Utvärdering av refaktorisering

Emil Ståhl
Skolan för elektroteknik och datavetenskap (EECS)
Kungliga Tekniska Högskolan (KTH)
Sverige
emil.stah@kth.se

Abstract

Software development is one of the areas of technology that is currently experiencing the fastest rate of progress. A systems code base can easily consist of hundreds of thousands lines of code. With continuous development of these systems, it can result in problems regarding maintainability, comprehension and effectiveness. The aim of this study is to investigate how refactoring as a software practice affects the performance, lines of code and cyclomatic complexity of a code base. This study was conducted during a software project during four weeks at the Royal Institute of Technology in Stockholm. The results of this study show that refactoring a code base results in no noteworthy improvement in the performance of the code. However, the study shows a consistent reduction of the amount of code in the code base. Regarding cyclomatic complexity the results show a small improvement after the completed refactoring.

Keywords: refactoring, software practice, cyclomatic complexity, maintainability, intelligibility

Innehållsförteckning

1. Inledning

Systemutveckling och programmering i synnerhet är en av de delar som idag utvecklas snabbast inom det tekniska området. En ordinär mobilapplikation består normalt av över 40 000 rader kod. Till följd av den stora mängd kod som ett mjukvaruprojekt kräver tar det inte lång tid innan källkoden blir ohanterlig på grund av osammanhängande kod i kombination med stora filer. Det uppskattas att 75% av en utvecklares tid spenderas på felsökning vilket för den genomsnittlige uppgår till 1500 timmar per år. Detta är ingenting som kommer att minska när komplexiteten av vardagliga system blir allt högre i takt med teknikutvecklingen.

För att komma till rätta med problemen och försäkra sig om att systemets källkod är driftsäker, effektiv och av allmän hög kvalitet är det fördelaktigt att använda sig av refaktorisering. Refaktorisering strävar efter att förbättra kvaliteten såväl som underhållbarheten av systemets källkod. Refaktorisering kan syfta till en rad olika tillvägagångssätt, däribland omstrukturering av kod, namnbyte av variabler och funktioner samt att extrahera kodblock. Denna åtgärd är dock både tids- och resurskrävande.

Den här rapporten kommer att analysera huruvida refaktorisering bidrar till minskad komplexitet och ökad underhållbarhet och prestanda vid exekvering. Undersökningen baseras på ett mjukvaruprojekt vid Kungliga Tekniska Högskolan i Stockholm och utförs av studenter i årskurs 2. Syftet är att utvärdera resultatet av refaktoriseringen och föreslå förbättringar utifrån den metod som använts.

Rapporten kommer att ha följande struktur som består av sex stycken delar: I avsnitt 2. *Bakgrund* beskrivs mjukvaruprojektet och delger även kontext för kommande delar. Avsnitt 3. *Metod* kommer att redovisa den forskningsmetod som använts för att klassificera projektets kod utifrån tre olika kriterium. Vidare i 4. *Resultat* presenteras forskningsresultaten i dess ursprungliga form för att sedan analyseras i avsnitt 5. *Diskussion.* Slutligen i 6. *Slutsats* fastslås slutsatser samt förslag på framtida forskning.

2. Bakgrund

Detta avsnitt ger en första teoretisk bakgrund till området som forskningen grundar sig på. Avsnitt 2.1 börjar med att beskriva det projekt som forskningen baserats på. Sedan följer en beskrivning av traditionell- och agil systemutveckling. Avslutningsvis redogörs för refaktoriseringens metoder.

2.1 Presentation av projekt

Det mjukvaruprojekt som denna forskning grundar sig på ägde rum under fyra veckor. Projektgruppen bestod av sju stycken andraårsstudenter på programmet Informationsteknik vid Kungliga Tekniska Högskolan, varav en är författare till denna rapport. Målet med mjukvaruprojektet var att utveckla en webbapplikation avsedd för att underlätta hanteringen av personliga önskningar till diverse högtider utifrån två faktorer:

1. Att bevara huvudpersonens omedveten angående vilken person som införskaffat vad från en ursprunglig lista.

2. Möjligheten till att göra anspråk på specifika ackusativobjekt i en lista utan huvudpersonens vetskap för att undvika dilemmat med att fler än en individ införskaffar identiska föremål riktade till personen i fråga.

Projektets huvudsakliga arbete bestod av att integrera ett stort antal olika moduler och komponenter, som till majoriteten behövde byggas från grunden. Ramverket som användes för att konstruera det grafiska webbgränssnittet var React.js. Det inre systemet skapades med hjälp av Firebase som tillhandahåller en realtidsdatabas som tjänst för externa utvecklare. Tjänsten erbjuder utvecklare ett applikationsprogrammeringsgränssnitt som möjliggör att applikationens data kan synkroniseras mellan klienter och lagras på Firebases servrar.

Arbetssättet som användes under projektet namnges som Scrum vilket beskrivs som ett ramverk för att utveckla och underhålla komplexa produkter i större projekt för att uppnå optimal effektivitet. Scrum är ett agilt sätt att fördela arbetsuppgifter i tiden med bibehållet fokus på levererad affärsnytta gentemot de mål som ska uppnås. De fyra veckorna, som projektet pågick, delades upp i fyra stycken sprintar, där varje sprints längd uppgick till en vecka (fem arbetsdagar). I början av vardera sprint hölls en sådan kallad sprintplanering där hela projektgruppen närvarade för att

bestämma den kommande veckans arbetsuppgifter, mål och tillvägagångssätt samt att estimera hur lång tid vardera uppgift bör ta. I slutet av vardera sprint hölls på samma sätt en retrospektiv för att utvärdera, analysera och finna förbättringsområden från den kommande veckan för att förbättra gruppens arbete för nästkommande arbete. Dessutom startade vardera arbetsdag med en ”Daglig Scrum” som var avsedd för att sammanfatta föregående dags arbete likväl som att bestämma vardera utvecklares uppgift för dagen.

2.2 Traditionell- och agil systemutveckling

Inom dagens mjukvaruutveckling finns två huvudsakliga metoder att tillämpa för att organisera arbetet i ett mjukvaruprojekt, dessa består av den traditionella metoden (vattenfallsmodellen) och den agila metoden.[1]

2.2.1 Vattenfallsmodellen

Vattenfallsmetoden är ett sätt att arbeta i ett linjärt och sekventiellt flöde. Inom mjukvaruutveckling är denna metod en av de minst iterativa och flexibla metoderna eftersom arbetet fortskrider i endast en riktning. Det är också detta som gett namn åt modellen där man liknar arbetet vid ett vattenfall som flödar nedåt i en bestämd riktning. Enligt den originella beskrivningen beskrivs arbetets olika faser som följer:

1. System- och mjukvarukrav fastslås
2. Analys i form av modeller, scheman och förhållningspunkter
3. Design av systemets arkitektur
4. Utveckling och integration av mjukvaran
5. Provdrift av systemet och allmän felsökning
6. Installation, distribution, migration och underhåll av det färdiga systemet

Modellen utgår från att flytt till nästa fas endast godkänns om den nuvarande fasens krav har uppnåtts på ett tillfredsställande sätt.[2]

---

1 Lotz, M. Waterfall vs. Agile: Which Methodology is Right for Your Project? 2013.

2 Bassil, Youssef. A Simulation Model for the Waterfall Software Development Life Cycle. Lebanese Association for Computational Sciences. 2012.

2.2.2 Agil systemutveckling

Agil utveckling är, tillskillnad från vattenfallsmetoden, ett arbetssätt där projektet fortskrider i ett ickelinjärt flöde där krav och lösningar dynamiskt utvecklas. Metoden klassas som en så kallad iterativ modell i det avseende att projektgruppen upprepande gånger genomgår de faser som beskrivits i föregående avsnitt 2.2.1. Metoden förespråkar adaptiv planering, tidig leverans och kontinuerlig förbättring och är speciellt förmånlig för hastig handlingskraft gentemot förändringar i planeringen.[3]

2.3 Programvaruutvecklingspraxis

På samma sätt som ett större mjukvaruprojekt tillämpar en viss modell för hur arbetet skall struktureras är det även vanligt att använda sig av mindre metoder som fokuserar till att förbättra en viss del inom projektet. Några av dessa visas i bilden nedan.

Pair programming	Estimation
Test-first development	Planning
Iterative development	Risk management
Retrospectives	Release planning
Informative workspaces	Iterative acceptance testing
User Stories	Refactoring
Story points	Communication (oral and written)

Figur 2.1. Exempel på programvaruutvecklingspraxis som beskrivet av Mira Kajko-Mattson[4]

Programvaruutvecklingspraxis kan enligt Capers Jones sägas bestå av en samling koncept, principer, metoder och verktyg som utvecklare använder i sitt arbete. Vidare förser det projektgruppen med teknisk hantering om hur specifika uppgifter ska utföras.[5]

3 Dingsøyr, Torgeir. et al. A decade of agile methodologies: Towards explaining agile software development. Juni 2012.

4 Kajko-Mattson, Mira; docent vid skolan för elektroteknik och datavetenskap, Kungliga Tekniska högskolan. Föreläsning 2019-03-18.

5 Jones, Caspers. Software Engineering Best Practices. 2010.

Denna rapport kommer specifikt att fokusera på refaktorisering som programvaruutvecklingspraxis.[6]

2.4 Refaktorisering

Refaktorisering som programvaruutvecklingspraxis syftar till att omstrukturera existerande programkod utan att ändra det externa beteendet. Refaktorisering är i första hand avsett för att förbättra icke-funktionella attribut av mjukvaran. Eventuella fördelar inkluderar förbättrad läsbarhet (förståelse) och reducerad komplexitet av kodblock. Detta bidrar i sin tur till mer fördelaktig underhållbarhet av källkod samt underlättar för en mer expressiv intern arkitektur och objektiv modell som gör det enklare att komplettera systemet med nya moduler och funktioner.[7] Allmänt kan refaktorisering vara då utvecklaren praktiserar något av följande:

Tekniker för att omstrukturera kod till mer logiska delar

- Bryta ut kodblock till självständiga komponenter som möjliggör återanvändning av väldefinierade gränssnitt
- Extrahera klasser, genom att flytta ett kodblock från en klass till en ny klass
- Extrahera metoder, genom att omstrukturera en stor metod till ett flertal mindre metoder

Tekniker som möjliggör högre abstraktion

- Skapa mer generella typer som möjliggör för koddelning
- Ersätta villkorlig programmering med polymorfism

Tekniker som förbättrar relevans

- Namnbyte av klasser, metoder och variabler som bättre representerar dess syfte
- Flytta upp kod: Flytt av kod till en superklass
- Flytta ned kod: Flytt av kod till en subklass

I mjukvaruprojekt existerar dessa tekniker parallellt och används i korrelation med varandra.[8]

---

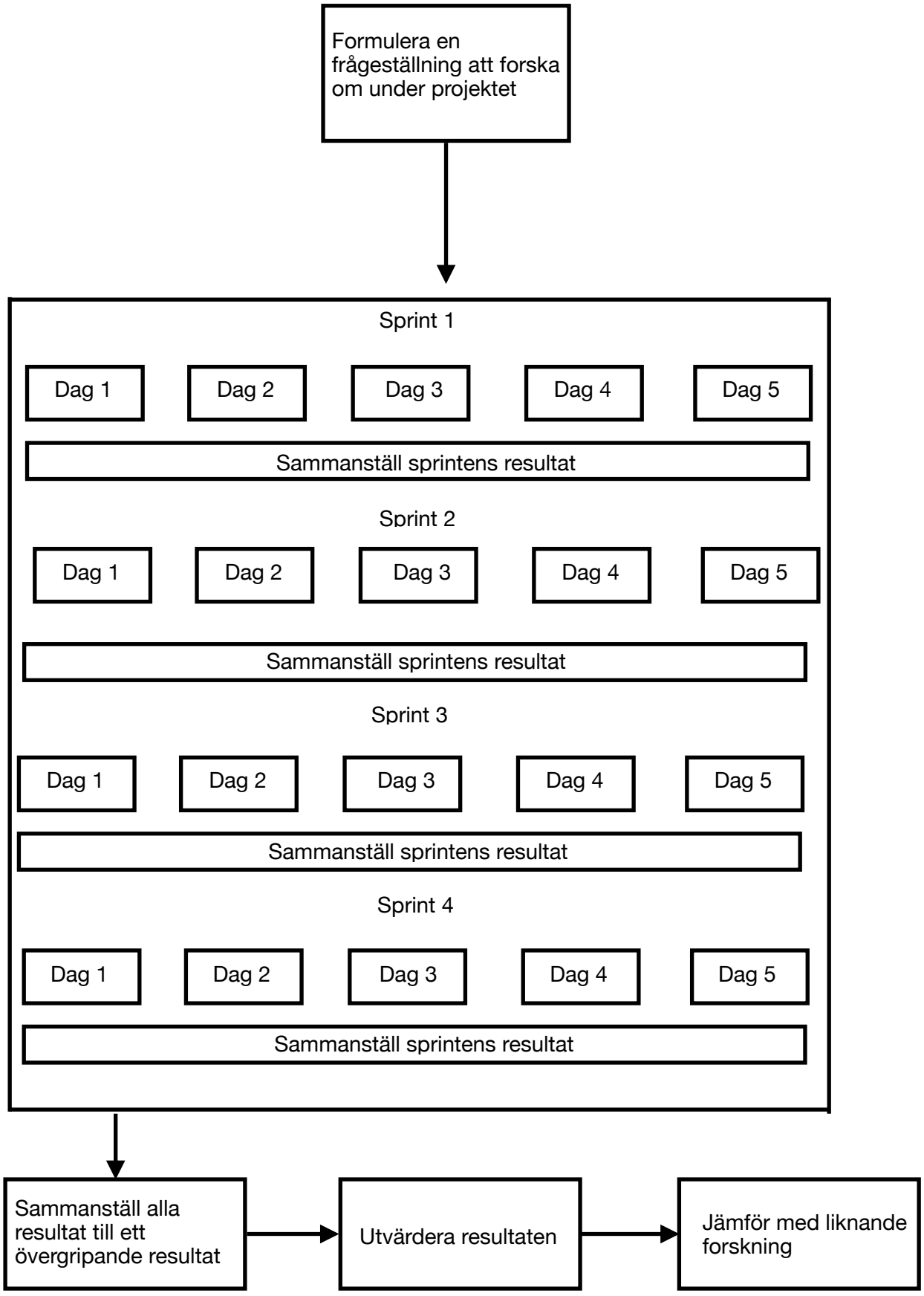
6 Griswold, William G. Program restructuring as an aid to software maintenance. University of Washington. 1991.

7 Griswold, William G. Program restructuring as an aid to software maintenance. University of Washington. 1991.

8 Griswold, William G. Program restructuring as an aid to software maintenance. University of Washington. 1991.

3. Metod

Nedan beskrivs forskningsmetoden som använts under projektet. Avsnitt 3.1 *forskningsfaser* redogör för de arbetssteg som projektet bestod av. Vidare i 3.2 beskrivs utvärderingsmodellen.



Figur 3.1. Utvärderingens faser som tillämpades under projektet

3.1 Forskningsfaser

Forskningsarbetet bestod av fem stycken faser som grafiskt beskrivs i figur 3.1 i föregående avsnitt. Mer utförligt beskrivs de fem faserna som följer: (1) Val av programvaruutvecklingspraxis och formulering av frågeställning. (2) Analys av det valda området inom projektet, och sammanställning av data för vardera sprint. (3) Efter slutfört projekt sammanställdes alla sprintars data till ett övergripande resultat. (4) Med den slutliga datan från hela projektet utvärderades sedan den insamlade datan. (5) Forskningens resultat jämfördes med existerande litteratur.

Existerande litteratur och referenser hittades med hjälp av Kungliga Tekniska Högskolans elektroniska bibliotek[9]. De sökord som användes för att finna litteraturen var: *software development refactoring, software development practices, efficiency of refactoring, agile software development.*

3.2 Utvärderingsmodellen

För att utvärdera refaktorisering som programvaruutvecklingspraxis upprättades tre stycken kriterium som bedömningsgrund varvid de förändringar som gjordes under projektets gång kunde klassificeras systematiskt. Då refaktorisering av kod dels är en subjektiv uppfattning var målsättningen att i så stor utsträckning som möjligt använda objektiva verktyg för att kvantifiera eventuell förändring av vald programvaruutvecklingspraxis. De kriterium som valdes var *prestanda, mängd kod* och *cyklomatisk komplexitet.*

- *Prestanda*:
  Prestanda är i majoriteten av fallen en viktig faktor för ett mjukvarusystem samtidigt som det är möjligt att mäta objektivt. Detta kriterium utvärderades genom att mäta källkodens exekveringstid före och efter utförd refaktorisering. Mätningen utfördes med kommandot ”npm test | gnonom” i Bash kommandotolk på macOS Mojave 10.14.5. Resultatet noterades som antal millisekunder (ms) och skillnaden mellan före och efter refaktorisering arkiverades som procent.

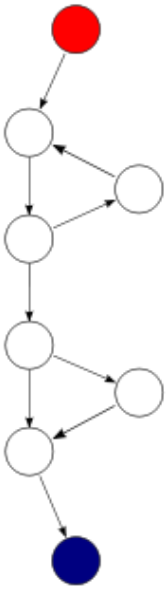
9 KTH bibliotek, 2019

- *Mängd kod:*

  Antalet rader kod som ett system består kan ha inverkan på dess prestanda. Det är också ett kriterium som är möjligt att mäta objektivt vilket är positivt. Tillvägagångssättet för att utvärdera detta kriterium var att mäta antalet rader kod före och efter utförd refaktorisering. Mätningen av antalet rader kod i godtycklig fil utfördes med kommandot ”wc -l <filnamn>” i Bash kommandotolk på macOS Mojave 10.14.5. Resultatet av mätningen presenteras som procentuell minskning eller ökning av antalet rader kod efter utförd refaktorisering.

- *Cyklomatisk komplexitet*

  Cyklomatisk komplexitet är ett sätt att mäta komplexiteten av ett program. Anledningen till att denna metod valdes grundar sig i att det är önskvärt att använda sig av en metod som ej förlitar sig på mänskliga bedömningar. Cyklomatisk komplexitet är en kvantitativ mätning av antalet linjärt oberoende komponenter i ett programs källkod. Komplexiteten återges som ett positivt heltal där ett större tal relaterar till högre komplexitet. Kortfattat kan metoden beskrivas som ett flödesdiagram av ett programs genomströmning som beskrivet i figur 3.2. Exekveringen startas i den röda noden och påbörjar sedan en iteration som består av en grupp med tre noder nedanför den röda. Vid utgången av iterationen finns ett villkor (gruppen nedanför iterationen). Slutligen avslutas programmet i den blå noden. Grafen av detta program består av nio kanter, åtta noder och en sammansatt komponent vilket resulterar i en cyklomatisk komplexitet av 9 - 8 + 2*1 = 3. Denna forskning har använt sig av ett genomsnittligt värde baserat på den cyklomatiska komplexiteten av en fils alla metoder.



Figur 3.2. Flödesdiagram av ett simpelt program

Den fullständiga teorin som metoden bygger på är alltför komplicerat för att beskrivas i detalj här.[10] För att utvärdera detta kriterium användes ett internetbaserat verktyg vid namn lizard.ws som automatiskt kalkylerar den cyklomatiska komplexiteten av angiven källkod. Fördelen med detta kriterium är att det ger möjligheten att objektivt klassificera komplexiteten av en källkod.

4. Resultat

I det här avsnittet redovisas resultaten för vardera kriterium som analyserats under projektet. Resultaten i sin ursprungliga form för vardera kriterium redovisas i en egen sektion följt av en kort förklaring. Tabellen som innefattar vardera kriteriums resultat har följande struktur: Första kolonnen visar vilken sprint som resultaten tillhör, andra kolonnen åskådliggör filnamnet varvid refaktorisering gjorts, tredje kolonnen beskriver resultaten före utförd programvaruutvecklingspraxis på samma sätt som fjärde kolonnen redovisar resultaten efter verkställande, slutligen visas den procentuella skillnaden mellan före och efter utförd refaktorisering i femte kolonnen. När det gäller rader i tabellerna syftar den sjunde raden till att visa en genomsnittlig förändring baserat på samtliga resultat under projektet.

4.1 Prestanda

Som beskrivet i avsnitt 3.2 *utvärderingsmodellen* gjordes en mätning av källkodens prestanda före och efter utförd refaktorisering.

Tabell 4.1. Exekveringstid före och efter refaktorisering

Sprint	Filnamn	Före refaktorisering	Efter refaktorisering	Procentuell skillnad
1	ChatWindow.js	63 ms	63 ms	0 %
2	WishListMembers.js	47 ms	46 ms	–2,13 %
3	WishListItem.js	76 ms	78 ms	2,63 %
4	InvitedWishlistPage.js	52 ms	53 ms	1,92 %
5	DashBoardNav.js	67 ms	66 ms	–1,49 %
Totalt	-	61 ms	61,2 ms	0,186 %

10 Mathworks, cyclomatic complexity. 2019.

Tabell 4.1 visar på någorlunda konstanta resultat före och efter utförd refaktorisering. Den största ökningen skedde under tredje sprinten i filen *WishListItem.js* där exekveringstiden ökade med 2 ms motsvarande 2,63%. Den största minskningen ägde rum under andra sprinten där refaktoriseringen minskade exekveringstiden för filen *WishListMembers.js* med 1 ms motsvarande 2,13%. I genomsnitt ökade exekveringstiden med 0,186% baserat på alla mätningar. En utförligare analys av detta återfinns i avsnitt 5. *Analys*.

4.2 Mängd kod

Som beskrivet i avsnitt 3.2 *utvärderingsmodellen* gjordes en mätning av antal rader kod i källkoden före och efter utförd refaktorisering.

Tabell 4.2. Antal rader kod före och efter refaktorisering. Mätningen innefattar både procentuell skillnad per fil och vecka samt totalt givet alla mätningar

Sprint	Filnamn	Före refaktorisering	Efter refaktorisering	Procentuell skillnad
1	ChatWindow.js	193	161	–16,6 %
2	WishListMembers.js	72	58	–19,4 %
3	WishListItem.js	189	166	–12,2 %
4	InvitedWishlistPage.js	137	108	–21,2 %
5	DashBoardNav.js	254	220	–13,4 %
Totalt	-	-	-	–16,5 %

Som beskrivet ovan i tabell 4.2 visar resultaten på en enhetlig reducering av antalet rader kod, dock med varierande grad. En intressant aspekt att notera är skillnaden i filstorlek där den minsta filen stod för den näst största minskningen av antalet rader kod sett till procentuell skillnad. Resultatet kommer att analyseras närmare och diskuteras i avsnitt 5. *Analys*.

4.3 Cyklomatisk komplexitet

Som beskrivet i avsnitt 3.2 *utvärderingsmodellen* gjordes en mätning av källkodens cyklomatiska komplexitet före och efter utförd refaktorisering.

Tabell 4.3. Den cyklomatiska komplexiteten angivit före och efter utförd refaktorisering

Sprint	Filnamn	Före refaktorisering	Efter refaktorisering	Procentuell skillnad
1	ChatWindow.js	1,6	1,5	–6,25 %
2	WishListMembers.js	1,25	1	–20 %
3	WishListItem.js	1	1	0 %
4	InvitedWishlistPage.js	1,16	1,16	0 %
5	DashBoardNav.js	1,5	1,25	–16,6 %
Totalt	-	-	-	–8,57 %

Inverkan på den cyklomatiska komplexiteten i relation till utförd refaktorisering gav blandade resultat. Som mest noterades en minskning av den cyklomatiska komplexiteten i filen *WishListMembers.js* där minskningen uppgick till 20%. I två filer var den cyklomatiska komplexiteten identisk både före och efter utförd refaktorisering. Ingen fil visade på ökad cyklomatisk komplexitet. En utförligare analys av resultaten återfinns i avsnitt 5. *Analys*.

5. Analys

I detta avsnitt kommer resultaten från 4. *Resultat* att beskrivas och främst analyseras närmare. Vardera kriteriums resultat analyseras i en individuell sektion.

5.1 Prestanda

Resultaten från mätningen av källkodens prestanda visar på en genomsnittlig ökning av exekveringstid med 0,186% baserat på alla filers resultat, vilket är ogynnsamt. Som mest noterades en ökning på 2,63% efter refaktorisering medan den största minskningen var 2,13%. Skillnaden mätt i tid var dock 2 respektive 1 ms vilket i sig inte är en noterbar skillnad för mänskligt användande.

Resultaten som uppmätts grundar sig på vilken av de metoder från avsnitt 2.4 *Refaktorisering* som använts. Till exempel lär inte de tekniker som avser att förbättra kodens relevans ha inverkan på den slutliga prestandan. Detta inkluderar sådant som namnbyte av klasser, metoder och variabler.

Då en av de refaktoriseringsmetoder som används inkluderar *mängd kod* i kombination med att avsikten var att reducera antalet rader kod kan detta ur vissa aspekter haft en negativa inverkan på prestandan. Då utvecklaren strävar efter färre antal rader kod kan detta resultera i mer komplicerad logik som kompilatorn måste kompilera. Då dagens datorer generellt har en kraftfull beräkningsförmåga resulterar detta i att ett par fler assemblerinstruktioner kommer ha en nästintill omätbar skillnad på prestandan. Som exempel på detta är att då utvecklaren implementerar fler funktioner i en källkod gör detta att kompilatorn måste hoppa mellan dessa för att utföra önskad funktionalitet, vilket trots vissa vunna fördelar kan ha en negativ inverkan på prestandan.[11]

De variationer som registrerats behöver i sig inte bero på refaktoriseringen. Prestanda är en mätning som beror på flera variabler som påverkas av ett stort antal olika områden. Exempel på faktorer som påverkar prestandan i ett system kan vara temperatur, konfiguration av servern eller oväsentliga processer som körs i bakgrunden men som inte hör till den källkod som testas. För att erhålla så

11 Wellisson G. P. da Silva. et al. Evaluation of the impact of code refactoring on embedded software efficiency. SBRC. Januari 2010.

felfria resultat som möjligt bör man utföra testning på en dedikerad testrigg där dessa saker i större utsträckning kan kontrolleras.

Resultaten från denna forskning har visat att refaktorisering som metod för att förbättra ett systems prestanda möjligtvis inte är det mest effektiva sättet att uppnå önskat resultat. Om prestanda är det mest önskvärda kriteriet att förbättra bör en annan metod väljas.

5.2 Mängd kod

Resultaten från att refaktorisera källkoden visar på en kollektiv minskning av filstorleken baserat på de fem filerna som ingick i mätningen. Den största minskningen uppstod i filen *InvitedWishlistPage.js* där antal rader kod minskade med 29 st vilket stod för en minskning på 21,2%. Som beskrivet i föregående avsnitt 5.1 *Prestanda* beror det slutgiltiga resultatet på vilken typ av refaktorisering som används.

En intressant aspekt är att då man använder tekniker för att omstrukturera kod till mer logiska delar lär detta i de allra flesta fall resultera i en ökning av antalet rader kod på grund av att utvecklaren skapat nya funktioner eller dylikt. Dock är detta inte något som kan ses i forskningens resultat. Kan detta bero på att det använts en kombination av tekniker som gjort att den eventuella ökning har absorberats av andra tekniker som möjliggör högre abstraktion? Denna forskning har varit alltför begränsad för att kunna kunna analysera detta närmare.

Då antalet rader kod minskas kan detta resultera i dels att det blir lättare för utvecklare att underhålla källkoden, men det kan också resultera i mer komplicerade kodstycken där man försöker kombinera flera logiska delar till en vilket resulterar i mer invecklad kod för eventuella nya utvecklare att förstå vilket är negativt för underhållbarheten. Den minskning av antalet rader kod som resultatet visar på är enligt Robillard et al. en positiv aspekt då det bidrar till att lättare kunna förstå kodens syfte.[12]

Resultaten från denna forskning har visat att refaktorisering som metod för att minska antalet rader kod i ett systems källkod möjligtvis inte bidrar till ökad kvalitet på källkoden som helhet. För att

12 Robillard, M. et al. Recommendation Systems in Software Engineering. IEEE Software. 2014.

uppnå önskvärda och konsekventa resultat bör man troligen endast praktisera en refaktoriseringsteknik.

5.3 Cyklomatisk komplexitet

Resultaten från att refaktorisera källkoden visar på i vissa fall en minskning av den cyklomatiska komplexiteten. Den största minskningen uppgick till 20% medan två filer lämnades oförändrade efter refaktorisering.

Som beskrivet i avsnitt 2. *Bakgrund* är systemet forskningen grundar sig på byggt med hjälp av ramverket React.js som till sin natur använder sig av ett flertal mindre komponenter som i sin tur består av ett flertal funktioner som står för applikationens funktionalitet. Systemet i fråga består inte av några större algoritmer eller längre logiska genomströmningar, därav den låga cyklomatiska komplexiteten som presenteras i avsnitt 4. *Resultat.* Skulle refaktorisering som metod ge större skillnader i ett system bestående av större och mer komplicerade funktioner?

Forskningen har även varit alltför begränsad för att djupare kunna analysera vilken av de tre tekniker från avsnitt 2.4 som lett till minskningen av cyklomatisk komplexitet. Har det till exempel varit positivt eller negativt att extrahera kodblock till nya funktioner? Det kan också spela roll hur man presenterar den cyklomatiska komplexiteten. Att presentera resultatet som ett medelvärde av alla funktioners cyklomatiska komplexitet kan ge en skev bild där vissa tekniker framstår som bättre än andra. Att beräkna hela filens cyklomatiska komplexitet skulle mycket väl kunna ge andra resultat beroende på hur man utför refaktoriseringen.

En lägre cyklomatisk komplexitet är enligt litteraturen synonymt med bättre underhållbarhet i längden. I rapporten ”Does refactoring improve reusability?” skriver R Moser, A Sillitti, P Abrahamsson och G Succi att lägre komplexitet gör det lättare att återanvända kod i framtiden. Fördelarna med refaktorisering som programvaruutvecklingspraxis ska dock noga övervägas mot den extra tid och arbetsinsats det tar att utföra dessa tekniker vilket återges i rapporten ”Refactoring - improving coupling and cohesion of existing code”.[13]

13 Du Bois, B. et al. Refactoring - improving coupling and cohesion of existing code. 2004.

Generellt verkar refaktorisering ha en positiv inverkan på källkodens cyklomatiska komplexitet som i längden bidrar till ökad underhållbarhet.

6. Slutsats

Den viktigaste slutsats som kan dras av denna forskning är att refaktorisering som programvaruutvecklingspraxis har olika inverkan på de tre kriterium som undersökts i rapporten. Då det huvudsakliga målet är att förbättra ett systems prestanda visar resultaten att refaktorisering troligen inte är det mest effektiva sättet att uppnå förbättrad prestanda. Forskningen visar dock konsekventa resultat när det gäller minskningen av antalet rader kod, däremot är inte minskningen synonym med bättre kvalitet på koden som helhet då minskningen kan bero på att flera enklare rader kod kombinerats till endast en men mer komplicerad rad kod. Den cyklomatiska komplexiteten har också sett en viss positiv förbättring på grund av refaktoriseringen som i sin tur bidrar till att lättare kunna underhålla koden framöver. Som litteraturen beskriver ska dock dessa kriterium ställas i relation till den tid och arbetsinsats det tar att utföra dem.

Framtida forskning bör utgå från att utföra liknande eller fler tester på en dedikerad testrigg för att bättre kunna övervaka de förhållanden som testet utförs under. För att få en bättre överblick över vilka tekniker som minskar antalet rader kod bör framtida forskning utgå från att upprätta metoder för att samla in information om detta. Förslagsvis genom att endast applicera en av de tre teknikerna från avsnitt 2.4 åt gången. För att förstå refaktoriseringens fulla potential när det gäller att reducera den cyklomatiska komplexiteten bör man utföra liknande tester på mer komplicerad källkod med högre tidskomplexitet.

7. Referenser

Bassil, Youssef. A Simulation Model for the Waterfall Software Development Life Cycle. *Lebanese Association for Computational Sciences.* 2012. https://arxiv.org/pdf/1205.6904.pdf (Hämtad 2019-05-29)

Dingsøyr, Torgeir. et al. A decade of agile methodologies: Towards explaining agile software development. Juni 2012. https://www.sciencedirect.com/science/article/pii/S0164121212000532?via%3Dihub (Hämtad 2019-05-28)

Du Bois, B. et al. Refactoring - improving coupling and cohesion of existing code. 2004. https://www.researchgate.net/publication/4114677_Refactoring_-_Improving_coupling_and_cohesion_of_existing_code (Hämtad 2019-06-02)

Griswold, William G. Program restructuring as an aid to software maintenance. *University of Washington*. 1991. http://cseweb.ucsd.edu/~wgg/Abstracts/gristhesis.pdf (Hämtad 2019-05-30)

Jones, Caspers. Software Engineering Best Practices. 2010. https://dl.acm.org/citation.cfm?id=1594755 (Hämtad 2019-05-29)

Kajko-Mattson, Mira; Docent vid skolan för elektroteknik och datavetenskap, Kungliga Tekniska högskolan. Föreläsning 2019-03-18.

KTH, https://www.kth.se/biblioteket

Lotz, M. Waterfall vs. Agile: Which Methodology is Right for Your Project? 2013. https://www.seguetech.com/waterfall-vs-agile-methodology/ (Hämtad 2019-05-28)

Mathworks, cyclomatic complexity. https://www.mathworks.com/discovery/cyclomatic-complexity.html (Hämtad 2019-05-30)

Robillard, M. et al. Recommendation Systems in Software Engineering. *IEEE Software*. 2014. https://ieeexplore.ieee.org/abstract/document/5235134 (Hämtad 2019-06-01)

Wellisson G. P. da Silva. et al. Evaluation of the impact of code refactoring on embedded software efficiency. *SBRC*. Januari 2010. https://www.researchgate.net/publication/229519223_Evaluation_of_the_impact_of_code_refactoring_on_embedded_software_efficiency (Hämtad 2019-05-31)