[4] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, Jun 1994. ISSN 0098-5589. doi: 10.1109/32.295895.

[5] ISO/IEC/IEEE. Systems and software engineering – vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, pages 1–418, Dec 2010. doi: 10.1109/IEEESTD.2010.5733835.

[6] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY, USA, 1977. ISBN 0444002057.

[7] Rüdiger Lincke and Welf Löwe. Validation of a standard- and metric-based software quality model. In *IN: PROCEEDINGS OF THE 10TH ECOOP WORKSHOP ON QUANTITATIVE APPROACHES IN OBJECT-ORIENTED SOFTWARE ENGINEERING (QAOOSE*, pages 81–90, 2006.

[8] Alain Abran. *Cyclomatic Complexity Number: Analysis of Its Design*, pages 131–143. John Wiley & Sons, Inc., 2010. ISBN 9780470606834. doi: 10.1002/9780470606834.ch6. URL `http://dx.doi.org/10.1002/9780470606834.ch6`.

[9] Alain Abran. *Halstead's Metrics: Analysis of Their Designs*, pages 145–159. John Wiley & Sons, Inc., 2010. ISBN 9780470606834. doi: 10.1002/9780470606834.ch7. URL `http://dx.doi.org/10.1002/9780470606834.ch7`.

[10] M. Bunge. *Treatise on Basic Philosophy: Ontology I : The Furniture of the World*. Boston: Riedel, 1977.

[11] W. Li and S. Henry. Maintenance metrics for the object oriented paradigm. In *[1993] Proceedings First International Software Metrics Symposium*, pages 52–60, May 1993. doi: 10.1109/METRIC.1993.263801.

[12] Rüdiger Lincke, Jonas Lundberg, and Welf Löwe. Comparing software metrics tools. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 131–142, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-050-0.

doi: 10.1145/1390630.1390648. URL http://doi.acm.org.focus.lib.kth.se/10.1145/1390630.1390648.

[13] Kecia A.M. Ferreira, Mariza A.S. Bigonha, Roberto S. Bigonha, Luiz F.O. Mendes, and Heitor C. Almeida. Identifying thresholds for object-oriented software metrics. *Journal of Systems and Software*, 85(2):244 – 257, 2012. ISSN 0164-1212. doi: http://dx.doi.org/10.1016/j.jss.2011.05.044. URL http://www.sciencedirect.com/science/article/pii/S0164121211001385. Special issue with selected papers from the 23rd Brazilian Symposium on Software Engineering.

[14] Joost Visser, Sylvan Rigal, Rob van der Leek, Pascal van Eck, and Gijs Wijnholds. *Building Maintainable Software, Java Edition: Ten Guidelines for Future-Proof Code*. O'Reilly Media, Inc., 1st edition, 2016. ISBN 1491953527, 9781491953525.

[15] Vizzanalyzer, 13 Mars 2017. URL http://www.arisa.se/vizz_analyzer.php.

[16] Ndepend, 13 Mars 2017. URL http://www.ndepend.com/.

[17] Axivion bauhaus suite, 13 Mars 2017. URL https://www.axivion.com/en/products-services-9\#products_bauhaussuite.

[18] Finn Wiedersheim Lars Torsten Eriksson. *Att utreda, forska och rapportera*. Liber AB, 10th edition, 2014. ISBN 978-91-47-11169-5.

[19] Anna Hedin. En liten lathund om kvalitativ metod med tonvikt på intervju, 15 Mars 2011. URL https://studentportalen.uu.se/uusp-filearea-tool/download.action?nodeId=459535\&toolAttachmentId=108197.

Bilaga A

Definitioner av underhållbarhet

Följande definitioner gjordes tillgängliga för de svarande i intervjuerna. De är översatta från [1].

1. moduläritet
   (a) Grad till vilken ett system eller datorprogram är komponerat av diskreta komponenter så att en förändring i en komponent har minimal effekt på andra komponenter.
2. återanvändbarhet
   (a) Grad till vilken en resurs kan användas i mer än ett system, eller till att bygga andra resurser
3. analyserbarhet
   (a) Grad av effektivitet med vilken det är möjligt att bedöma inverkan på ett system av en önskad förändring av en eller flera av dess delar, eller att utvärdera en produkt med avseende på felkällor, eller för att identifiera vilka delar som ska modifieras.
4. modifierbarhet
   (a) Grad till vilken en produkt eller system på ett effektivt sätt kan modifieras utan att introducera fel eller minska den existerande produktens kvalitet.
5. testbarhet
   (a) Grad av effektivitet med vilken testkriterier kan etableras för ett system, en produkt eller en komponent och test kan utföras för att bestämma huruvida dessa kriterier har uppnåtts.

Bilaga B

Citat från intervjuerna

B.1 Citat ur allmänna intervjudelen och deras kategorisering

Problem att för mycket mönster och arkitektur användes

1. Problemet var att dels var den väldigt komplext uppbyggt, det var väldigt mycket design patterns och väldigt mycket indirection.

2. Det var också väldigt väldigt komplext för att det var interfaces, indirektion och delegater och callbacks precis överallt och de var inte vettiga på något sätt, de var delegater för sin egen skull.

3. Det var det här med overengineering [, det var ett problem]

4. Det var en sorts flexibilitet och det var generellt, men generalitet kostar ju.

5. Han har skapat ett jättemaskineri bara för att skapa nya textfält, för att man ska behöva skriva så lite kod som möjligt för att lägga till ett nytt fält.

6. Helt felaktiga avvägningar av vilken sort flexibilitet och utökningsbarhet som behövs.

7. Varje dialogbox, varje liten input, varje liten service, varenda liten modul var ett WorkItem och de är ganska komplexa som måste skapas på ett visst sätt, initieras och sen tas hand om, tas bort.

Problem att förstå syftet med delar av koden

8. Alla de här WorkItems var inte testade, och inte testbara, därför att det gick inte att förstå vad poängen med dem var.

Problem att förstå hur en komponent beror på andra komponenter

9. Det gjorde att om man försökte fixa an bugg någonstans, det gick liksom inte att se hur det påverkade det övriga systemet med mindre än att köra det i runtime och debugga det.

10. Försökte man fixa en ny feature, så small det på tre andra ställen på grund av beroenden som inte gick att föstå genom att se koden.

11. Så ingen visste någonting om någon annan, vilket gjorde att det gick inte att förstå hur någonting hängde ihop.

12. Den [kartmodulen] var ju då väldigt avancerad och klarade av väldigt stora datamängder, men det var ingen som förstod den eller kunde lägga till saker när han hade lämnat företaget.

13. Det [allt i kartmodulen] var för invävt så den var ett stort problem också.

14. Allt var en egen modul som inte visste någonting om någon annan och sen satts allt ihop i runtime.

15. Allting sattes ihop i runtime, det gick inte att förstå hur saker och ting hängde ihop med mindre än att köra koden och tracea den.

16. Allting var så indirekt och löskopplat, så analyserbarheten var total crap, det var det grunläggande problemet.

17. Andra utvecklare brukade beskriva det som att de petade lite försiktigt i ena delen av appliaktionen och så *pang* exploderade en helt annan del.

Problem att se om koden användes alls

18. För att se om någon kod användes var man i princip tvungen att sätta en brytpunkt där och använda systemet och se om man kom dit.

19. Det gick inte att utläsa ut kodbasen [om en specifik kodsnutt kördes] på ett vettigt sätt för det var så mycket delegater.

B.2 Citat ur specifika intervjudelen och deras kategorisering

Problem med att förstå vad koden gör

20. Den [kartmodulen] kraschade och det gick inte att förstå krascherna tillräckligt väl.

21. Datat den [funktionen] tar emot är en tupel av en string och en nullable double och vad strängen är och vad doublen är och vad det innebär när doublen är null är sådant man bara förväntas veta.

23. Det här går ju inte att förstå, med rimlig insats.

27. Vad som jag tycker är dåligt är att jag får ingen översikt, jag förstår inte, vad är det för skillnad på de här båda, GenericDataInputStream och DataInputStream?

Problem när metoder inte är uppdelade ordentligt

28. Det känns som att minst hälften [av den här långa funktionen] är felhantering.

29. Den här metoden skulle jag säga är alldeles för lång och borde splittas upp i fler.

30. Den hanterar en massa filer för att ta en snapshot... det borde inte behövas.

Problem med att förstå tanken och syftet med koden

31. Jag förstår inte riktigt varför den ser ut som den gör.

32. Jaha, en fold som tar parametrar och en delegat, som returnerar en tuple och skapar en tuple från en annan tuple?

33. Den kan man förstå om man har lite dokumentation, men de här tre första [parametrarna] vad gör de här?

34. Paint() och repaint(), det är lite svårt att veta när de triggas, är det en uppmaning eller är det en information om att det har hänt?

33. ClearHistory()? Vad fan är history i det här sammanhanget? RemoveTimeConstraint()? Från vem? Var kom den ifrån?

34. Det är oklart för mig vad MapController-klassen har för concerns.

Problem att se hur koden hänger ihop

22. Det är ju massvis som kan gå fel, jag är lite osäker på om den städar upp efter sig.

35. Skickar du in fel sorts parametrar här så säger den nog kaboom.

24. Ja, jag vågar inte ändra i den, jag har ingen aning om vad som skulle hända.

36. Petar man på något här så kommer garanterat något gå sönder.

25. Det är massa indirection här, som är ganska opak.

37. Den verkar göra massor med saker och ibland triggar den event med något som någon annan uppenbarligen ska göra, vem då?

26. Jag sökte mig uppåt här [i en call-stack] och hamnade i en polymorfisk hierarki som liksom grenade ut sig och det var svårt att se vem som anropade vem.

38. Den här sitter någonstans i mitten och skyfflar information fram och tillbaka och sen finns det vissa saker som den gör också verkar det som, men i stor utsträckning verkar den vara någon sorts spindel i nätet som skyfflar information.

39. Den här sitter ju uppenbarligen invävd i något komplext nät av klasser som måste sättas upp i runtime, så här är ett typiskt exempel på att ändrar man någonting här, så går något annat sönder, för att det sitter ihop på något sätt som inte är uppenbart, med en väldig massa andra saker.

40. Den är beroende av andra klasser som skall anropa den och ta emot event från den, så det är väldigt svårt att veta vad som kommer hända om man ändrar något här, vad det kommer påverka för något.

41. Jag försöker hitta något som faktiskt använder den...

42. Det jag skulle bli tvungen att göra är att sätta en brytpunkt här och sen köra igång programmet för att se om den faktiskt används eller om det är död kod.

43. Jag skulle sätta en brytpunkt här och se vad som hände om jag ville veta det.

44. Den här har liksom settings, nån settingsklass som kommer in någonstans, problemet är kanske inte så mycket arvshierarkin, tror jag, utan alla andra klasser den beror av också.

Problem med att avancerade eller onödiga konstruktioner används

45. Bara att använda den inbyggda språkkonstruktionen hade ju varit enklare istället för att anropa en statisk funktion för det.

46. Här här du en fullständigt onödig indirection.

47. Varför komplicera saker så mycket?

48. Det finns mönster för hur man ska implementera det här, men han har hittat på ett eget.

49. Att använda de här jättemekanismerna för det, då blir allt mycket mer komplicerat än det behöver va.

50. Istället för att skriva foreach och sen loopa, så anropar man en metod och skickar in en delegat som gör det som ska stå i loopen.

51. Han har implementerat foreach själv. För att? Det här är en av de största källorna till irritation och svårläst kod i kodbasen, för den här används precis överallt.

52. Att man använder Tupler istället för klasser med specifika variabelnamn, så att man förstod vad det var för någonting.

53. Han har försökt generalisera saker och ting som inte borde generaliseras.

54. Det är en väldigt konstlad arvshierarki, det är liksom inte en ”is a, is a, is a” utan det är väldigt, jag vet inte vad man ska kalla det, implementationscentrisk arvshierarki, att den här är ett barn av den, därför att den där gör saker som den här också behöver göra och då ärver vi av den och lägger till lite nytt stuff.

Problem med att saker inte ligger på rätt plats i koden

55. Den verkar både äga själva COM-ports hanteringen, alltså att ta in data från en COM-port, och att parsa data och skicka det vidare och det tycker jag sitter ihop lite för mycket och det ena ligger i en basklass till den här.

56. Det är lite tokigt, man skulle lika gärna kunna lägga arvshierarkin precis tvärt om om man ville och det är ett klart tecken på att det inte makes sense att ha en arvshierarki.

57. Så här ska det visas på skärmen, vy-logik mitt i objektet som öppnar COM-portar och vet hur man parsar datat.