HÖGSKOLAN
I GÄVLE

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

Kodanalys med mjukvarumetriker

En fältstudie hos Monitor ERP System AB

Pontus Sund

2017

Examensarbete, Grundnivå (högskoleexamen), 15 hp
Datavetenskap
Dataingenjörsprogrammet

Handledare: Åke Wallin
Examinator: Jonas Boustedt

Tack

Jag vill tacka Monitor ERP System AB för att de lät mig utföra detta arbete hos dem och min handledare för sin vägledning.

Abstrakt

Det finns inga definitiva överenskommelser om vad som är ”bra” programkod vilket gör det svårt att mäta kvaliteten hos programvara. Ändå är det viktigt för företag som säljer mjukvara att hålla hög kvalitet på koden för att underlätta ändringar när nya funktioner introduceras. Metriker kan användas för att mäta olika aspekter på objektorienterad design. Detta arbete undersöker (1) vilka metriker som är intressanta för Monitor ERP System AB, (2) vilka värden som är rekommenderade för dessa metriker, (3) vilka värden metrikerna producerar utifrån Monitors kodbas och (4) om värdena stämmer överens med framtagna rekommendationer. För att göra detta undersöks tidigare studier bland olika databaser, ett program skapas för att ta fram metrikvärdena och relativa gränsvärden genereras utifrån dessa. Till sist utförs en analys, där det visar sig att kodbasen till stor del följer rekommenderade gränsvärden.

Keywords: mjukvara, metriker, gränsvärden

Innehåll

1 Inledning

Lämpliga mätvärden för olika typer av företag ger möjligheten att göra välgrundade påståenden om kvaliteten på deras tjänster och produktivitet. Till exempel, i projektbaserade arbeten är det relativt enkelt att avgöra om tidsplanen följs och om alla milstolpar nås. I fall en produkt tillverkas åt en kund ska kvaliteten hos tjänsten hållas hög. För att försäkra sig om att kvaliteten tillfredsställer kunden kan det vara önskvärt att kunna ta fram statsistik om produkten eller tjänsten på ett snabbt och enkelt sätt. Inom mjukvaruutveckling är det relativt svårt att bestämma vad som är ”bra” kod eller inte av olika anledningar. Genom att använda metriker för att ta fram numersika värden kan man göra mätningar av källkod. En metrik är ett sätt att mäta olika egenskaper av mjukvara. Det är nödvändigt att förstå innebörderna av metrikerna och få en uppfattning om vad dess värden betyder.

Monitor ERP System AB [1] har huvudkontor i Hudiksvall och säljer affärssystem till tillverkande företag. De utövar agila metoder för mjukvaruutveckling som bl.a. Scrum. Story Points används för att mäta utvecklingshastighet; ett team har som mål att klara av 70% av det arbete som bestämdes i början av en sprint. Varje sprint pågår vanligtvis i två veckor. Buggar rapporteras via Team Foundation Server [2] och innehåller en beskrivning av buggen, personal som avklarat olika steg av rättningen och eventuellt finns det länkar till relaterade tasks och git commits och ansvarigt team. Monitor utvecklar deras produkt med programmeringsspråket C# i Windowsmiljö. Kvaliteten på Monitors kod är viktig för de som företag. Hög kvalitet låter dem utveckla nya funktioner, göra ändringar och rätta till buggar utan att introducera nya. Monitor använder idag inga metriker för källkod. För att hålla hög kvalitet görs kodgranskningar av medarbetarna. Man vill nu undersöka mätetal som är relevanta för Monitor.

1.1 Syfte

Arbetet undersöker kodbasen för Monitor Affärssystem med hjälp av utvalda mjukvarumetriker. Metrikerna mäter olika aspekter av underhållbarhet. För att sätta resultaten inom ett sammanhang används tidigare erfarenheter och forskning från litteraturen. Syftet är att analysera källkoden och att identifiera delar av kodbasen som inte följer rekommenderade gränsvärden.

1.2 Frågeställningar

För att uppnå syftet från föregående kapitel svarar arbetet på följande frågor:

1. Vilka metriker är relevanta för Monitor?
2. Vad anses vara bra värden för utvalda metriker enligt tidigare studier?

3. Vilka mätvärden erhålls med valda metriker applicerade på kodbasen?
4. Hur förhåller sig resultaten till tidigare forskning och erfarenheter?

Punkt 1 innebär att tidigare studier undersöks. Vad som avgör om en metrik är intressant baseras på mängden teoretiska och empiriska undersökningar som finns tillgängliga och hur relevanta de är för Monitor. Punkt 2 ämnar ge sammanhang till resulterande värden för varje metrik. Punkt 3 handlar om att presentera resultaten från kodmätningen och eventuellt uppmärksamma intressanta delar som t.ex. extremvärden. Till sist, med punkt 4 analyseras resultaten och slutsatser dras.

1.3 Avgränsning

Med tiden som har givits till detta arbete är det inte möjligt att implementera alla metriker som finns och jämföra dem med olika gränsvärden. Detta arbete fokuserar därför mest på metriker som har studerats och empiriskt undersökts mycket relativt till andra metriker. När det gäller gränsvärden för varje metrik väljs endast några få ut p.g.a. tidsbrist.

1.4 Bakgrund

I följande avsnitt ges bakgrund till vad som är vanliga sätt att mäta på källkod, vad som kan argumenteras vara goda kvaliteter hos kod och några välkända metriker för källkod.

1.4.1 Metriker

Mjukvarumetriker (engelska *software metrics*) är sätt att utföra mätningar på mjukvara. En metrik kan liknas med en matematisk funktion: utifrån en parameter (källkod) kan den framställa ett resultat (mätvärde). Nedan kommer några exempel på metriker som hittats i litteraturen och beskrivs kortfattat.

McCabe presenterade cyclomatic complexity [3], en metrik som beskriver komplexiteten hos en funktion. Den gör det genom att mäta antalet unika vägar genom metoden. Givet en metod med endast en villkorssats kommer den metoden ha en cyclomatic complexity (CC) på 2 (även kallat McCabe-värdet). McCabe samarbetade med ett team Fortran-programmerare och hittade en korrelation mellan storleken på CC och erfarna mjukvaruutvecklares åsikter om komplexiteten hos moduler. Ju mer komplex modul, desto högre värde på CC. Som konsekvens är moduler med höga CC-värden svårare att enhetstesta och bör refaktoreras. McCabe ansåg i sin studie att metoder med värden över 10 är svåra att underhålla och bör skrivas om.

Halstead Metrics använder antalet operatorer och operander i koden för att beräkna olika värden: Vocabulary, Volume, Difficulty, Effort, Errors och Testing Time [4].

C.K Metrics suite är en uppsättning metriker framtagna för att mäta olika aspekter av objektorienterad design [5]. Dessa är: *Weighted Methods per Class* (WMC), *Depth of Inheritance Tree* (DIT), *Number of Children* (NOC), *Coupling between object classes* (CBO), *Response for a Class* (RFC) och *Lack of Cohesion* (LoC). Enligt studien från Arvanitou m.fl. [6] är dessa metriker väldigt vanliga och välkända inom området. Metrikerna är grundade på matematiska principer och är anpassade för objektorienterad design, något som författarna ansåg vara vanlig kritik mot tidigare metriker som var anpassade för strukturerade programspråk.

1.4.2 Problem med mätning

Fenton och Neil [7] skriver att tidiga försök att mäta en programmeras produktivitet var att räkna antalet kodrader som skrivs (Lines of Code, förkortat till LoC). De argumenterar att LoC inte är passande för programmeringsspråk med höga abstraktionsnivåer jämfört med Assembler-språk. LoC reflekterar inte kodens funktionalitet, komplexitet eller hur mycket ansträngning som behövdes för att skriva den. Däremot besitter den tre lockande egenskaper för ett mätvärde; LoC är enkel att beräkna, förståelig även för icke-programmerare och den är språkoberoende [8]. Fowler [9] skriver att LoC kan leda till dåliga programmeringsvanor som att kopiera och klistra in kod (om målet är att få ett högt värde för LoC), men den är användbar för att mäta storleken på ett program.

Ett problem med att använda olika metriker är att avgöra vad de resulterande värdena har för innebörd. Som beskrivet ovan mäter LoC inte kvalitet på kod. Istället mätern den storleken på ett program eller en modul som i sin tur kan användas tillsammans med andra metriker för att exempelvis göra en bedömning om kvalitet. Ett program med tio kodrader kan inte anses vara bättre än ett program med elva kodrader. Däremot om man applicerar flera metriker och upptäcker att programmet med elva kodrader är mer komplext ur ett visst perspektiv, kan man göra en bedömning om att det är ”sämre”. Det krävs att man sätter metriker i ett sammanhang för att det ska bli möjligt att ta viktiga beslut baserat på dem och att man noggrant väljer metriker beroende på vad man vill åstadkomma.

1.4.3 Kvalitet på kod

För att kunna mäta kod måste det finnas en förståelse för vad som utgör hög och låg kvalitet på kod. I detta avsnitt redogörs olika kvalitetsaspekter för programmeringskod.

Arvanitou m.fl. menar att kvalitet på mjukvara har olika betydelser från olika perspektiv. För användaren är det hur mjukvaran utför sin uppgift. För utvecklaren handlar det om mjukvaran överensstämmer med specifikationerna. För själva produkten handlar det om struktur och design på programmet. Till sist från ett ekonomiskt perspektiv, betalar kunden en likvärdig summa som prestandan hos den levererade produkten? [6]

Källkod ska inte vara onödigt komplex, men hur kan man avgöra hur komplex koden är? Coupling och cohesion är välkända begrepp inom området och mäter kodens strukturella komplexitet [8]. Flera metoder har föreslagits. Cyclomatic complexity och Halstead Metrics är exempel på sådana.

Coupling beskriver kopplingen mellan klasser; en stark koppling gör klasser beroende av varandra, vilket försvårar enhetstestning som ska ske isolerat från andra moduler i programmet. Coupling ska hållas låg [10].

Cohesion innebär att de flesta metoder i en viss klass arbetar mot de flesta av klassens attribut (instansvariabler). En klass som går att bryta ner till flera klasser anses ha låg cohesion. Cohesion ska hållas hög [10].

I vissa fall kan det vara svårt att undvika låg cohesion eller hög coupling, men sådana moduler bör befinna sig i delar av programmet som inte ändras ofta och vara medvetet skapade på så sätt.

2 Metod

Detta kapitel redovisar hur arbetet gick till i olika steg och vilka metoder som användes för att besvara på frågeställningarna. Avsnitt 2.1 beskriver hur information kring tidigare studier samlades. Avsnitt 2.2 presenterar genomförandet av kodmätningen och vilka metriker som användes. Slutligen, i kapitel 2.3 redovisas hur data från kodmätningen behandlades och vilka gränsvärden som användes.

2.1 Insamling av information från tidigare studier

Litteratur samlades in från databaserna ACM Digital Library, IEEE Xplore, ScienceDirect och Google Scholar. Söksträngarna som användes var ”software AND measurement”, ”software AND metrics”, ”software AND metric threshold”, ”(software OR code) AND metric AND threshold”. Rapporter var av intresse baserat på följande kriterier: (1) titel, (2) abstrakt, (3) slutsater. En rapport som hade dragit slutsatser som ansågs var relavanta till detta arbete sparades och lästes vid ett senare tillfälle. När rapporterna lästes kunde fler relevant litteratur identifieras genom att undersöka referenserna i varje rapport.

2.2 Genomförande av kodmätning

För att utföra mätningar i arbetet utvecklades ett program som beräknar ut ett antal metriker utifrån Solutionfiler (.sln). En Solution innehåller information om ett eller flera .NET-projekt och varje projekt innehåller källkodsfiler. I systemet som analyserades fanns ca. 20 av dessa Solutionfiler. Enhetstester ignorerades i mätningen. Microsoft erbjuder ett API för .NET-kompilatorn Roslyn [11], vilket gör det möjligt att på ett relativt enkelt sätt manipulera källkoden programmatiskt. Med Roslyn som grund skapades metrikräknare som använder sig av symboler och syntaktiska enheter för klasser i källkoden för att räkna ut värdet på en metrik. För varje räknare skapades enhetstester för att validera att de fungerade som det var tänkt. Programmet är skrivet i C# därför att det stöds av Roslyn API:et. I den slutgiltiga versionen av programmet tog det ungefär 25 minuter att analysera hela systemet. Datorn som användes hade en Intel i7-2630QM (8 kärnor, 2.0GHz) och 12GB RAM.

Anledningen till att ett nytt program utvecklades för detta arbete var att existerande verktyg, som NDepend [12] och FxCop [13], inte täckte alla behov. Några av dessa behov var att verktyget var gratis och/eller open source. Existerande verktyg mäter vissa metriker på olika sätt, eftersom det inte alltid finns exakta definitioner för dem. Utvecklandet av ett eget program gjorde att det inte fanns någon tvekan för hur varje metrik räknas ut, vilket kanske inte vore fallet om ett existerande (icke-open source) verktyg hade använts.

Metrikerna som användes i detta arbete kommer från Chidamber och Kemerer, det så kallade C.K. Metrics suite [5]. Anledningen till att dessa valdes var att de förekommer ofta i litteraturen och det har utförts studier som visar att de är effektiva för att hitta fel i klasser [14] och att de används för att mäta underhållbarhet och återanvändning [6]. Dessutom föreslogs metrikerna med objektorienterad design i åtanke [5]. I Tabell 1 nedan listas samtliga metriker som användes i arbetet, sedan ges beskrivningar av vad de mäter och några fakta om hur de implementerades. Alla metriker mäter på klassnivån. Med ”klass” i detta sammanhang menas vanliga klasser, interface, enums o.s.v.

TABELL 1: METRIKER SOM ANVÄNDES I MÄTNINGEN

Namn	Förkortning
Weighted Methods per Class	WMC
Depth of Inheritance Tree	DIT
Number of Children	NOC
Coupling between object classes	CBO
Response for a Class	RFC
Lines of Code	LoC

WMC beskriver summan av komplexiteten hos en klass. För att beräkna komplexiteten hos en metod kan t.ex. cyclomatic complexity användas, sedan summeras resultaten från alla metoderna. Ju fler metoder en klass har kan alltså innebära att det blir svårare att underhålla klassen. Cyclomatic complexity-implementationen räknade följande syntaktisk enhet i en metod: if-sats, while-sats, for-sats, foreach-sats, continue-sats, goto-sats, catch-block, logiska AND, logiska OR, logiska NOT, case och default i switch-satser, villkorsoperatorer och null coalescing operatorer. För varje syntaktisk enhet programmet hittar inkrementeras McCabe-värdet. Denna implementation är inspirerad av NDepend [12].

DIT beskriver hur djup en arvshierarki är, från ”rot” till ”löv” i trädet. Den mäter hur många superklasser som kan påverka en given klass. En klass som har ett högt DIT, alltså befinner sig djupt ner i en arvshierarki, har sannolikt fler metoder än sina superklasser och är därmed mer komplex. Däremot kan djupa träd innebära att mer kod kan återanvändas via arv. När man mäter en design kan man med DIT undersöka om den är ”top heavy” eller ”bottom heavy”, d.v.s. om de flesta klasserna befinner sig långt uppe eller nere i hierarkin. En design som är ”top heavy” kan innebära att kodåteranvändning via arv inte utnyttjas, men en ”bottom heavy” kan betyda att designen är mer komplex. Ett högt DIT-värde kan göra klasser överlag kräva mer ansträngning att enhetstesta eftersom de potentiellt ärver fler metoder. Implementationen av DIT räknar antalet superklasser hos en typ och inkluderar alltid *System.Object* (alla klasser ärver av den i C#) för klasser, men för interface är DIT alltid 0.

NOC mäter hur många direkta subklasser en viss klass har. Högre värden kan visa att arvsmekanismen missbrukas och att det blir svårare ett testa. NOC-implementationen letar efter alla typer i en Solution och kollar om bastypen är samma som klassen i fråga. Endast klasser räknas, vilket innebär att för interface är NOC alltid 0. Ett alternativ vore att låta NOC för interface vara definierat som antalet klasser/interface som implementerar/ärver av det.

CBO räknar hur många klasser som är kopplade till en viss klass. Som det beskrevs tidigare ska coupling hållas låg. Kopplingen är från båda hållen; klasser som klassen i fråga använder sig av och klasser som använder klassen räknas till CBO. En hög CBO gör det svårare att testa och återanvända klassen. Programmet räknar ut CBO genom att leta efter samtliga typer i en Solution och kollar om typen i fråga förekommer någonstans. Räknaren kollar också vilka typer som klassen använder sig av. För varje unik typ som hittas inkrementeras CBO-värdet.

RFC mäter antalet metoder som kan potentiellt exekveras från ett metodanrop. Ett exempel är en metod som i sin tur anropar två andra metoder kommer ha RFC-värdet 3 då själva metoden också räknas. Detta innebär att RFC ≥ 1 så länge klassen har åtminstone en metod och det följer att en klass med RFC $= 0$ är oönskad för klasser. För interface räknar RFC alltså bara antalet metoder. Ett högt värde på RFC innebär att klassen blir mer komplex och det innebär att testande blir mer utmanande och kräver mer förståelse för klassen av programmeraren. Programmet räknar varje metodanrop och metoddeklaration, inklusive konstruktorn, för att räkna ut RFC-värdet.

LOC har beskrivits tidigare i rapporten. Den räknar helt enkelt antalet kodrader i en klass. Anledningen till att den användes var för att den är enkel att relatera till och ger ytterligare perspektiv på vad för slags klass det är som mäts. I implementationen räknas alla rader förutom de som är tomma.

2.3 Analys av kodmätning

Här beskrivs det hur resultaten behandlades efter att de hade tagits fram. Varje värde för respektive metrik jämfördes med två typer av gränsvärden. Den första typen är gränsvärden som hittades i litteraturen och är baserade på beprövad erfarenhet eller är framtagna genom att tillämpa en speciell metod. Den andra typen genererades baserat på resultaten av kodmätningen, s.k. relativa gränsvärden. Nedan presenteras valda gränsvärden från litteraturen och sedan presenteras olika metoder för att beräkna relativa gränsvärden.

Två källor användes för rekommenderade gränsvärden: Arar och Ayan [15] och verktyget NDepend [12]. En sammanfattning av gränsvärdena från dessa finns i Tabell 2. Studien från Arar var en återskapelse av en tidigare studie som räknade ut tröskelvärden för RFC, CBO och WMC. Till skillnad från originalet försökte författarna ta fram generella gränsvärden för de metrikerna och några till genom att analysera ett större dataset av Java-applikationer. I detta arbete anses gränsvärden för C# gälla för Java och tvärtom, eftersom båda är objektorienterade och har liknande syntax. Gränsvärdena delades upp i två kategorier i studien: ett tröskelvärde (THR) som förutsäger att det finns en eller flera felaktigheter hos en klass och ett värde som förutsäger tre eller flera felaktigheter (THR1 respektive THR3). Till exempel, för WMC är THR1 = 7, vilket innebär att det finns åtminstone ett fel och THR3 = 12 att det finns åtminstone tre fel. I detta arbete användes både THR1 och THR3, så i Tabell 2 representeras varje par av värden som ”THR1 - THR3” (förutom CBO som hade samma värde för båda). Det är värt att uppmärksamma att WMC räknades ut genom att räkna antalet metoder i en klass, vilket innebär att två varianter av WMC användes i detta arbete för att jämförelserna skulle bli meningsfulla. Den ena varianten räknar metoder (WMC_Simple) och den andra använder cyclomatic complexity (WMC_CC).

TABELL 2: GRÄNSVÄRDEN FRÅN OLIKA KÄLLOR.

Metrik	**Arar** [15] **(THR1-THR3)**	**NDepend** [12]
WMC	7 - 12*	-
DIT	-	6
NOC	-	-
CBO	8	-
RFC	20 - 30	-
LOC	114 - 171	20**

*Komplexiteten beräknades genom att räkna antalet metoder.

**Räknar ”logiska” kodrader istället för fysiska.

NDepend är ett kommersiellt verktyg som kan räkna ut olika metriker på .NET-applikationer. För vissa av metrikerna som stöds finns rekommenderade gränsvärden [12]. De rekommenderar en övre gräns på 6 för DIT men noterar att högre värden inte behöver vara dåliga därför att ibland ärver klasser av tredjepartsbibliotek som kan ha stora hierarkiträd. LoC beräknas genom att räkna antalet ”logiska” kodrader, d.v.s. inte på det sätt som rader förekommer i källkodsfilerna. Det är av den anledningen som Arars LoC föredras framför NDepend för LoC. Varken NDepend eller Arar gav någon rekommendation för NOC.

För att beräkna relativa gränsvärden användes tre metoder som hittades i litteraturen. Dessa tre metoder föreslogs av Alves, Ypma, Visser [16], Ferreira m.fl. [17] och Oliveira, Valente, Serebenik [18]. För att applicera metoderna användes verktyget TDTool [19]. En fjärde metod framtagen av Veado, Vale, Fernandes och Figueiredo [20] hittades också, men tidiga resultat visade att inkonsistenta värden producerades av TDTool. Vare sig detta beror på hur verktyget är skapat eller om metoden i sig inte är tillräckligt beprövad är oklart, men deras metod användes inte för detta arbete. I resten av denna rapport används Alves, Ferreira respektive Oliveira för att referera till metoderna som faktiskt användes. Anledningen till att dessa metoder användes var att olika system och projekt värderar olika kodstilar. Ett team Java-programmerare kanske utnyttjar arvmekanismen mer än interface, vilket skulle innebära att medelvärdet för NOC/DIT är högre jämfört med ett annat team som föredrar motsatsen. Varje gränsvärde för metriker måste därför anpassas efter behov. Ett alternativ är att teamen själva prövar sig fram för att hitta lämpliga värden. Metoderna Alves etc. beräknar rekommenderade gränsvärden som är relativa till ett eller flera projekt. I litteraturen förekommer det föreslagna gränsvärden för metriker [3] [15] [12] men de är inte menade för alla typer av projekt och system. Det är av den anledningen som både gränsvärden som föreslagits i litteraturen och gränsvärden som tagits fram av metoderna Alves etc. användes i analysen för detta arbete. Nedan sammanfattas de tre metoderna. För mer utförliga förklaringar och diskussioner kring dem hänvisas läsaren till det ursprungliga rapporterna för respektive metod.

Alves metod [16]. Metoden består av sex steg. I det första steget beräknas metriker från en eller flera källor. Varje källa representerar ett system som innehåller moduler (t.ex. klasser) och varje modul har ett värde för varje metrik samt ett viktat värde (LOC användes av Alves och även i detta arbete). I det andra steget beräknas förhållandet mellan det viktade värdet och metrikens värde. I det tredje steget grupperas varje förhållande med samma värde och varje grupp av moduler med samma metrikvärde representerar en viss del av system i procent. Det fjärde steget är en liknande process som slår ihop alla system. Femte steget sorterar varje grupp baserat på procentvärde i fallande ordning och sedan väljs det största metrikvärdet ut i varje grupp. I det sjätte och sista steget väljs ett procentvärde ut för att ta fram ett tröskelvärde. Ett exempel, tagen från Alves, är att för 90% av koden, har varje modul metrikvärdet 14 eller lägre. Fyra kategorier kan identifieras med metoden: låg risk (0-70%), medel risk (70-80%), hög risk (80-90%) och väldigt hög risk (över 90%). I ett sådant fall skulle en modul med metrikvärdet 14 falla på gränsen till väldigt hög risk, vilket är den ”värsta” kategorin.

Ferreiras metod [17]. Denna metod kan delas upp i fyra steg. Först beräknas metriker från en källa. I andra steget grupperas metrikerna. Sedan i tredje steget skapas grafer för varje metrik och i fjärde steget analyseras den och delas upp i tre kategorier: bra, vanlig och dålig sorterat på frekvens.

Oliveiras metod [18]. För att räkna ut tröskelvärden används en funktion *ComplianceRate* som returnerar två värden p och k för en given metrik M, där k är tröskelvärdet och p är ett procentvärde. Förhållandet $M \leq k$ bör gälla för p% av alla klasser. Författarna hävdar att alla olika metriker kan användas med deras metod.

3 Resultat

I detta kapitel redovisas resultaten från kodmätningen och resultaten från verktyget som beräknade relativa gränsvärden, vilket svarar på frågeställning 3: ”Vilka mätvärden erhålls med valda metriker applicerade på kodbasen?”.

3.1 Kodmätning

I Tabell 3 visas varje Solution i systemet och statistik över dem.

TABELL 3: STORLEKAR PÅ SYSTEMET. MED KLASSER MENAS ÄVEN INTERFACE, ENUMS OCH LIKNANDE.

Namn	Antal klasser	Antal kodrader
Monitor.DocumentServer.sln	18	802
Monitor.Web.sln	94	2478
Monitor.Backup.sln	108	3330
Monitor.Client.WinForms.sln	3511	119823
Monitor.Service.Contracts.sln	2174	71821
Monitor.LoadTestTool.sln	152	4087
Monitor.sln	2	25
Monitor.Client.Shared.sln	70	2298
Monitor.NET.sln	13768	752680
Monitor.Installer.sln	652	17702
Monitor.ConvertPreparer.sln	338	8063
TimeTransponderG5.sln	6	358
UnusedPhrasesDigger.sln	7	223
CopyNewForG5.sln	15	423
ChangeLogGenerator.sln	3	112
Monitor.RecipePreviewTool.sln	16	559
Monitor.RecipeCompiler.sln	10	389
Monitor.DatabaseConverter.sln	986	56424
Monitor.API.sln	169	7865
Monitor.Server.sln	105	3986
Monitor.Installer.Logic.sln	68	2628
MonitorG5.DatabaseConverter.sln	102	2185
Monitor.Updater.sln	28	444
Totalt	22402	1058705

Den största delen av systemet består av Monitor.NET.sln med över 50% av alla klasser och kodrader i systemet. Nedan i Tabell 4 visas medelvärden för varje metrik på varje Solution. I Tabell 5 visas medianen för varje Solution.

TABELL 4: MEDELVÄRDEN FÖR VARJE METRIK, AVRUNDAT TILL TRE DECIMALER.

Namn	DIT	RFC	WMC_CC	WMC_Simple	NOC	CBO	LoC
Monitor.RecipePreviewTool.sln	0,938	12,063	4,438	4,063	0,000	3,125	34,938
UnusedPhrasesDigger.sln	1,000	13,143	4,714	1,857	0,000	2,429	31,857
Monitor.RecipeCompiler.sln	0,900	15,700	3,900	2,200	0,000	3,000	38,900
Monitor.LoadTestTool.sln	1,000	7,000	2,000	1,000	0,000	5,000	17,000
Monitor.sln	3,000	1,000	0,000	0,000	0,000	0,000	12,500
TimeTransponderG5.sln	2,000	25,500	8,500	4,500	0,000	2,500	59,667
Monitor.DocumentServer.sln	1,833	11,333	5,500	2,556	0,000	2,889	44,556
Monitor.Updater.sln	1,120	5,600	2,320	1,080	0,000	4,520	16,680
Monitor.Server.sln	1,500	14,471	4,892	2,284	0,225	3,931	38,716
Monitor.Installer.Logic.sln	1,582	14,881	5,358	2,836	0,075	3,881	38,881
Monitor.Service.Contracts.sln	1,430	8,738	4,014	2,096	0,186	8,190	32,947
Monitor.ConvertPreparer.sln	2,577	9,296	2,553	1,825	0,633	5,216	23,855
Monitor.DatabaseConverter.sln	1,764	27,237	5,995	2,766	0,727	5,965	56,919
Monitor.API.sln	1,101	2,746	1,633	0,450	0,089	53,740	46,538
Monitor.Client.Shared.sln	0,786	11,929	5,857	4,057	0,057	4,514	32,829
Monitor.Installer.sln	2,207	8,876	3,537	1,859	0,419	7,129	27,150
CopyNewForG5.sln	1,867	11,000	3,000	1,600	0,067	5,267	28,200
ChangeLogGenerator.sln	3,000	16,333	4,000	1,000	0,000	3,000	37,333
Monitor.Client.WinForms.sln	2,225	10,659	4,869	2,702	0,213	7,780	34,159
Monitor.NET.sln	2,174	22,572	6,879	3,169	0,417	12,934	54,706
MonitorG5.DatabaseConverter.sln	1,863	9,951	1,833	0,990	0,000	4,069	21,422
Monitor.Web.sln	1,323	9,398	3,290	1,806	0,151	4,452	26,226
Monitor.Backup.sln	1,991	7,611	2,991	1,361	0,019	2,102	30,833

TABELL 5: MEDIANER FÖR VARJE METRIK.

Namn	DIT	RFC	WMC_CC	WMC_Simple	NOC	CBO	LoC
Monitor.DocumentServer.sln	1	5	1,5	1	0	2,5	17,5
Monitor.Web.sln	1	4	2	1	0	4	15
Monitor.Backup.sln	2	1	0	0	0	0	9,5
Monitor.Client.WinForms.sln	1	4	2	1	0	5	14
Monitor.Service.Contracts.sln	1	2	1	1	0	3	12
Monitor.LoadTestTool.sln	1	7	2	1	0	5	17
Monitor.sln	3	1	0	0	0	0	12,5
Monitor.Client.Shared.sln	1	3	2	2	0	3	12,5
Monitor.NET.sln	1	5	2	1	0	6	18
Monitor.Installer.sln	2	3	1	1	0	5	11
Monitor.ConvertPreparer.sln	2	4	1	1	0	3	11
TimeTransponderG5.sln	1	11	3	2,5	0	1,5	22
UnusedPhrasesDigger.sln	1	10	2	1	0	1	26
CopyNewForG5.sln	1	9	1	1	0	4	20
ChangeLogGenerator.sln	1	1	0	0	0	1	6
Monitor.RecipePreviewTool.sln	1	4	2	2	0	2	20
Monitor.RecipeCompiler.sln	1	6	1,5	1,5	0	2	20,5
Monitor.DatabaseConverter.sln	2	8	2	2	0	2	19
Monitor.API.sln	1	1	0	0	0	6	19
Monitor.Server.sln	1	6	2	1	0	3	18,5
Monitor.Installer.Logic.sln	1	7	3	2	0	3	18
MonitorG5.DatabaseConverter.sln	2	1	0	0	0	0	9
Monitor.Updater.sln	1	3	1	1	0	3	9

3.2 Relativa gränsvärden

Tabeller 6, 7 och 8 visar resultaten för Alves, Ferreiras respektive Oliveiras metoder.

TABELL 6: RESULTAT FRÅN ALVES METOD.

Metrik	0-70% Låg	70-80% Medel	80-90% Hög	>90% Väldigt hög
DIT	0	2	3	6
RFC	0	43	58	97
WMC_CC	0	13	22	40
WMC_Simple	0	6	10	16
NOC	0	0	0	0
CBO	0	7	10	21
LoC	1	128	195	303

TABELL 7: RESULTAT FERREIRAS METOD.

Metrik	<70% Bra	70-80% Vanlig	>80% Dålig
DIT	<2	2-3	>3
RFC	<11	11-19	>19
WMC_CC	<4	4-6	>6
WMC_Simple	<2	2-4	>4
NOC	0	0	>0
CBO	<9	9-12	>12
LoC	<34	34-54	>54

TABELL 8: RESULTAT FRÅN OLIVEIRAS METOD. N% AV KODEN SKA INTE VARA STÖRRE ÄN K.

Metrik	Procent (%)	k-värde (Gränsvärde)
DIT	75	3
RFC	65	16
WMC_CC	80	11
WMC_Simple	80	5
NOC	90	0
CBO	75	8
LoC	80	76

4 Analys

Detta kapitel analyserar resultaten och ämnar att svara på frågeställningar 1, 2 och 4. Först analyseras de uträknade värdena från kodmätningen. Sedan besvaras frågeställning 1 då tillräckligt med information har samlats in. Sedan kan frågeställning 2 besvaras. Till sist jämförs och analyseras resultaten från kodmätningen med gränsvärden från föregående kapitel, vilket besvarar frågeställning 4.

4.1 Kodmätning

Genom att undersöka Tabell 4 och 5 ser man att NOC-värdet (*Number of Children)* generellt är under 1 och DIT-värdet (*Depth of Inheritance Tree*) är 1 eller 2, vilket skulle kunna tyda på att arv inte används mycket. För att dra nytta av polymorfism används istället interface, vilket i mätningen alltid har NOC = 0. De klasser som har hög NOC brukar vara abstrakta, vilket innebär att de troligen är designade att ärvas av.

Samtliga medianer är lägre än medelvärdet för alla metrikerna, vilket innebär att det finns klasser som har mycket höga värden jämfört med de andra. Genom att undersöka resultaten från Monitor.NET.sln, den största delen av systemet, kunde man se att fyra klasser hade RFC-värden (*Response for a class*) på över 1000 och tre av dem befann sig inom samma projekt (Monitor.Modules.csproj). När klassen med störst värde, som också är den största inom hela dess Solution med LoC på över 3000, undersöks närmare så verkar den vara en del av ett ramverk som hanterar GUI-relaterade operationer. Förutom de fyra med över 1000 RFC, så fanns det några hundratals klasser med RFC över 100, vilket kan förklara medelvärdet 23 jämfört med medianen 5. Klasser med RFC- och DIT-värden lika med 0 skulle kunna tyda på ett interface med inga metoder, men efter nogrannare undersökningar är det troligt att de flesta av dem är klasser eller interface med endast *properties*, en egenskap hos C#, vilket implementationen ej räknar som metod.

En klass DepencyRegistrations visade en hög CBO (*Coupling between object classes*) på 1844 jämfört med resten (nästhögsta hade 1261). Som namnet tyder på så registrerar klassen beroenden, vilket innebär att det inte är konstigt att den har en hög koppling till andra klasser.

Det finns klasser där CBO är lika med noll, vilket innebär att den inte använder eller används av någon klass. I vanliga fall skulle klassen sakna värde, men många av dessa klasser innehåller *Extension*-metoder, vilket är en funktion hos C# för att utöka klasser med nya metoder och som det skapade programmet tilldelar CBO-värdet noll om den inte har en koppling till någon klass deklarerad i dess Solution. Tre Solutions har medianen noll för CBO (Monitor.Backup.sln, MonitorG5.DatabaseConverter.sln och Monitor.sln) vilket beror på att de innehåller många deklarationer av *Attributes*. Monitor.sln verkar vara ”startpunkten” för hela systemet och i miljön som programmet exekverades på uppstod det problem att öppna den, vilket skulle kunna förklara varför några metriker för den rapporterades felaktigt.

Klassen AccountsPayableApp.cs har över hundra metoder, ca 2200 LoC och innehåller många nästade if- och switch-satser, vilket förklarar varför den har högst WMC_CC (*Weighted Methods per Class*, beräknat med cyclomatic complexity) i Monitor.NET.sln. Resten av klasserna med liknande WMC_CC har liknande egenskaper.

4.2 Jämförelse med framtagna gränsvärden

I Tabellerna 6, 7 och 8 ser man att DIT-gränsvärdena är väldigt lika varandra; värden större än 3 är höga men 1-2 är normala. Gränsen på DIT från NDepend (Tabell 2) är 6 som är större än dessa, men däremot stämmer det värdet överrens med ”Väldigt hög” från Alves metod (Tabell 6). RFC-värdena liknar varandra med undantag från Tabell 5, där 43 är medel jämfört med resten som föreslår att värden större än 15-20 anses vara högt.

För WMC som beräknades med McCabe-värden för varje metod var gränsvärdena mellan ca 10-20, vilket är intressant för McCabe använde 10 som gränsvärde för en enda metod [3]. Den enkla versionen av WMC där antalet metoder räknades hade liknande värden för alla förutom Ferreiras metod, som var lägre. NOC är intressant för samtliga resultat gav ett gränsvärde på 0, vilket tyder på att allt arv anses vara högt. Eftersom NOC är låg över hela systemet kan slutsatsen dras att NOC inte är väldigt intressant att mäta i nuläget. Självklart kan detta ändras i framtiden om t.ex. kodstilen ändras.

Svar till frågeställning 1, ”Vilka metriker är relevanta för Monitor?”, blir då DIT, RFC, WMC, CBO och LoC eftersom alla kan användas till att upptäcka komplexa delar i systemet. Självklart är det möjligt att det finns fler intressanta metriker, men som det var skrivet i kapitlet Avgränsningar är det inte möjligt att undersöka alla metriker som finns i detta arbete p.g.a. tidsbrist.

Frågeställning 2, “Vad anses vara bra värden för utvalda metriker enligt tidigare studier?”, går inte att ge ett definitivt svar på, då det inte fanns några överenskomna, definitiva värden att följa. De relativa gränsvärden från Tabell 6, 7 och 8 samt vad andra har kommit fram till från Tabell 2 presenteras som förslag till möjliga gränsvärden till den studerade kodbasen.

När man observerar medianer och medelvärden och jämför med framtagna gränsvärden ser man att DIT aldrig går över gränserna. Som noterat tidigare har det flesta klasserna NOC-värdet 0, vilket stämmer överens med DIT då metrikerna har ett samband – DIT mäter höjden på arvshierarkin och NOC mäter bredden.

En Solution, Monitor.DatabaseConverter.sln, hade ett dåligt medelvärde för RFC enligt alla gränser förutom Alves (Tabell 6). Medianen var 8 och medelvärdet 27, vilket innebär att vissa klasser har ovanligt höga värden för metriken. Likt Monitor.NET.sln finns det ett fåtal klasser som har RFC-värden över 1000 och ett flertal över 100, vilket är mycket höga värden enligt gränserna. När klasserna med högst värden studerades upptäcktes det att klasserna innehåller väldigt mycket objektskapande och ett fåtal nästade villkorssatser, vilket förklarar varför programmet rapporterade sådana höga RFC-värden. TimeTransponderG5.sln och UnusedPhrasesDigger.sln har största medianerna för RFC och medelvärden på 26 respektive 13. TimeTransponderG5 har en GUI-klass som hanterar databasanrop som höjde medelvärdet. Varje klass i UnusedPhrasesDigger förutom en har RFC mellan 10 och 24 vilket förklarar dess höga median.

WMC_CC, beräknat med cyclomatic complexity, har maxvärden 3 och 7 för medianen respektive medelvärdet vilket innebär att den ligger under alla gränser förutom Ferreira i Tabell 7 (Arar har 7 som gränsvärde, men medelvärdet 7 är avrundat upp från ~6.8). WMC_Simple, beräknat genom att räkna antal metoder, har lägre eller likadana värden som WMC_CC, som förväntat. Två Solutions har medelvärdet 4, vilket ligger på gränsen att gå över till ”Dålig” för Ferreira (Tabell 7).

Gränserna för CBO var 8, 10 och 12 och medianerna för varje Solution är under dessa gränser, men medelvärdena av två Solutions var större. Monitor.NET.sln hade ett medelvärde på ungefär 13 och Monitor.API.sln hade värdet 54, vilket är mycket högre än resten. CBO för Monitor.NET.sln nämndes i förra kapitlet; den har en klass som registerar beroenden vilket leder till ett väldigt högt CBO, men som man kan se i Tabell 5 är medianen 6, vilket inte överskrider någon gräns. Monitor.API.sln har också flera klasser med CBO-värden över 100, men eftersom den har färre klasser totalt blir medelvärdet mycket högre. Även Monitor.API.sln har en median på 6, så generellt ligger den inte i riskzonen.

Resultaten för LoC visar att generellt så är antalet kodrader få relativt till framtagna gränser. Endast några medelvärden anses vara dåliga enligt Ferreiras metod (Tabell 7), men LoC-gränsvärdena för Ferreira var låga relativt till de andra (54 jämfört med 76, 114 och 195).

Sammanfattningsvis hade de flesta klasserna generellt lägre metrikvärden än de framtagna gränsvärdena. Det fanns klasser som hade mycket högre värde än resten, vilket kan ses när medianen och medelvärdet jämfördes. Nu har frågeställning 4, ”Hur förhåller sig resultaten till tidigare forskning och erfarenheter?”, besvarats. Resultaten förhåller sig alltså bra.

5 Diskussion

Gränsvärdena som togs fram ur litteraturen och som genererades från existerande metoder med hjälp av ett verktyg är inte menade att vara absoluta värden som måste följas i alla situationer. Efter att ha läst en del rapporter från området bildades en uppfattning om att det inte finns några sådana värden, de måste anpassas för varje projekt eller organisation. Formella metoder för att ta fram dessa värden har föreslagits och några av dem har använts i detta arbete. Som tidigare nämnts i denna rapport är ett alternativ att man prövar sig fram till gränsvärden som passar. Det var inte förrän arbetet hade kommit igång som detta insågs, vilket gör att frågeställningen inte kan få ett definitivt svar. Därför föreslås ett antal gränsvärden som en rekommendation till Monitor och kanske till andra företag med liknande metodiker eftersom de framtagna värdena liknade de som togs från litteraturen. Dessa kan användas som en startpunkt för att få en känsla för vad siffrorna som metrikerna producerar betyder. Man kan också fråga sig om dessa metriker är nödvändiga om kodgranskning redan används för att hålla en konsistent stil och hög kvalitet på koden. Troligen kan metriker komplementera kodgranskningar om de används för att uppmärksamma områden som ligger i ”riskzonen”. Det är viktigt att tänka på att om man börjar använda metriker för kod måste man alltid använda samma verktyg för att räkna ut dem (eller vara helt säker på att de alltid räknas ut på exakt samma sätt). Ett tydligt exempel är WMC: det är uppenbart att WMC som räknas ut genom att räkna metoderna i en klass är annorlunda än WMC som summerar alla McCabe-värden. Ett gränsvärde för det ena gäller självklart inte för det andra. Det finns fler, mycket mindre detaljer hos andra metriker som kan ge olika resultat för samma kodstycke, vilket betonar hur viktigt det är att vara konsistent. Det är viktigt att inte förlita sig för mycket på metriker. När en programmerare hittar en elegant lösning på ett problem ska den lösningen inte nekas endast för att den producerade för höga mätvärden. Programmerare ska inte heller behöva känna sig pressade att skriva kod som är ”bra” enligt metriker.

I varje Solution fanns det klasser som hade mycket höga värden. Klasserna med höga metrikvärden behöver nödvändigtvis inte vara ”dåliga”, så länge klasserna inte ändras ofta är de nödvändigtvis inte en källa för buggar. Det är upp till experter eller andra individer med tillräcklig erfarenhet med systemet att avgöra om klasserna har varit problematiska tidigare. Vad metrikerna gör är att uppmärksamma delar som möjligtvis är för komplexa. Samtidigt är det mycket möjligt att vissa klasser som faktiskt är buggbenägna inte upptäcktes under detta arbete. Metrikerna som presenterats som intressanta behöver inte vara de enda; fler metriker kanske skulle kunna hjälpa med att avslöja andra typer av klasser som är svåra att underhålla.

Eftersom programmet som skapades räknade med interface, enums och andra typer som inte är klasser, kan vissa metriker ha fått lägre värden än om endast klasser användes. Till exempel, WMC som räknas ut med cyclomatic complexity kommer alltid ha lika stort eller mindre värde för ett interface jämfört med en klass som implementerar det. När det finns många interface kommer medelvärdet och medianen för WMC möjligtvis att bli mindre än om endast klasser räknas med i uträkningen. Ett alternativ till detta vore att bara använda klasser för WMC och andra metriker som DIT och RFC (som har liknande problem). Samtidigt är det intressant att mäta WMC för interface, för vad man får för värde är antal metoder som deklarerats i interfacet, vilket kan avslöja interface som är för komplexa. Valet att räkna *Properties* som metod eller inte påverkar också metriker som WMC. I språk som Java skulle en property vara en getter- och/eller settermetod som används för att hämta eller mutera egenskaper hos ett objekt, men som inte har någon komplex logik vilket leder till att metoderna inte gör att det blir svårare att underhålla koden. Properties är alltså inte menade att vara komplexa metoder, så att räkna dem till den totala komplexiteten i en klass kan också vara missledande.

I utvecklandet av programmet som utförde mätningen gjordes det försök till att följa goda designprinciper. Funktionaliteten prioriterades så att tiden som fanns till vara kunde spenderas till viktigare uppgifter. Det största förbättringsområdet i programmet är troligen den del som hanterar exportering av resultat. Det uppstod problem när CBO skulle beräknas. När CBO för en given klass skulle räknas ut, lopades klasserna i dess Solution och för varje klass gjordes ytterligare en loop för att se om den känner till givna klassen. Detta ledde till att programmet tog lång tid att exekvera på kodbasen (åtminstone två timmar). En optimering gjordes där varje klass och tillhörande typer den använder sparades i minnet när programmet startades. Sedan när CBO skulle räknas ut för en klass, kunde det sparade klasserna användas för att göra en enda loop. Grovt uppskattat, gick tidskomplexiteten från $O(n^2)$ till $O(n)$ och i slutändan tog det endast 25 minuter att utföra mätningen. Med hjälp av enhetstester kunde det bekräftas att funktionaliteten var den samma.

Arbetet kunde haft en annan inriktning än att analysera och jämföra metrikerna med jämförda gränsvärden. Det fanns tankar att analysera metriker i olika delar av systemet och försöka hitta korrelationer med metrikerna och rapporterad buggighet hos olika moduler. För att utföra en sådan analys i denna typ av arbete krävs det att det finns historisk data om buggar. För varje bugg måste det vara tydligt vilken del av systemet den tillhör, även om den är generell. Tyvärr fanns det ingen sådan data tillgänglig hos Monitor. En sådan analys skulle vara mycket intressant att utföra för alla parter.

6 Slutsatser

Detta arbete har svarat på följande frågeställningar:

1. Vilka metriker är relevanta för Monitor?
2. Vad anses vara bra värden för utvalda metriker enligt tidigare studier?
3. Vilka mätvärden erhålls med valda metriker applicerade på kodbasen?
4. Hur förhåller sig resultaten till tidigare forskning och erfarenheter?

Svaret till den första frågeställningen var metrikerna *Depth of Inheritance Tree*, *Response for a Class*, *Weighted Methods per Class*, *Coupling between object classes* och *Lines of Code*. Svaret till den andra frågeställningen hittas i Tabellerna 6, 7 och 8. I kapitel 3 redovisas svaret till den tredje frågeställningen. Svaret på den fjärde och sista frågeställningen var att mätvärdena för metrikerna på Monitors kodbas förhåller sig bra.

Kodmätningen inkluderade klasser och interface. Detta innebar att medianer och medelvärden av mätvärdena för vissa metriker var större eller mindre än om endast klasser skulle ha använts.

Referenser

[1] ”MONITOR Affärssystem - Monitor ERP System AB,” Monitor ERP System AB, [Online]. Available: http://www.monitor.se/. [Använd 3 April 2017].

[2] Microsoft, ”Team Foundation Server,” 2017. [Online]. Available: https://msdn.microsoft.com/en-us/library/ms181238(v=vs.90).aspx. [Använd 9 May 2017].

[3] T. J. McCabe, ”A Complexity Measure,” *IEEE Transactions on Software Engineering,* vol. 2, nr 4, 1976.

[4] IBM, ”IBM Knowledge Center - Halstead Metrics,” [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SSSHUF_8.0.0/com.ibm.rational.testrt.studio.doc/topics/csmhalstead.htm. [Använd 25 April 2017].

[5] S. R. Chidamber och C. F. Kemerer, ”A Metrics Suite for Object Oriented Design,” *IEEE Transactions on Software Engineering,* vol. 20, nr 6, pp. 476-493, 1994.

[6] E. M. Arvanitou, A. Ampatzoglou, A. Chatzigeoriou, M. Galster och P. Avgeriou, ”A mapping study on design-time quality attributes and metrics,” *The Journal of Systems and Software,* vol. 127, pp. 52-77, 2017.

[7] N. E. Fenton och M. Neil, ”Software metrics: successes, failures and new directions,” *The Journal of Systems and Software ,* vol. 47, pp. 149-157, 1999.

[8] M. Sarker, ”An overview of Object Oriented Design Metrics,” Umeå University, 2005.

[9] M. Fowler, ”CannotMeasureProductivity,” 29 August 2003. [Online]. Available: https://martinfowler.com/bliki/CannotMeasureProductivity.html. [Använd 19 April 2017].

[10] R. S. Pressman och B. R. Maxim, Software Engineering A Practitioner's Approach, New York: McGraw-Hill Education, 2015.

[11] GitHub, Inc., ”GitHub - dotnet/roslyn: The .NET Compiler Platform ("Roslyn") provides open-source C# and Visual Basic Compilers with rich code analysis APIs.,” 2017. [Online]. Available: https://github.com/dotnet/roslyn. [Använd May 17 2017].

[12] Zen Program Ltd, ”NDepend Code Metrics Definitions,” 2017. [Online]. Available: http://www.ndepend.com/docs/code-metrics. [Använd 21 May 2017].

[13] Microsoft, ”FxCop,” 2017. [Online]. Available: https://msdn.microsoft.com/en-us/library/bb429476(v=vs.80).aspx. [Använd 23 May 2017].

[14] V. R. Basili och L. M. W. Briand, ”A Validation of Object-Oriented Design Metrics as Quality Indicators,” *IEEE Transactions on Software Engineering,* vol. 22, pp. 751-761, 1996.

[15] Ö. F. Arar och K. Ayan, ”Deriving thresholds of software metrics to predict faults on open source software: Replicated case studies,” *Expert Systems With Applications,* vol. 61, pp. 106-121, 2016.

[16] T. L. Alves, C. Ypma och J. Visser, ”Deriving Metric Thresholds from Benchmark Data,” *26th IEEE International Conference on Software Maintenance,* 2010.

[17] K. A. Ferreira, M. A. Bigonha, R. S. Bigonha och L. F. H. C. Mendes, ”Identifying thresholds for object-oriented software metrics,” *The Journal of Systems and Software,* 2011.

[18] P. Oliveira, F. P. Lima, M. T. Valente och A. Serebenik, ”RTTool: A Tool for Extracting Relative Thresholds for Source Code Metrics,” *International Conference on Software Maintenance and Evolution,* 2014.

[19] L. Veado, G. Vale, E. Fernandes och E. Figueiredo, ”TDTool: Threshold Derivation Tool,” *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering,* pp. 1-5, 2016.

[20] G. Vale, D. Albuquerque, E. Figueiredo och A. Garcia, ”Defining Metric Thresholds for Software Product Lines: A Comparative Study,” *Proceedings of the 19th International Conference on Software Product Line,* pp. 176-185, 2015.