De wijzigingshistorie als informatiebron voor toekomstige wijzigingen

The change history as a source of information for future changes

Master's Thesis

Riccardo Lippolis

September, 2008

Universiteit van Amsterdam

vrije Universiteit amsterdam

Afstudeerdocent:
Hans Dekkers

Begeleider:
Patricia Lago

logica

Projectleider:
Bram Vranken

Begeleider:
Edwin Essenius

"If history repeats itself, and the unexpected always happens, how incapable must Man be of learning from experience."

George Bernard Shaw
Iers toneelschrijver & socialist (1856–1950)

Samenvatting

Het grootste gedeelte van de kosten van softwareprojecten gaat op aan onderhoud. Het wijzigen van een softwaresysteem in de onderhoudsfase blijkt een foutgevoelige taak te zijn. Ontwikkelaars zijn vaak niet in staat om de volledige impact van hun wijzigingen te voorspellen. Het incorrect uitvoeren van een kleine wijziging kan hierdoor grote gevolgen hebben.

In dit onderzoek is een experiment uitgevoerd waarin bekeken is in hoeverre informatie uit de wijzigingsgeschiedenis van een softwareproject hergebruikt kan worden bij het voorspellen van de impact van toekomstige wijzigingen. Binnen twee commerciële projecten en twee open source-projecten is gezocht naar tekstuele overeenkomsten tussen de beschrijvingen van wijzigingen. Vervolgens is bekeken in hoeverre de wijzigingen ook leiden tot aanpassingen in dezelfde source code-bestanden.

Uit het experiment is gebleken dat voor een klein percentage wijzigingen volledig automatisch bepaald kon worden welke bestanden aangepast moesten worden. Dit lage percentage is niet hoog genoeg om te leiden tot betrouwbare resultaten voor een praktische tool. De tekstuele relaties bleken vaak te algemeen van aard te zijn. Verder onderzoek is nodig om te onderzoeken op welke manier betrouwbaardere tekstuele relaties gevonden kunnen worden tussen wijzigingen.

Abstract

Software maintenance is the most expensive phase of the software development cycle. Performing a maintenance task on a software system appears to be a very error-prone task. Developers are often not capable of assessing the complete impact of the software change. Inadvertently performing a small change can cause ripple effects which in turn can be fatal for a software system.

In this research an experiment is performed assessing the effectiveness of using change history knowledge for predicting the impact of future changes. The change histories of two commercial and two open source software projects have been analysed, searching for textual relations between the commit logs of the changes, based on word similarities. For the textual similar changes the lists of altered files have been compared.

The results of the experiment indicate that for a small percentage of the changes the impact could, in fact, be fully predicted. However, this percentage is not large enough to warrant the use of the examined method in practice. The textual relations often turned out to be too generic. Further research is needed to explore new, more reliable, ways of finding textual relations.

Voorwoord

Voor u ligt het afstudeerverslag van Riccardo Lippolis, met als onderwerp: 'De wijzigingshistorie als informatiebron voor toekomstige wijzigingen'. Dit verslag is het resultaat van het afstudeertraject ter afsluiting van de Masterstudie Software Engineering aan de Universiteit van Amsterdam. Het project is uitgevoerd bij Working Tomorrow, het afstudeerprogramma van Logica, in Rotterdam.

Het is een druk, maar vooral erg leerzaam jaar geweest waarvoor ik alle docenten en medestudenten wil bedanken.

Verder gaat mijn dank uit naar de volgende personen:

- Hans Dekkers (UvA) en Patricia Lago (VU) voor de kritische begeleiding die mij zeer hebben geholpen om mijn weg te vinden in het afstudeertraject.
- Bram Vranken en Edwin Essenius voor de intensieve begeleiding vanuit Logica.
- Alle mensen binnen Logica (en een paar daarbuiten) voor de zeer waardevolle informatie en de altijd aanwezige hulp.
- Alle collega-WT'ers, die de sfeer er al die tijd in gehouden hebben.
- En niet te vergeten, mijn familie en vrienden, die, ondanks dat ze vaak geen flauw benul hadden van wat ik precies deed, mij toch al die tijd bleven steunen.

Riccardo Lippolis, Rotterdam, september 2008

Inhoudsopgave

1 Inleiding **1**

1.1 Leeswijzer 2

2 Vooronderzoek **3**

2.1 Literatuurstudie 3

2.1.1 Wat is traceability? 3

2.1.2 Welke doelen heeft traceability? 4

2.1.3 Hoe wordt traceability toegepast? 4

2.1.4 Welke problemen spelen er rond traceability? 6

2.1.5 Conclusie 7

2.2 Interviews 8

3 Onderzoeksformulering **11**

3.1 Doelstelling 11

3.2 Rationale 12

3.3 Gerelateerde werken 12

3.4 Kernproblemen 13

3.5 Hypothesen 13

3.6 Scope 14

3.7 Aanpak 15

3.8 Gebruikte projectdata 15

3.9 Kwaliteit 16

3.9.1 Betrouwbaarheid en validiteit 17

3.9.2 Generaliseerbaarheid 17

4 Onderzoeksaanpak **19**

4.1 Meetmethode 19

4.1.1 Wijzigingshistorie indexeren 20

4.1.2 Tekstuele overeenkomsten zoeken 21

4.1.3 Code-overeenkomsten bepalen 22
4.2 Kwantitatieve analyse 25
4.3 Kwalitatieve analyse 25

5 Resultaten en analyse **29**
5.1 Kwantitatief onderzoek 29
5.2 Kwalitatief onderzoek 33

6 Conclusies en aanbevelingen **39**
6.1 Conclusies 39
6.2 Aanbevelingen 41

Bibliografie **43**

Bijlagen

A Traceability tools **51**
A.1 Enkele tools uit de literatuur 51
A.2 Enkele commerciële tools 52

B Grafieken **55**

C Stopwoorden **65**

Hoofdstuk 1

Inleiding

Hoe kan worden bepaald welke bestanden aangepast moeten worden in een softwaresysteem voor een wijzigingsverzoek? Het incorrect uitvoeren van een wijziging door bepaalde bestanden te vergeten, kan achteraf leiden tot bijwerkingen (ongewenst gedrag van een systeem) of rimpeleffecten (een kleine wijziging die grote gevolgen heeft voor andere delen van een systeem). Bij het uitvoeren van de wijzigingen worden namelijk eenvoudig bepaalde delen van het systeem over het hoofd gezien. De realiteit is dat het maken van impact analyses een tijdrovende en complexe taak is (Arnold & Bohner, 1996).

Ontwikkelaars en projectmanagers zijn niet altijd in staat om zelf de impact van wijzigingen te voorspellen (Lindvall & Sandahl, 1998). Omdat circa 60 procent van de softwarekosten op gaat aan onderhoud (Glass, 2002), loont het om te onderzoeken hoe het impact analyse-proces verbeterd kan worden.

Op verschillende manieren wordt getracht om met behulp van allerlei methoden en technieken het onderhoudsproces te vereenvoudigen. Een hierbij veelgenoemd begrip is traceability. Traceability wordt vaak aangedragen als mogelijk hulpmiddel in het impact analyse-proces (Abbattista et al., 1994; Antoniol et al., 2001; Bratthall et al., 2000; Cleland-Huang et al., 2004; Egyed & Grünbacher, 2002; Gotel & Finkelstein, 1994; Harrington & Rondeau, 1993; Jönsson, 2005; von Knethen, 2002; Ramesh et al., 1997; Strens & Sugden, 1996; Watkins & Neal, 1994). Een van de grote problemen met traceability is echter, dat het al snel om grote hoeveelheden gegevens gaat, welke lastig te structureren zijn, en vooral moeilijk te onderhouden zijn wanneer de software evolueert.

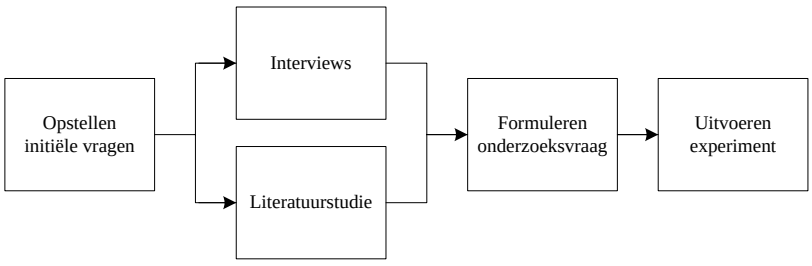
Automatische generatie van traceability links is een mogelijke oplossing voor dit traceability-probleem. In dit onderzoek zal daarom een vorm van automatische traceability op basis van de wijzigingshistorie (uit het versiebe-

heersysteem) van softwaresystemen onderzocht worden. Er zal onderzocht worden of het mogelijk is om kennis over eerder uitgevoerde wijzigingen te hergebruiken bij het uitvoeren van wijzigingen.

1.1 Leeswijzer

Figuur 1.1 geeft de onderzoeksaanpak globaal weer. De volgorde van de hoofdstukken stemt overeen met het verloop van het onderzoek.

Ten eerste wordt in hoofdstuk 2 het resultaat gegeven van de literatuurstudie en de interviews. Hoofdstuk 3 formuleert het experiment welke het resultaat is van het vooronderzoek. De aanpak van het experiment wordt in hoofdstuk 4 uitgebreid beschreven, gevolgd door de resultaten in hoofdstuk 5. Ten slotte worden de conclusies en aanbevelingen gegeven in hoofdstuk 6.



Figuur 1.1: De onderzoeksaanpak

Hoofdstuk 2

Vooronderzoek

Voorafgaand aan het onderzoek gedefinieerd in hoofdstuk 3, heeft een vooronderzoek plaatsgevonden. In dit vooronderzoek is er een literatuurstudie uitgevoerd en zijn verscheidene werknemers van Logica en andere personen uit het traceability-vakgebied geïnterviewd.

In paragraaf 2.1 worden de resultaten van de literatuurstudie beschreven en paragraaf 2.2 geeft een overzicht van de behaalde inzichten uit de interviews.

2.1 Literatuurstudie

Het doel van de literatuurstudie was om inzicht te krijgen in het begrip traceability. Hierbij is gekeken naar de mogelijkheden en problemen. De interviews waren bedoeld om de theorie over traceability te valideren met de praktijk. Ook zijn de interviews gebruikt om te bepalen in hoeverre traceability binnen Logica wordt toegepast en waar mogelijk ruimte tot verbetering is.

2.1.1 Wat is traceability?

Traceability is een ruim begrip, dat op verschillende manieren wordt geïnterpreteerd en gedefinieerd (Gotel & Finkelstein, 1994; Ramesh & Jarke, 2001). Het IEEE heeft ook een definitie van traceability, waaraan duidelijk het brede karakter van het begrip traceability te zien is:

> *"The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate*

relationship to one another" (IEEE, 1990)

In een softwareontwikkelproces zijn verschillende 'producten' aanwezig, zoals een requirementsdocument, architectuurdocument, source code, testdocument, enzovoorts. Elk document bevat weer een aantal 'elementen'. Een requirementsdocument bevat bijvoorbeeld onder andere requirements, en een testdocument bevat test cases. Traceability betreft het vermogen om relaties te leggen tussen verschillende 'elementen' in een softwareontwikkelproces. Deze elementen kunnen verschillen op basis van granulariteit (source code kan worden opgesplitst in modules, klassen, procedures of zelfs regels code) of op basis van onderwerp (een requirementsdocument bevat onder andere requirements, stakeholderbeschrijvingen en een scope).

De relaties kunnen hierbij ook verscheidene vormen hebben, waarvan er twee in de definitie genoemd worden (de predecessor-successor-relatie en de master-subordinate-relatie). Afhankelijk van het type element en het doel verschilt ook het soort relatie dat gelegd wordt tussen de elementen (Ramesh & Jarke, 2001). Een relatie tussen een requirement en een ontwerpbeslissing kan bijvoorbeeld inhouden dat de ontwerpbeslissing de implementatie beschrijft van dat requirement, terwijl een relatie tussen een source code module en een test case kan betekenen dat de test case de source code module verifieert.

2.1.2 Welke doelen heeft traceability?

In het CMMI[1] wordt bi-directionele requirements traceability vermeld als een best practice. Regelmatig wordt traceability aangeraden of verplicht gesteld (Arkley & Riddle, 2005; Asuncion et al., 2007; Cleland-Huang, 2006; Harrington & Rondeau, 1993; IEEE Std 830-1998, 1998; Pohl, 1996; Ramesh et al., 1997). Traceability wordt in de literatuur door velen gezien als een nuttig onderdeel van een ontwikkelproces. Er worden dan ook een ruim aantal doelen vernoemd waar traceability voor gebruikt zou kunnen worden.

In tabel 2.1 wordt een overzicht gegeven van de doelen van traceability en wordt aangegeven waar deze doelen in de literatuur zijn beschreven.

2.1.3 Hoe wordt traceability toegepast?

De manier waarop traceability wordt toegepast blijkt sterk af te hangen van de manier waarop het softwareontwikkelproces ingericht is.

[1] `http://www.sei.cmu.edu/cmmi/`

Doel	**Bronnen**
Change management	Abbattista et al. (1994); Antoniol et al. (2001); Arnold & Bohner (1996); Bratthall et al. (2000); Cleland-Huang et al. (2004); Egyed & Grünbacher (2002); Gotel & Finkelstein (1994); Harrington & Rondeau (1993); Jönsson (2005); Von Knethen (2002); Ramesh et al. (1997); Strens & Sugden (1996); Watkins & Neal (1994)
Conflicten in requirements ontdekken	Wieringa (1995)
Inzicht krijgen in een applicatie	Antoniol et al. (2002); De Lucia et al. (2005); Gotel & Finkelstein (1994); Marcus et al. (2005); Pohl (1996)
Requirements accountability	Harrington & Rondeau (1993); Watkins & Neal (1994)
Verificatie & validatie	Antoniol et al. (2002); Gotel & Finkelstein (1994); Harrington & Rondeau (1993); Hayes et al. (2004); Ramesh et al. (1997); Watkins & Neal (1994); Wieringa (1995)
Software hergebruik	Abdelfettah (2005); Antoniol et al. (2002)
Test coverage analyse	Antoniol et al. (2002); Gotel & Finkelstein (1994); Hayes et al. (2004); Marcus et al. (2005)

Tabel 2.1: Doelen van traceability

In de literatuur worden verscheidene traceability-oplossingen besproken, welke vaak gepaard gaan met een bepaalde vorm van tooling. Daarnaast zijn er ook een aantal commerciële tools beschikbaar, die zich voornamelijk richten op requirements traceability. De verschillende oplossingen variëren van eenvoudige tools waar veel informatie handmatig in bijgehouden moet worden, tot semi-automatische of volautomatische oplossingen.

Hoewel de manier waarop de traceability-informatie wordt aangelegd flink kan verschillen per oplossing, wordt er als datamodel vaak gekozen voor een relationeel model. De informatie die hierbij in het model wordt vastgelegd verschilt echter wel per oplossing. Andere methoden voor het vastleggen van traceability zijn een traceability matrix (tabel) en het gebruik van hyperlinks in documenten. Het nadeel van deze vormen is echter dat er geen onderscheid wordt gemaakt tussen verschillende soorten relaties, maar alleen wordt vastgesteld dat er een relatie is tussen twee 'elementen' (Ramesh & Jarke, 2001).

Bijlage A geeft een overzicht van enkele traceability tools. Dit overzicht toont aan dat er al vele verschillende tools ontwikkeld zijn. In combinatie

met de verscheidenheid aan doelen toont dit de veelzijdigheid van het begrip traceability aan. Traceability kan verschillende doelen hebben (zoals beschreven in paragraaf 2.1.2) en een eventuele tool moet ingericht worden op dat specifieke doel. Dit is niet altijd mogelijk, want tools zijn vaak specifiek gericht op een bepaald doel. Daarnaast hebben de verschillende traceerbare elementen weer verschillende methoden waarop deze aan elkaar gerelateerd moeten worden. De veelzijdigheid van het 'traceability-probleem' in zijn algemeenheid zorgt hierdoor voor het veelzijdige aanbod van tools.

2.1.4 Welke problemen spelen er rond traceability?

Het blijkt lastig te zijn om traceability in de praktijk toe te passen en deel uit te laten maken van een softwareontwikkelproces. Er zijn verschillende problemen te identificeren.

Ten eerste vereisen verschillende doelen van traceability ook verschillende typen traceability-informatie en een verschillende granulariteit (Gotel & Finkelstein, 1994; Ramesh & Jarke, 2001; Ramesh et al., 1997). Ook gebruiken bedrijven verscheidene softwareontwikkelmethoden die onderling sterk kunnen verschillen. Door deze veelzijdigheid is het tot op heden nog niet mogelijk gebleken om een generieke traceability-oplossing te definiëren die direct implementeerbaar is in elke bedrijfssituatie. Wel zijn er referentiemodellen (Ramesh & Jarke, 2001) en standaard technische oplossingen (Cleland-Huang et al., 2004) bedacht, die bedoeld zijn als startpunt voor het ontwikkelen van een traceability-aanpak voor een specifieke situatie. Echter zijn deze modellen vaak zo abstract, dat het nut ervan in twijfel getrokken mag worden.

Ten tweede is voor het implementeren van een traceability-oplossing een grote investering nodig in tijd en geld, en het is niet altijd duidelijk hoeveel deze investering later zal opleveren. Kosten betreffen bijvoorbeeld de aanschaf of ontwikkeling van tools, training van personeel en een initieel productiviteitsverlies vanwege de verandering in het ontwikkelproces (Arkley et al., 2002). Door deze kosten is het heel belangrijk dat aangetoond kan worden welke voordelen het traceability-proces oplevert (Asuncion et al., 2007), anders zal het management niet gemotiveerd genoeg zijn om zich bezig te houden met traceability. Ook het personeel zal duidelijk het voordeel van traceability moeten inzien, omdat het voor hen vooral een grote tijdsinvestering is (Arkley et al., 2002; Cleland-Huang, 2006; Gotel & Finkelstein, 1994; Hayes et al., 2004; Ramesh et al., 1997; Wieringa, 1995).

Ten derde blijkt het aantonen van het daadwerkelijke voordeel van traceability een heikel punt te zijn. In de wetenschappelijke wereld zijn er onderzoeken gedaan naar de voor- en nadelen van traceability. Op het gebied van

traceability in combinatie met change impact analyses zijn er weinig overtuigende onderzoeken te vinden waarin beweerd wordt dat traceability een daadwerkelijk voordeel biedt. De onderzoeken die een voordeel aantonen, zijn voornamelijk gebaseerd op gebruikerspercepties (Gotel & Finkelstein, 1994; Neumuller & Grunbacher, 2006; Ramesh & Jarke, 2001). Een nadeel van dergelijke onderzoeken is dat het lastig te bepalen is in hoeverre de subjectiviteit van de gebruikers het onderzoek hebben beïnvloed. Onderzoeken waarbij bijvoorbeeld empirisch is gemeten wat het voordeel is van de aanwezigheid van een design rationale bij het maken van change impact analyses, toonden in sommige gevallen een licht voordeel aan, maar leverden geen overtuigende resultaten op (Abbattista et al., 1994; Bratthall et al., 2000).

Verder spelen er nog enkele andere punten die traceability bemoeilijken. Er is een gebrek aan goede tooling die aansluit op het bestaande ontwikkelproces, waardoor bedrijven vaak zelf tools ontwikkelen die wel de gewenste functionaliteit bieden (Asuncion et al., 2007; Gotel & Finkelstein, 1994; Ramesh et al., 1997).

Ten slotte speelt nog het probleem van de zogeheten *requirements explosion* (Glass, 2002). Een high-level requirement kan uitmonden in een aantal low-level requirements. Deze requirements op zich kunnen ook weer exploderen in een groot aantal implementatiebeslissingen. Hierdoor is het een moeilijke taak om te analyseren op welke manier alle elementen in het ontwikkelproces aan elkaar gerelateerd zijn. Het kan al heel snel een wirwar van verbindingen worden, waarbij de vraag ontstaat in hoeverre het de investering waard is om dit te onderhouden. Glass (2002) noemt een klein voorbeeld van een high level requirement die gelinkt kan worden aan 50 designkeuzes, die elk weer naar een groot aantal source code elementen gelinkt kunnen worden. Ramesh (1998) noemt voorbeelden van systemen met 1000 tot 10.000 requirements, waarbij het aantal links dus vele malen hoger zal liggen. Een toename van het aantal requirements zal het aantal links meer dan lineair doen stijgen. Het moeten onderhouden van een dergelijk groot aantal traceability-links kan ervoor zorgen dat de kosten het uiteindelijke voordeel zullen overstijgen.

2.1.5 Conclusie

Uit de literatuurstudie is gebleken dat er al veel onderzoek is gedaan op het gebied van traceability, maar dat er ook nog een aantal vragen open staan.

In de literatuur zijn vele modellen te vinden voor allerlei mogelijke traceability-situaties (paragraaf 2.1.3). Wat hierbij nog onduidelijk blijft is in hoeverre deze modellen daadwerkelijk toegepast (kunnen) worden in de praktijk. Hebben bedrijven baat bij dergelijke modellen? Is het wel mogelijk

om goede modellen te ontwikkelen voor het implementeren van traceability, als een model aan de ene kant algemeen toepasbaar moet zijn, maar aan de andere kant een erg contextgevoelig probleem moet oplossen?

De onderhoudbaarheid van de (mogelijk vele) traceability-links is nog steeds een erg actueel onderzoeksonderwerp. Vaak wordt geautomatiseerde traceability gezien als een mogelijke oplossing voor dit probleem. Is het mogelijk om met een geautomatiseerde aanpak de onderhoudbaarheid te vereenvoudigen, maar ook de nauwkeurigheid van handmatige traceability te behouden?

Traceability wordt vaak genoemd als een hulpmiddel bij het uitvoeren van change impact analyses (paragraaf 2.1.2). Onderzoeken waarin dit beweerd wordt zijn echter vaak slechts onderbouwd met interviews en enquêtes, waarbij onduidelijk is hoeveel voordeel er nu echt behaald wordt. Enkele empirische onderzoeken waarin wordt getracht om met metrieken het voordeel te meten, hebben geen overtuigende resultaten opgeleverd. Dus of het daadwerkelijk voordeel biedt, is ook een openstaande vraag.

2.2 Interviews

Als onderdeel van het vooronderzoek zijn er interviews gehouden. Hieronder wordt toegelicht welke personen geïnterviewd zijn, welke vragen gesteld zijn en wat het resultaat van de gesprekken is.

Geïnterviewden

In totaal zijn 12 personen geïnterviewd, bestaande uit software engineers, testers, projectleiders, een systeemontwerper en een technisch manager. Het grootste gedeelte van de groep geïnterviewden is werkzaam bij Logica, een internationaal IT-bedrijf met circa 40.000 werknemers in 41 landen. De overige geïnterviewden zijn werkzaam bij andere IT-bedrijven.

Vragen

Voor alle interviews is een korte vragenlijst gebruikt als leidraad in het gesprek. De vragenlijst bestaat uit de volgende punten:

- Maakt u of uw projectgroep gebruik van traceability-methodieken?
- Zo ja, om welke redenen wordt dit gedaan? Zo niet, waarom niet?
- Wat is uw eigen mening betreffende het onderwerp traceability?

- Wordt er gebruik gemaakt van (custom-made / out-of-the-box) tooling en/of standaardmethodieken?

Bovendien is elk van de geïnterviewden gevraagd naar de mogelijkheden om een experiment uit te kunnen voeren binnen zijn project.

Resultaten

Binnen Logica is er geen standaard methodiek voor het toepassen van enige vorm van traceability in een project. De gekozen ontwikkelmethodiek, en daarbij ook de overweging om traceability toe te passen binnen de projecten, kan afhangen van de wensen van de klant, of van de voorkeuren van de projectmedewerkers.

Er zijn wel een aantal templates beschikbaar voor documenten, waarbij in een van de templates een hoofdstuk is gereserveerd voor een Requirements Traceability Matrix. Deze wordt in de praktijk echter niet altijd ingevuld. Als reden hiervoor wordt vooral tijdsdruk genoemd, en het feit dat de klant er niet altijd behoefte aan heeft.

Toch is traceability niet altijd afwezig. Het kan bijvoorbeeld een eis zijn van de klant (*verifiability*), maar ook de projectmedewerkers zelf kiezen er soms voor.

De problemen die optreden op het moment dat traceability wordt toegepast, bevestigen de problemen die in de literatuur naar voren komen (zie paragraaf 2.1.4).

Een voorbeeld van een probleem is het gebrek aan goede tooling. Soms wordt er bijvoorbeeld gebruik gemaakt van zelfgebouwde scripts wegens het ontbreken van een bepaalde functionaliteit in de aanwezige traceability tools. Ook zijn de tools niet altijd geïntegreerd in de bestaande ontwikkeltools, waardoor traceability wordt gezien als een apart onderdeel, waarbij de traceability-informatie vaak dan ook achteraf wordt ingevoerd.

Het eerder genoemde tijdsdruk-probleem treedt vaak op. Een voorbeeld hiervan is een project waarin de requirements als issues in de bugtracker zijn ingevoerd. Elke source code commit werd gekoppeld aan een bepaalde issue uit de bugtracker (een requirement), om traceability-relaties te vormen tussen requirements en source code. Alhoewel niet iedereen hier het voordeel van in zag, werd deze methode in de beginfase van het project goed toegepast. Naarmate de deadline van het project in zicht kwam, werd er echter steeds minder aandacht aan besteed, omdat het geen prioriteit meer had. Er werden bijvoorbeeld meerdere checkins tegelijk gedaan, waarbij ook niet altijd alle issues vermeld werden. Dit zorgde ervoor dat de traceability-

informatie ook geen toegevoegde waarde meer had.

Ook binnen Logica is men zich bewust van de complexiteit van het onderhouden van traceability links. Een voorbeeld hiervan is een project waar men trachtte de (± 350) requirements te relateren aan (± 30) test cases in een traceability matrix. Hierdoor moesten er meer dan 10.000 potentiële relaties onderzocht worden. Uiteindelijk werd het gezien als teveel werk en is besloten om de matrix niet meer in te vullen.

Aan de huidige situatie binnen Logica is te zien dat er af en toe wel degelijk een zekere drang is naar het bereiken van volledige traceability, maar dat de juiste ondersteuning (van mensen, maar ook van bijvoorbeeld tooling) vaak ontbreekt. De algemene tendens is dat de voordelen van traceability zeer gewild zijn, maar dat het bijhouden ervan weinig tot geen extra moeite mag kosten.

Hoofdstuk 3

Onderzoeksformulering

In de komende paragrafen zal worden toegelicht welk doel het onderzoek heeft en welke vragen er worden beantwoord. Ook wordt in het kort toegelicht hoe het experiment wordt aangepakt. Een uitgebreidere aanpakbeschrijving is te vinden in hoofdstuk 4.

3.1 Doelstelling

Het doel van dit onderzoek is om te bekijken of vergelijkbare beschrijvingen van wijzigingen aan een softwaresysteem (in de vorm van commit logs in het versiebeheersysteem) leiden tot vergelijkbare aanpassingen aan de source code. Indien er een verband wordt aangetoond, houdt dit mogelijk in dat de kennis over eerder uitgevoerde wijzigingen hergebruikt kan worden bij het voorspellen van de impact van nieuwe wijzigingen, als er een tekstuele overeenkomst is tussen de beschrijvingen van de eerder uitgevoerde wijzigingen en de nieuwe wijziging.

De hoofdvraag die in dit onderzoek met behulp van een experiment wordt beantwoord is:

> *In hoeverre kan de kennis over de wijzigingen aan een softwaresysteem hergebruikt worden bij latere wijzigingen, door wijzigingen te relateren op basis van tekstuele overeenkomsten in de commit logs?*

3.2 Rationale

Uit de literatuurstudie en de interviews blijkt dat er behoefte is aan een geautomatiseerde traceability-aanpak. Dit vanwege het feit dat de moeite die nodig is om de traceability-links aan te leggen en te onderhouden wordt aangeduid als het grote probleem van traceability. Een geautomatiseerde aanpak zou dit probleem mogelijkerwijs sterk verkleinen.

Geautomatiseerde traceability is op verschillende manieren aan te pakken. Een voorbeeld hiervan is het automatisch linken van requirements aan source code door het traceren van de source code-footprint van test cases, die gekoppeld zijn aan de requirements (Egyed, 2001).

Na het analyseren van de beschikbare projectgegevens bij Logica en andere bedrijven, bleek dat er nauwelijks projecten te vinden waren waar traceability al werd toegepast, en de projecten waar dit wel het geval was, waren niet beschikbaar voor het uitvoeren van een experiment. Verder waren er op korte termijn geen projecten te vinden waar de requirements één op één te linken waren met test cases, waardoor de geautomatiseerde aanpak van Egyed niet gebruikt kon worden.

De keuze is uiteindelijk gevallen op een experiment waarbij gebruik wordt gemaakt van de wijzigingshistorie. Hier waren binnen Logica wel geschikte projecten voor te vinden. Bovendien konden door het kiezen van deze aanpak ook enkele open source-projecten meegenomen worden in het onderzoek.

3.3 Gerelateerde werken

Er zijn verscheidene onderzoeken geweest naar de toepassing van traceability-technieken voor het verbeteren van impact analyses.

Abbattista et al. (1994) en Bratthall et al. (2000) onderzochten de invloed van het gebruik van design-informatie op de effectiviteit van impact analyses, met wisselende, maar grotendeels positieve resultaten.

Von Knethen & Grund (2003) ontwierpen een traceability-model voor impact analyses. Ook Cleland-Huang et al. (2005) presenteren een model, welke vooral gericht is op niet-functionele requirements.

Ook zijn er onderzoeken gericht op volautomatische traceability. Antoniol et al. (2002) gebruiken Information Retrieval-technieken om traceability-relaties te vormen tussen source code en documentatie. Egyed (2001) creëert dynamisch traceability-relaties tussen requirements en source code door test cases, die gekoppeld zijn aan requirements, uit te voeren en de source code-footprint vast te leggen.

Betreffende het gebruik van de wijzigingshistorie van softwaresystemen zijn er eveneens een aantal onderzoeken uitgevoerd (Canfora & Cerulo, 2006; Chen et al., 2001; Ying, 2003; Zimmermann et al., 2004). Deze onderzoeken verschillen op een aantal punten met het in deze scriptie beschreven onderzoek:

Gebruikte dataset: De genoemde onderzoeken maken gebruik van open source-projecten. Dit onderzoek zal ook gebruik maken van enkele commerciële projecten.

Aanpak: Waar in de werken van Canfora & Cerulo (2006) en Ying (2003) ook teksten uit de bugtracker in het onderzoek worden meegenomen, zal dit onderzoek de bugtracker buiten beschouwing laten. Chen et al. (2001) hebben een gebruikerstest uitgevoerd onder 74 studenten. De gebruikers (ontwikkelaars) zullen niet bij dit onderzoek betrokken worden.

3.4 Kernproblemen

In het onderzoek komen enkele kernproblemen naar voren welke aandacht vereisen bij het opzetten van een experiment om de onderzoeksvraag te kunnen beantwoorden. Deze kernproblemen worden hieronder benoemd.

- *Het vinden van tekstuele overeenkomsten tussen twee wijzigingen.*

  Om goede relaties te kunnen vinden tussen de commit logs zal onderzocht moeten worden welke technieken hiervoor beschikbaar zijn en welke daarvan toepasbaar zijn in het experiment.

- *Het bepalen van de gelijkenis in code tussen twee wijzigingen.*

  Nadat er relaties zijn gevormd tussen wijzigingen op basis van een tekstuele overeenkomst, moet er beoordeeld worden in hoeverre deze relaties gebruikt kunnen worden om de impact van wijzigingen te voorspellen. Er moet een methodiek worden opgesteld waarmee bepaald kan worden in hoeverre twee wijzigingen 'gelijk' zijn.

3.5 Hypothesen

Voor het onderzoek zijn enkele hypothesen opgesteld, welke middels de kwantitatieve en kwalitatieve analyse (zie paragraaf 4.2 en 4.3 in het experiment gevalideerd zullen worden. Deze hypothesen worden hieronder beschreven.

Hypothese 1: *Naarmate een wijzigingshistorie groeit, zal de hoeveelheid nuttige informatie sneller groeien ten opzichte van de totale hoeveelheid informatie, en een grotere wijzigingshistorie zal dus beter presteren dan een kleinere.*

Als een softwaresysteem evolueert zal de wijzigingshistorie groeien. Het zal aan de ene kant lastiger worden om deze 'informatiebron' (= een verzameling wijzigingen) te doorzoeken, maar aan de andere kant is de verwachting dat de diversiteit in de informatie dusdanig stijgt, dat de informatie relatief steeds nuttiger wordt. Deze hypothese wordt gevalideerd door het analyseren van projecten van verschillende grootten in het experiment. Ook zal worden geanalyseerd of de mate waarin kennis uit eerdere wijzigingen hergebruikt kan worden stijgt gedurende het verloop van elk individueel project.

Hypothese 2: *Open Source projecten zullen relatief beter presteren dan commerciële projecten vanwege kwalitatief betere commit logs.*

Open source projecten hechten over het algemeen veel waarde aan duidelijke commit logs, omdat de ontwikkelaars gedistribueerd werken (Gutwin et al., 2004). Ook worden in open source projecten over het algemeen bijna geen ongerelateerde wijzigingen gegroepeerd tot een enkele commit. Uit gesprekken met ontwikkelaars bij Logica blijkt dat deadlines vaak tot gevolg hebben dat er minder aandacht wordt besteed aan duidelijke commit logs en dat wijzigingen worden gegroepeerd tot een enkele commit. Om deze hypothese te valideren zullen in het experiment een aantal open source-projecten kwantitatief worden vergeleken met commerciële projecten, gevolgd door een kwalitatieve analyse waarin de kwaliteit van de commit logs wordt vergeleken.

Hoofdstuk 6 zal het resultaat van de validatie van bovenstaande hypothesen beschrijven.

3.6 Scope

Binnen vier projecten (twee open source- en twee commerciële projecten) wordt één specifieke implementatie van een traceability-techniek voor het verbeteren van change impact analyses getoetst, welke gebruik maakt van de wijzigingshistorie van de projecten.

Het verslag zal voldoende informatie bieden om als basis te dienen voor een eventuele tool.

Er zal ook geen onderzoek worden gedaan naar de manier waarop traceability-informatie op een toegankelijke manier gepresenteerd (en gevisuali-

seerd) kan worden aan potentiële gebruikers (ontwikkelaars).

3.7 Aanpak

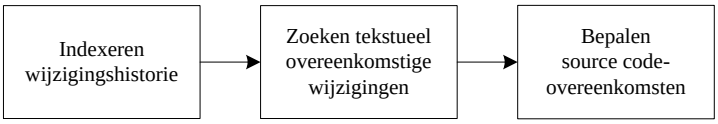
Figuur 3.1 geeft schematisch de aanpak weer die in het experiment gebruikt wordt. Van twee open source projecten en twee commerciële projecten wordt de wijzigingshistorie geanalyseerd.

In elk project wordt gezocht naar wijzigingen waarvan de commit logs tekstuele overeenkomsten met elkaar hebben. Bij elke wijziging met een tekstuele relatie wordt vervolgens gemeten in hoeverre de wijzigingen overeenkomen met betrekking tot de source code. In hoofdstuk 4 wordt de aanpak verder uitgewerkt.

3.8 Gebruikte projectdata

Voor het experiment zijn vier projecten gebruikt. De gegevens over deze projecten zijn vermeld in tabel 3.1 op pagina 16. De namen van de Logica-projecten zijn geanonimiseerd. De Lines of Code en de gebruikte programmeertalen zijn bepaald met behulp van 'SLOCCount' van David A. Wheeler[1].

Door de geringe hoeveelheid beschikbare projectdata bij Logica was het niet mogelijk om een keuze te maken uit meerdere projecten. Project X en Project Y zijn dus puur gekozen vanwege de beschikbaarheid ervan. Het Trac-project is door een expert uit de open source-wereld aangeraden. Dit project werd geprezen om de duidelijke commit logs en de taakgerichte manier van werken van de ontwikkelaars. Het Mozilla-project is gekozen om het feit dat het relatief een groot project is, en dus mogelijkerwijs goed gebruikt kan worden om de invloed van de grootte van de wijzigingshistorie te analyseren.



Figuur 3.1: Aanpak van het experiment

[1] `http://www.dwheeler.com/sloccount/`

Project:	Project X
Type:	Logica-project
Lines of Code:	22.930
Aantal revisies:	1368
Talen:	Java (93,22%), sh (6,78%)
Projectduur:	4 jaar, 6 maanden

(a) Project X

Project:	Project Y
Type:	Logica-project
Lines of Code:	263.686
Aantal revisies:	36.257
Talen:	Java (96,62%), sh (3,21%), Python (0,11%), Awk (0,06%)
Projectduur:	5 jaar, 4 maanden

(b) Project Y

Project:	Mozilla
Type:	Open source-project
Lines of Code:	3.723.079
Aantal revisies:	251.492
Talen:	C++ (50,05%), ANSI C (32,40%), Java (6,04%), Perl (5,15%), overig (6,36%)
Projectduur:	10 jaar, 3 maanden

(c) Mozilla

Project:	Trac
Type:	Open source-project
Lines of Code:	36.945
Aantal revisies:	7452
Talen:	Python (99,09%), overig (0,91%)
Projectduur:	4 jaar, 11 maanden

(d) Trac

Tabel 3.1: Gegevens over de vier gebruikte softwareprojecten

3.9 Kwaliteit

De kwaliteit van het uitgevoerde experiment wordt hier besproken op basis van de betrouwbaarheid, validiteit en generaliseerbaarheid. Er zullen enkele punten benoemd worden die betrekking hebben op de manier waarop het experiment is uitgevoerd. Details betreffende deze aanpak van het experiment zullen gegeven worden in hoofdstuk 4.

3.9.1 Betrouwbaarheid en validiteit

Om de gebruikte metrieken zo betrouwbaar mogelijk te houden, worden deze gebaseerd op algemeen geaccepteerde metrieken voor de evaluatie van Information Retrieval-technieken (Järvelin & Kekäläinen, 2002; Manning et al., 2008; van Rijsbergen, 1979; Sakai, 2007).

Het experiment gebruikt bestaande commits in het versiebeheersysteem om de mate waarin de impact van de wijzigingen voorspeld kon worden te toetsen. Dit kan zowel voordelig als nadelig zijn. Het voordeel is dat de wijzigingen reeds uitgevoerd zijn, en er dus vanuit gegaan mag worden dat deze wijzigingen grotendeels kloppen. Het nadeel is dat er een kans aanwezig is dat er enkele slechte wijzigingen tussen zullen zitten. Een commit in het versiebeheersysteem is niet altijd gelijk aan een wijziging. Het is bijvoorbeeld mogelijk dat meerdere wijzigingen in een enkele commit worden gegroepeerd. Andersom is het ook mogelijk dat een wijziging slechts gedeeltelijk verwerkt is in een commit, omdat er bestanden vergeten zijn. Omdat er geen gegevens beschikbaar zijn over de regelmaat waarin dit voorkomt, kan niet gegarandeerd worden dat deze aanpak volledig betrouwbaar is.

In het kwalitatieve onderzoek zullen slechts totaal twintig wijzigingen per project geanalyseerd. Er is niet onderzocht in hoeverre dit aantal statistisch significant is voor de totale groep wijzigingen in de projecten.

3.9.2 Generaliseerbaarheid

Er zijn een aantal duidelijke overeenkomsten en verschillen tussen de projecten, die de mate waarin de projecten generaliseerbaar zijn beïnvloedt.

Betreffende de gebruikte programmeertalen is er een redelijke diversiteit. De drie meest voorkomende talen zijn Java, C++ en Python. Ondanks de diversiteit betreft het hier wel drie objectgeörienteerde talen.

Een andere variërende eigenschap is de grootte van de projecten. Het aantal Lines of Code verschilt significant per project.

Nadelig aan de Logica-projecten is dat deze allen van dezelfde afdeling vandaan komen. Gezien het geringe aantal beschikbare projecten voor het onderzoek was dit onvermijdelijk. Het gevolg hiervan is dat de twee projecten gedeeltelijk dezelfde ontwikkelaars delen. Gedeeltelijk omdat er binnen de projecten wel degelijk sprake was van verloop onder de ontwikkelaars. Ook komen de projecten uit hetzelfde businessdomein komen.

Aan de ene kant zijn er dus een aantal diverse eigenschappen aanwezig in de gekozen selectie projecten, zoals de grootte en de gebruikte programmeertalen, die positief zijn voor de generaliseerbaarheid. Vooral op het gebied

van de Logica-projecten zijn er echter enkele eigenschappen (ontwikkelaars, businessdomein) die ervoor zorgen dat deze projecten niet representatief zijn voor andere Logica-projecten en dus ook niet, nog algemener, voor andere commerciële projecten.

Hoofdstuk 4

Onderzoeksaanpak

Dit hoofdstuk beschrijft uitgebreid de aanpak van het experiment. Eerst zal in paragraaf 4.1 worden toegelicht op welke manier de resultaten verzameld worden. Vervolgens wordt in paragraaf 4.2 en paragraaf 4.3 toegelicht hoe in respectievelijk de kwantitatieve en kwalitatieve analyse naar de resultaten gekeken wordt.

In het experiment is voor elk van de vier projecten (paragraaf 3.8) dezelfde aanpak gebruikt. Deze aanpak zal in dit hoofdstuk worden toegelicht.

4.1 Meetmethode

De manier waarop de resultaten per project verzameld worden, is op te splitsen in drie onderdelen, zoals het is weergegeven in figuur 3.1. Eerst worden de tekstuele beschrijvingen geïndexeerd uit de wijzigingshistorie (4.1.1). Vervolgens wordt er gezocht naar relaties tussen deze tekstuele beschrijvingen (4.1.2). Ten slotte wordt er bepaald of er overeenkomsten in de code aanwezig zijn tussen wijzigingen met een tekstuele overeenkomst.

Voor het indexeren en doorzoeken van de commit logs uit de Subversion versiebeheersystemen is vanwege tijdsbeperkingen besloten om een bestaande tool te gebruiken. Er is een korte vergelijking gemaakt tussen drie tools: Lucene[1], ReqSimile[2] en Terrier[3]. De keuze is gemaakt voor Lucene, een open source zoekmachine van Apache, hoofdzakelijk vanwege de automatiseringsmogelijkheden en de grote hoeveelheid beschikbare documentatie.

[1] `http://lucene.apache.org/java/`
[2] `http://reqsimile.sourceforge.net/`
[3] `http://ir.dcs.gla.ac.uk/terrier/`

4.1.1 Wijzigingshistorie indexeren

De eerste stap is om de commit logs uit het versiebeheersysteem te indexeren. Hier wordt elke commit log opgesplitst in een lijst van woorden. Deze woordenlijsten worden gebruikt om gegevens te verzamelen over de frequentie en locatie van woorden in commit logs en in de totale wijzigingshistorie. Deze gegevens zullen later gebruikt worden bij het bepalen van de tekstuele relaties.

Voor het indexeren van de commit logs is een tool ontwikkeld. Deze tool verzamelt automatisch alle commit logs van een project uit een versiebeheersysteem en creëert een Lucene index. Bij het indexeren worden bovendien twee technieken gebruikt om de kwaliteit van de index te verbeteren. Deze worden hieronder besproken.

Stopwoordenlijst

Omdat veelvoorkomende woorden als 'bug' en 'commit' ook geïndexeerd zullen worden, wordt er gebruik gemaakt van een *stopwoordenlijst*. Deze stopwoordenlijst (te vinden in bijlage C) is samengesteld door van de vier projecten de top 50 van de meest voorkomende woorden te doorlopen. De lijsten van generieke woorden uit alle projecten zijn samengevoegd tot een enkele stopwoordenlijst die uiteindelijk voor elk project gebruikt is.

Het gebruiken van een stopwoordenlijst zal het aantal valse positieven drastisch omlaag brengen, omdat er op basis van deze woorden geen relaties meer gelegd zullen worden.

Stemming

Er wordt gebruik gemaakt van een stemming-algoritme (Snowball[4]) om Engelse woorden (de commit logs in de gebruikte projecten zijn in de Engelse taal) terug te brengen naar de 'stam'. Dit houdt in dat woorden terug worden gebracht naar een basisvorm.

Dit wordt gedaan om ervoor te zorgen dat bijvoorbeeld vervoegingen of meervoudsvormen van woorden aan elkaar gerelateerd kunnen worden. Een voorbeeld is het werkwoord 'handle'. Deze zou in de commit logs ook kunnen voorkomen als 'handling' of 'handler'. In elke situatie zal het teruggebracht worden naar 'handl', waardoor deze woorden aan elkaar gerelateerd kunnen worden.

[4] `http://snowball.tartarus.org/`

4.1.2 Tekstuele overeenkomsten zoeken

Voor elke revisie wordt een zoekopdracht uitgevoerd waarbij er wordt gezocht naar tekstueel overeenkomstige commit logs, die *eerder* plaatsvonden (er wordt dus alleen terug in de tijd gezocht). In figuur 4.1 is dit schematisch weergegeven. Op de figuur is te zien dat bijvoorbeeld wijziging 3 een tekstuele relatie (r) heeft met wijziging 1. Het getal achter de r geeft aan dat er een rangschikking in de relaties zit. Zo heeft wijziging 6 relaties van aflopende sterkte met wijziging 4, 2 en 1.

Een alternatief zou zijn om in samenwerking met de originele ontwikkelaars van het project een aantal hypothetische wijzigingen op te stellen. Dit is om verschillende redenen niet gedaan. Ten eerste is het een tijdrovendere aanpak, waarbij veel tijd gevraagd wordt van de ontwikkelaars. Ten tweede is het probleem aanwezig dat de impact van de wijzigingen handmatig gevalideerd moeten worden, waarbij alsnog fouten gemaakt kunnen worden.

Het voordeel van de gekozen aanpak is dat er meer wijzigingsgevallen gebruikt worden en dat deze in principe al gevalideerd zijn, omdat ze reeds uitgevoerd zijn.

Lucene maakt gebruik van een aantal Information Retrieval-technieken voor het berekenen van 'relevantiescores' tussen commit logs (Manning et al., 2008; van Rijsbergen, 1979). Als voor een commit log A bepaald moet worden of commit log B relevant is, wordt er gekeken naar:

- De hoeveelheid woorden uit A die in B voorkomen. Hoe meer woorden overeenkomen, hoe sterker de relatie is.
- Het aantal keer dat woorden uit A in B voorkomen (*Term Frequency*). Een hogere frequentie geeft een sterkere relatie aan.

Wijzigingsnr. → / **↓**	**1**	**2**	**3**	**4**	**5**	**6**	**... n**
1	x						
2		x					
3	r_1		x				
4		r_1	r_2	x			
5					x		
6	r_3	r_2		r_1		x	
⋮							
n							

Tabel 4.1: Schematische weergave van het tekstueel relateren van wijzigingen

- Het aantal keer dat woorden uit A en B in de gehele wijzigingshistorie voorkomen (*Inverse Document Frequency*). Woorden met een lagere frequentie in de gehele wijzigingshistorie zijn zeldzamer, dus een relatie op basis van een dergelijk woord is sterker dan een relatie op basis van een veelvoorkomend woord.

Het zoeken van de tekstuele overeenkomsten wordt, net als het indexeren, automatisch gedaan met de eerder genoemde tool. De tool creëert voor elke onderzochte commit log een lijst van relevante commit logs, gesorteerd op relevantiescore.

4.1.3 Code-overeenkomsten bepalen

Elke wijziging wordt vergeleken op source code-overeenkomsten met de top 20 van de zoekresultaten (de lijst tekstueel relevante wijzigingen). Het instellen van een grenswaarde op het aantal vergelijkingen heeft twee redenen. Ten eerste vanwege de performance. Voor het ophalen van de benodigde informatie uit de SVN-repositories zijn vele SVN-queries nodig. Dit kost veel tijd en bandbreedte, vooral in het geval van de open source-projecten, waarbij de server zich niet in het lokale netwerk bevindt. De tweede reden is dat mensen die lijsten van zoekresultaten doorzoeken niet verder willen kijken dan de eerste paar pagina's. Een effectief zoeksysteem zorgt er dus voor dat er zo min mogelijk zoekresultaten nodig zijn (Brin & Page, 1998).

Een source code-overeenkomst tussen twee wijzigingen kan meerdere dingen inhouden. Het zou bijvoorbeeld kunnen inhouden dat de wijzigingen op dezelfde *locatie* in de source code zijn geïmplementeerd. Ook zou het kunnen betekenen dat de wijzigingen op een vergelijkbare *manier* geïmplementeerd zijn.

In dit onderzoek wordt vanwege tijdsbeperkingen slechts gekeken naar de overeenkomsten in de locatie van wijzigingen. Dit wordt gedaan op bestandsniveau. Er wordt geen rekening gehouden met de verplaatsing van bestanden gedurende de evolutie van de software. Hierdoor zullen vooral op de lange termijn de resultaten lager uitvallen. Indien een hoger detailniveau gekozen zou worden, bijvoorbeeld op het niveau van procedures of regelnummers, zou deze veranderlijkheid nog zwaarder meetellen. Om het effect van de veranderlijkheid zo laag mogelijk te houden, en anderzijds toch het detailniveau dusdanig hoog te houden zodat het nut behouden blijft, is gekozen voor het bestandsniveau.

Elke wijziging heeft een lijst met bestanden. Hierin zitten bestanden die nieuw zijn sinds de wijziging, en reeds bestaande bestanden die zijn aangepast. Om code-overeenkomsten te bepalen tussen twee wijzigingen wordt

gebruik gemaakt van twee metrieken: *precision* en *recall*. Deze worden hieronder toegelicht.

Precision en recall

Een toekomstige wijziging (w) waarvoor voorspeld moet worden welke bestanden (C_w) uiteindelijk aangepast moeten worden, zal op basis van tekstuele overeenkomsten een lijst van relevante wijzigingen (R) hebben (de zoekresultaten). Elke relevante wijziging r in R zal ook een lijst bestanden (C_r) bevatten welke in die betreffende wijziging zijn aangepast.

De precision (P) van wijziging r ten opzichte van wijziging w wordt gedefinieerd als het percentage bestanden dat zich zowel in C_w als in C_r bevindt, ten opzichte van het totaal aantal bestanden in C_r (zie vergelijking 4.1). Met andere woorden: hoeveel procent van het aantal bestanden in de relevante wijziging zal in de toekomstige wijziging ook aangepast moeten worden?

$$P = \frac{|C_w \bigcap C_r|}{|C_r|} \tag{4.1}$$

De recall (R) van wijziging r ten opzichte van wijziging w wordt gedefinieerd als het percentage bestanden dat zich zowel in C_w als in C_r bevindt, ten opzichte van het totaal aantal bestanden in C_w (zie vergelijking 4.2). Met andere woorden: hoeveel procent van het aantal bestanden dat in de toekomstige wijziging aangepast zal moeten worden is terug te vinden in de relevante wijziging?

$$R = \frac{|C_w \bigcap C_r|}{|C_w|} \tag{4.2}$$

Gecombineerde precision en recall

Om de precision- en recall-score na een x aantal zoekresultaten te bepalen (ook wel gedefinieerd als de precision en recall op 'rang x'), zijn er ook gecombineerde precision- en recall-definities opgesteld. Deze definities zijn gebaseerd op de *cumulated gain-based (CG) measurements*, uitgelegd door Järvelin & Kekäläinen (2002). De daar gedefinieerde gain betreft hier dus een precision- of recall-score. De cumulated gain op rang x is de som van de scores van zoekresultaat 1 tot x. Deze som kan hier niet direct gebruikt worden. Als bijvoorbeeld zoekresultaat r_1 een bepaald bestand bevat, en vervolgens blijkt dat r_2 dit bestand ook bevat, zal de recall en precision namelijk niet stijgen of dalen, omdat het bestand al eerder is voorgekomen in de zoekresultaten (in r_1).

De precision op rang x ($P@x$) wordt gedefinieerd als het percentage unieke bestanden dat zich zowel in C_w als in $C_{r,1}$ tot en met $C_{r,x}$ bevindt, ten opzichte van het totaal aantal unieke bestanden in $C_{r,1}$ tot en met $C_{r,x}$ (zie vergelijking 4.3). Met andere woorden: hoeveel procent van het aantal unieke bestanden in de zoekresultaten tot en met rang x zal in de toekomstige wijziging ook aangepast moeten worden?

$$P@x = \frac{|C_w \bigcap (\bigcup_{i=1}^{x} C_{r,i})|}{|\bigcup_{i=1}^{x} C_{r,i}|} \tag{4.3}$$

De recall op rang x ($R@x$) wordt gedefinieerd als het percentage unieke bestanden dat zich zowel in C_w als in $C_{r,1}$ tot en met $C_{r,x}$ bevindt, ten opzichte van het totaal aantal bestanden in C_w (zie vergelijking 4.4). Met andere woorden: hoeveel procent van het aantal bestanden dat in de toekomstige wijziging aangepast zal moeten worden is terug te vinden in de zoekresultaten tot en met rang x?

$$R@x = \frac{|C_w \bigcap (\bigcup_{i=1}^{x} C_{r,i})|}{|C_w|} \tag{4.4}$$

Zowel de precision als de recall zijn van belang voor een goed resultaat. Een lage precision-score zou inhouden dat slechts een klein deel van de aangepaste bestanden van de relevante wijziging ook uiteindelijk voor de toekomstige wijziging van belang zijn. Een lage recall-score betekent dat niet alle bestanden die aangepast moeten worden terug te vinden zijn in de zoekresultaten.

De bepaling van het verschil in belang tussen precision en recall is afhankelijk van het uiteindelijke doel. Een ontwikkelaar met weinig kennis over een systeem wil zoveel mogelijk nuttige resultaten (hoge recall), want hij wil niets over het hoofd zien. Een niet al te hoge precision maakt in dat geval niet uit, omdat het gedeelte van het systeem dat hij zal moeten doorzoeken reeds beperkt is tot de zoekresultaten van de tool. Een ontwikkelaar met veel kennis over een systeem zal een groot aantal bestanden al zelf kunnen voorspellen, maar kan iets over het hoofd zien. Voor hem zou een systeem met een lagere recall mogelijk al afdoende zijn (Hassan & Holt, 2006). Omdat dit onderzoek zich voornamelijk richt op ontwikkelaars met relatief weinig kennis over een systeem, zal er meer waarde worden gehecht aan de recall-scores.

Ook het bepalen van de code-overeenkomsten wordt automatisch uitgevoerd met een ontwikkelde tool, welke door middel van Subversion-queries de lijsten van aangepaste bestanden opvraagt en de precision- en recall-scores berekent.

4.2 Kwantitatieve analyse

Voor de kwantitatieve analyse wordt hoofdzakelijk bekeken van hoeveel wijzigingen de impact voorspeld kon worden op basis van eerder uitgevoerde tekstueel overeenkomstige wijzigingen. Hierbij wordt ook bepaald of het aantal geanalyseerde zoekresultaten (5, 10, 15 of 20) van invloed is op de effectiviteit van het voorspellen.

Daarnaast wordt bekeken of met behulp van de resultaten van het experiment gezegd kan worden wat de invloed is van het aantal uitgevoerde wijzigingen (de grootte van de wijzigingshistorie) op de mate waarin de impact van wijzigingen voorspeld kan worden.

4.3 Kwalitatieve analyse

De kwalitatieve analyse vindt plaats met het doel de betrouwbaarheid van de kwantitatieve analyse te beoordelen, en om te bepalen of de resultaten afhankelijk zijn van bepaalde factoren, zoals het type wijziging, of het soort tekst in de commit logs. Ook wordt getracht de tweede hypothese te valideren (*Open Source projecten zullen relatief beter presteren dan commerciële projecten vanwege kwalitatief betere commit logs.*).

Er zullen enkele specifieke 'goede' en 'slechte' gevallen uit de experimentsresultaten nader bekeken worden. De categorie 'goede' gevallen bestaat uit gevallen met een recall-score van 100%, en de categorie 'slechte' gevallen bestaat uit gevallen met een recall-score van 0%. De bedoeling hiervan is om door middel van het analyseren van 'extreem goede' en 'extreem slechte' gevallen een beter beeld te krijgen van de bepalende factoren in het experiment die het meeste invloed hebben op de scores.

Voor elke categorie worden er 10 gevallen *willekeurig* uitgekozen per project en voor elk geval zullen de volgende vragen beantwoord worden:

- **Om wat voor type wijziging gaat het?**

  Met deze vraag wordt bekeken of de onderzochte methode misschien beter geschikt is voor slechts een bepaald type wijziging, die beter te voorspellen blijkt dan andere typen. Voor de classificering van de typen wijzigingen wordt gebruik gemaakt van de door Chapin et al. (2001) gedefinieerde typen (enhancive, corrective, reductive, adaptive, performance, preventive en groomative).

- **Wat wordt er beschreven in de commit logs van de wijziging en zijn tekstueel overeenkomstige wijzigingen?**

Er wordt gekeken wat de reden is dat de wijzigingen tekstueel overeenkomen, om te bepalen of het gaat om een 'nuttig' geval. Als de wijzigingen namelijk tekstueel overeenkomen omdat ze allen een bepaalde algemene tekst bevatten (bijvoorbeeld: "Fixed a bug"), betreft het een geval dat de onderzoeksresultaten mogelijk omlaag zou kunnen brengen, maar niet van belang zou zijn als deze methode van impact analyse in de praktijk gebruikt zou worden.

- **Wat is de afstand tussen de wijziging en zijn tekstueel overeenkomende wijzigingen?**

  Hierbij wordt gekeken of de 'afstand' tussen de wijzigingen van invloed is op het resultaat. Met afstand wordt hier het aantal wijzigingen bedoeld dat tussen de vergeleken wijzigingen zit. Het is bijvoorbeeld een mogelijkheid dat de goede gevallen alleen bestaan uit gevallen waarin de afstand tussen de wijzigingen klein is. Dat houdt in dat het wellicht een goede keuze is om een grenswaarde voor de wijzigingshistorie die doorzocht wordt in te stellen. Als de afstanden daarentegen juist altijd groot zijn, zou het zoeksysteem daar ook op ingericht kunnen worden. Voor deze vraag wordt een tweedeling gemaakt in de gemiddelde afstand tot de wijzigingen die geen relevante bestanden bevatten en de gemiddelde afstand tot de wijzigingen die wel relevante bestanden bevatten.

Om in de categorie 'goede' gevallen een beter beeld te krijgen van het daadwerkelijke nut van de zoekresultaten, zullen ook deze vragen beantwoord worden:

- **Komen in de zoekresultaten bepaalde bestanden vaker voor, en zijn deze bestanden relevant voor de onderzochte wijziging?**

  Het is mogelijk interessant om te bekijken of het loont om bestanden die vaker in de zoekresultaten terug komen een hogere 'betrouwbaarheid' te geven. Daarom wordt er, indien bepaalde bestanden meerdere keren voorkomen in de zoekresultaten, bekeken of deze bestanden dan ook eerder relevant zijn voor de onderzochte wijziging.

- **Hoe verhoudt het aantal bestanden in de zoekresultaten zich tot het totaal aantal bestanden in de betreffende revisie?**

  Om een beter beeld te krijgen bij de precision-scores uit de kwantitatieve analyse, wordt deze precision vergeleken met wat de precision zou zijn als alle bestanden van de gehele revisie doorzocht zou moeten worden naar de bestanden waar een wijziging impact op heeft.

Op basis van de antwoorden op bovenstaande vragen wordt bekeken hoeveel waarde gehecht kan worden aan de kwantitatieve onderzoeksdata. Verder wordt bekeken wat de waarde is van de gekozen projecten voor het experiment en op welke manieren het experiment mogelijk verbeterd zou kunnen worden.

Hoofdstuk 5

Resultaten en analyse

Dit hoofdstuk presenteert de resultaten van het experiment. Ook worden de resultaten geanalyseerd. In paragraaf 5.1 zullen eerst de resultaten van het kwantitatieve onderzoek gegeven worden, gevolgd door de resultaten van het kwalitatieve onderzoek in paragraaf 5.2.

5.1 Kwantitatief onderzoek

Het doel van de kwantitatieve analyse is om de hoofdvraag deels te beantwoorden en om de eerste hypothese te valideren. De hoofdvraag luidt:

> *In hoeverre kan de kennis over de wijzigingen aan een softwaresysteem hergebruikt worden bij latere wijzigingen, door wijzigingen te relateren op basis van tekstuele overeenkomsten in de commit logs?*

En de eerste hypothese luidt:

> *Naarmate een wijzigingshistorie groeit, zal de hoeveelheid nuttige informatie sneller groeien ten opzichte van de totale hoeveelheid informatie, en een grotere wijzigingshistorie zal dus beter presteren dan een kleinere.*

Het nut van de zoekresultaten

Uit het kwantitatieve onderzoek blijkt dat van een beperkt deel van de wijzigingen in de vier onderzochte projecten deels of volledig bepaald kon

worden welke bestanden aangepast moesten worden. De volgende paragrafen lichten dit toe op basis van de recall- en precision-scores.

Recall-scores

Tabel 5.1 geeft een overzicht van de behaalde recall-scores na maximaal 20 zoekresultaten.

In de tabel is te zien dat de resultaten enigszins verschillen per project. Op het gebied van recall behaalt Trac de hoogste score, aangezien van iets meer dan 11 procent van alle wijzigingen deels of volledig de impact voorspeld kon worden, en ruim 5 procent zelfs (bijna) volledig voorspeld werd.

Daarentegen werden in Project Y slechtere resultaten behaald. Minder dan 1 procent van de wijzigingen is in ieder geval deels voorspeld, waarbij twee derde daarvan (bijna) volledig voorspeld is.

Een reden dat de recall-scores in de hoogste categorie (75-100%) hoog liggen ten opzichte van de overige categorieën, is dat er relatief veel wijzigingen zijn waarin slechts 1 bestand aangepast wordt, waardoor hier redelijk eenvoudig een 100% (of 0%) recall-score te behalen valt. Tabel 5.2 geeft een overzicht van het percentage wijzigingen met een recall-score van 100% dat slechts om een enkel bestand gaat, ten opzichte van alle wijzigingen met 100% recall.

De recall-scores geven aan dat van een beperkt deel van de wijzigingen inderdaad de impact goed voorspeld kon worden. Ook zijn er duidelijke verschillen tussen de projecten te zien, waarbij er geen tweedeling te maken is tussen de open source-projecten en de commerciële projecten. Wel is te zien dat het Trac-project relatief hoog scoort, en Project Y relatief laag. Wat de oorzaak hiervan is, is een interessante kwestie welke door middel van de kwalitatieve analyse beantwoord dient te worden (paragraaf 5.2).

	Recall-score:				
Project	**0%**	**>0-25%**	**25-50%**	**50-75%**	**75-100%**
Project X	95,54%	0,37%	0,44%	1,02%	2,63%
Project Y	99,32%	0,08%	0,09%	0,08%	0,42%
Mozilla	96,37%	0,50%	0,51%	0,68%	1,94%
Trac	88,90%	0,95%	1,68%	2,75%	5,72%

Tabel 5.1: Het percentage wijzigingen met een bepaalde recall-score ten opzichte van het totaal aantal wijzigingen

Project	Percentage
Project X	47,06%
Project Y	44,26%
Mozilla	62,90%
Trac	69,34%

Tabel 5.2: Het percentage 1-bestands-wijzigingen in de wijzigingen met 100% recall

Precision-scores

De precision-scores zijn gegeven in tabel 5.3. De percentages bij een precision-score van 0% komen overeen met de 0%-recall-score percentages uit tabel 5.1, omdat het dezelfde collectie wijzigingen betreft (met een precision-score van 0% is het immers niet mogelijk om een recall-score van meer dan 0% te halen, en hetzelfde geldt andersom).

Het grootste deel van de wijzigingen behaalt een precision-score van onder de 25%. Echter, zoals in paragraaf 4.1.3 is besproken, hoeven deze scores niet hoog te zijn om voordeel te bieden in het werk van de ontwikkelaar. De vraag in hoeverre dit goede precision-scores zijn, zal beantwoord worden in de kwalitatieve analyse, waarbij onder andere gekeken wordt naar de grootte van de systemen (in de vorm van het aantal bestanden).

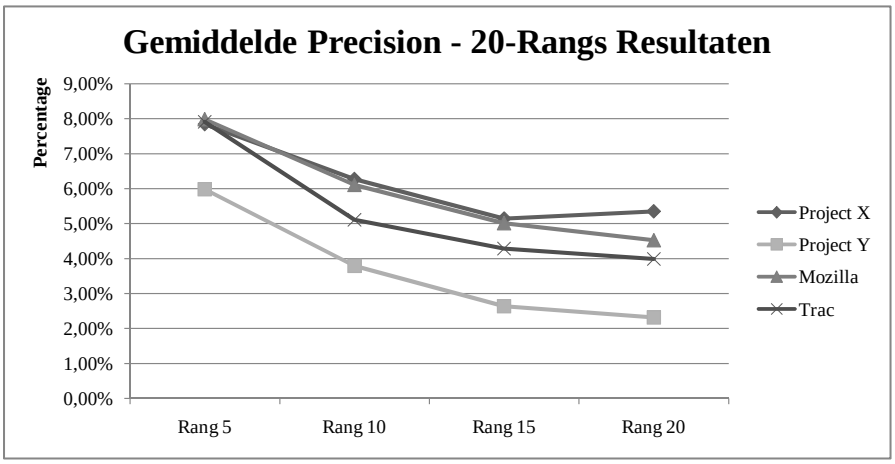
Invloed van het aantal zoekresultaten

Een interessante vraag is of het wellicht loont om een maximum te stellen aan het aantal zoekresultaten. Om dit te onderzoeken, is van de zoekacties met meer dan 15 zoekresultaten en een precision- en recall-score van meer dan 0% een overzicht gemaakt.

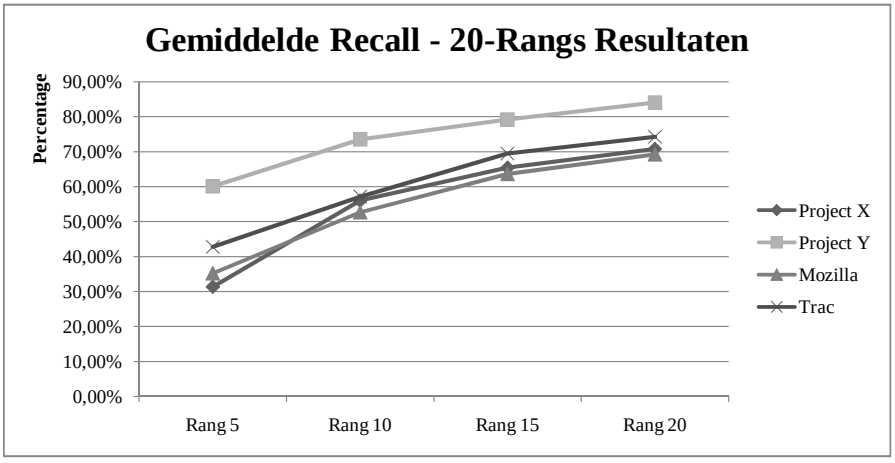
In figuur 5.1 en figuur 5.2 zijn de gemiddelde precision en recall te zien na 5, 10, 15 en 20 zoekresultaten (rang 5, 10, 15 en 20, respectievelijk). Het is te zien dat de gemiddelde precision tussen rang 5 en rang 20 minder dan 5% daalt, en de recall maximaal rond de 40% stijgt.

	Precision-score:				
Project	**0%**	**>0-25%**	**25-50%**	**50-75%**	**75-100%**
Project X	95,54%	4,09%	0,22%	0,15%	0,00%
Project Y	99,32%	0,60%	0,04%	0,01%	0,02%
Mozilla	96,37%	3,20%	0,28%	0,09%	0,06%
Trac	88,90%	9,22%	1,18%	0,46%	0,24%

Tabel 5.3: Het percentage wijzigingen met een bepaalde precision-score ten opzichte van het totaal aantal wijzigingen



Figuur 5.1: Gemiddelde precision-score van de 20-rangs resultaten



Figuur 5.2: Gemiddelde recall-score van de 20-rangs resultaten

Het instellen van een grenswaarde voor het aantal resultaten lijkt op basis van deze gegevens geen verbetering op te leveren. Dit zou namelijk een forse daling van de recall-scores op kunnen leveren, tegenover een lichte stijging in de precision-scores. De recall-score is echter, zoals in paragraaf 4.1.3 is besproken, van groter belang dan de precision-score.

Invloed van de grootte van de wijzigingshistorie

Hypothese 1 uit paragraaf 3.5 betrof het effect op de groei van de hoeveelheid nuttige informatie naarmate de wijzigingshistorie groeit. Naar aanleiding hiervan is geanalyseerd hoe de precision- en recall-scores zich verhouden gedurende de loop van het project, en op welk moment in het verloop van

het project relatief de meeste wijzigingen voorspeld kunnen worden. In bijlage B zijn de grafieken opgenomen die de resultaten van deze analyse weergeven voor alle projecten. Figuur 5.3 en 5.4 tonen deze grafieken voor het Mozilla-project.

Figuur B.1 tot en met B.4 tonen op welke momenten gedurende het projectverloop de meeste wijzigingen (enigszins) voorspeld kunnen worden. Zo is te zien dat bij Project X de score vooral later in het project begint te stijgen (met enkele dalingen tussendoor). Bij Project Y is opmerkelijk dat gedurende een groot gedeelte van het project er totaal geen wijzigingen voorspeld konden worden (dit wordt in de kwalitatieve analyse nader onderzocht). Bij het Mozilla- en het Trac-project is het aantal voorspelde wijzigingen echter vrijwel stabiel gedurende het volledige verloop van het project.

Figuur B.5 tot en met B.8 tonen de gemiddelde precision- en recall-scores in de loop van de projecten. Een trend in alle vier de projecten is dat de gemiddelde precision-score na een piek in het begin van de projecten gestaag daalt.

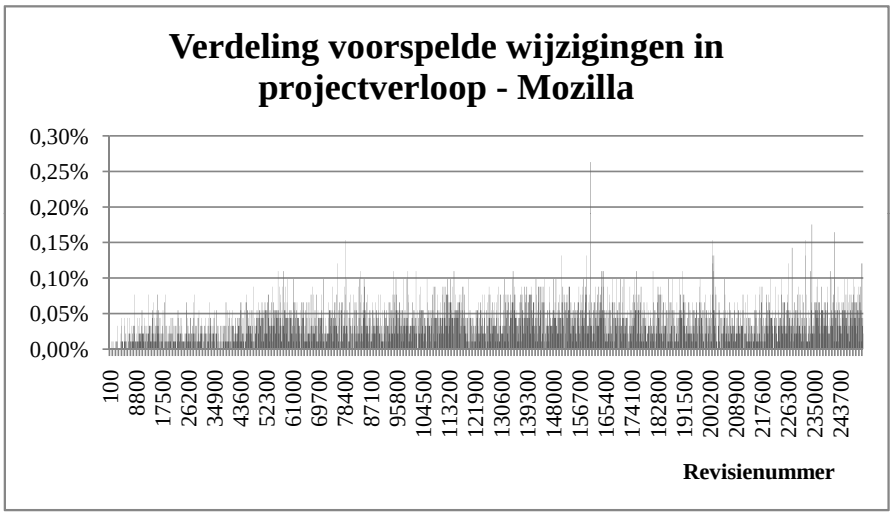
Opmerkelijk aan de gemiddelde recall-scores is dat deze in alle projecten hoog begint. Dit zou verklaard kunnen worden door het feit dat projecten in de beginfase relatief veel kleiner zijn dan in latere fases. De kans dat wijzigingen dan in dezelfde bestanden plaatsvinden is dan ook groter.

Afgezien van het Mozilla-project volgt er in elk project een daling, gevolgd door een langzame stijging. In het Mozilla-project is juist in het begin een stijging te zien, waarna een geleidelijke daling volgt tot een constante score van 70%. Het Mozilla-project is relatief veel groter dan de overige projecten. Het zou dus mogelijk zijn dat de score in de overige projecten na verloop van tijd, net als het Mozilla-project, steeds constanter worden.

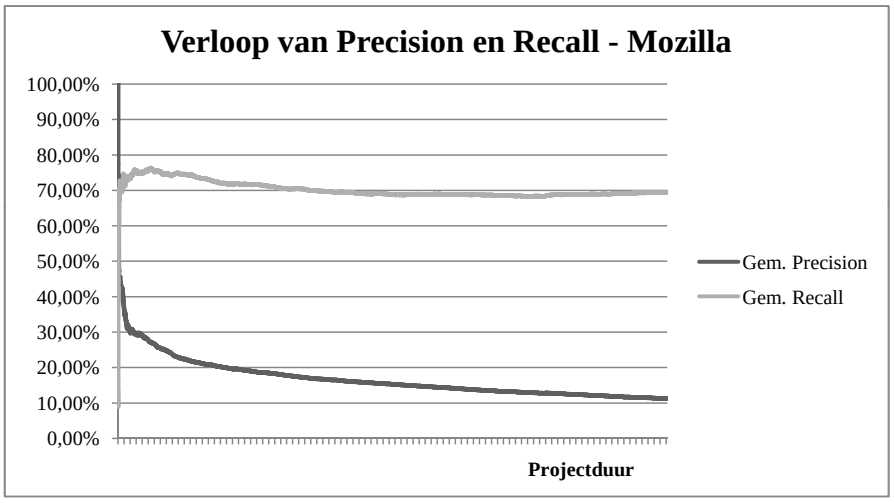
Uit de resultaten kan *niet* geconcludeerd worden dat in grotere wijzigingshistories de hoeveelheid nuttige informatie naar verhouding groter is dan in kleinere wijzigingshistories. Het Mozilla-project, dat het grootste onderzochte systeem is, laat in figuur 5.3 een redelijk gelijke verdeling zien van de voorspelde wijzigingen over het verloop van het project. Ook is te zien in figuur 5.4 dat de gemiddelde recall in het verloop van het project stabiliseert en niet stijgt. Op basis van deze resultaten kan dus niet beweerd worden dat grotere wijzigingshistories beter presteren dan kleinere wijzigingshistories.

5.2 Kwalitatief onderzoek

Met de resultaten van het kwalitatieve onderzoek wordt de waarde van het kwantitatieve onderzoek bepaald en wordt de tweede hypothese gevalideerd. De tweede hypothese luidt:



Figuur 5.3: Verdeling van de voorspelde wijzigingen in het projectverloop van Mozilla



Figuur 5.4: Verloop van de gemiddelde precision en recall in Mozilla

Open Source projecten zullen relatief beter presteren dan commerciële projecten vanwege kwalitatief betere commit logs.

Er zijn per project 10 goede en 10 slechte gevallen geanalyseerd. Een goed geval is gedefinieerd als een wijziging waarvoor alle aan te passen bestanden voorspeld konden worden (een recall-score van 100%). Een slecht geval daarentegen is gedefinieerd als een wijziging waarvoor geen enkel bestand voorspeld kon worden (een recall-score van 0%).

De resultaten van de kwalitatieve analyse worden hieronder gepresenteerd.

Typen wijzigingen

De onderzochte goede en slechte gevallen zijn gecategoriseerd op basis van het type wijziging. Het overzicht is te zien in tabel 5.4.

Tabel 5.4(a) laat zien dat de *corrective* wijzigingen (bugfixes) de grootste groep is in de categorie goede gevallen. Echter valt het merendeel van de slechte gevallen ook in deze categorie (zie tabel 5.4(b)). Er kan dus niet geconcludeerd worden dat wijzigingen in de *corrective* categorie beter te voorspellen zijn, maar wel dat deze categorie de grootste is.

Ook voor de andere typen wijzigingen is op basis van deze resultaten niet te zeggen of ze goed of slecht te voorspellen zijn.

Tekstuele beschrijvingen

Van de commit logs van de onderzochte gevallen en de bijbehorende zoekresultaten is geanalyseerd waar de tekstuele overeenkomsten op gebaseerd zijn. Uit de analyse bleek dat de wijzigingen om verschillende redenen tekstuele overeenkomsten hadden met elkaar.

In een ideale situatie worden er tekstuele overeenkomsten gevonden omdat commit logs over een gelijksoortig onderwerp (bijvoorbeeld een bepaalde functionaliteit) gaan en dus in hetzelfde deel van de code geïmplementeerd worden. In tabel 5.5 is een overzicht gemaakt van de goede gevallen per project die hieraan voldoen. Ook zijn hierin de slechte resultaten vernoemd

	Project X	**Project Y**	**Mozilla**	**Trac**	**Totaal**
Adaptive	0	0	0	1	1
Corrective	8	5	8	5	26
Enhancive	1	5	1	2	9
Groomative	1	0	1	2	4
Preventive	0	0	0	0	0

(a) Goede gevallen

	Project X	**Project Y**	**Mozilla**	**Trac**	**Totaal**
Adaptive	0	1	0	1	2
Corrective	4	2	8	4	18
Enhancive	6	3	1	4	14
Groomative	0	3	0	1	4
Preventive	0	1	1	0	2

(b) Slechte gevallen

Tabel 5.4: De goede en slechte gevallen, gecategoriseerd per type wijziging

Project	Goede gevallen	Slechte gevallen
Project X	4	3
Project Y	6	2
Mozilla	10	4
Trac	9	3

Tabel 5.5: Aantal gevallen met goede tekstuele resultaten op basis van gelijksoortig onderwerp

waarin commit logs een gelijksoortig onderwerp behandelden. Opvallend is dat van beide open source-projecten (bijna) alle goede gevallen het gevolg zijn van een tekstuele overeenkomst op basis van een gelijksoortig onderwerp. Bij de commerciële projecten ligt dit ongeveer op de helft.

De overige goede gevallen bij de commerciële projecten bleken veroorzaakt te zijn door het feit dat in de commit logs lijsten werden bijgehouden van aangepaste bestanden in de betreffende wijzigingen. Dit zorgde voor de situatie waarin wijzigingen die niet noodzakelijk aan elkaar gerelateerd waren, maar wel in dezelfde bestanden zijn geïmplementeerd, een hoge tekstuele overeenkomst met elkaar hadden. De berekende precision- en recall-scores liggen voor dergelijke resultaten hoog, en kunnen dus de uitkomst van het kwantitatieve onderzoek verbloemen.

Tekstuele overeenkomsten op basis van een gelijksoortig onderwerp leiden niet altijd tot een goed resultaat, zoals ook te zien is in tabel 5.5. Een mogelijke oorzaak hiervoor is dat de beschreven onderwerpen in de commit logs niet altijd specifiek genoeg zijn om teruggeleid te kunnen worden tot één bepaald bestand of een groep bestanden. Een ontwikkelaar bij Logica beaamt dat de beschrijvingen inderdaad niet altijd specifiek genoeg worden opgegeven. Wel wordt dit echter vaak gecompenseerd door een koppeling te maken naar de bugtracker, waarin wel een uitgebreidere beschrijving staat.

De overige slechte gevallen werden vooral veroorzaakt doordat wijzigingen gezamenlijk bepaalde algemene woorden bevatten. Deze woorden werden niet gefilterd door de stopwoordenlijst. Het feit dat deze resultaten geen goede score opleveren is niet van belang. Een ontwikkelaar zal deze woorden namelijk in een eventuele tool niet als zoektermen gebruiken. Een aantal slechte gevallen werden veroorzaakt door ambigue woorden in de commit logs. Al deze gevallen zijn het gevolg van het feit dat er puur op woorden wordt gezocht naar overeenkomsten. Het zou bijvoorbeeld voor kunnen komen dat een bepaald algemeen woord in een specifieke projectcontext een andere betekenis heeft, of dat ontwikkelaars voor bepaalde concepten verschillende termen gebruiken. Een meer semantische aanpak zou in dergelijke gevallen mogelijk betere resultaten opleveren.

Wat ten slotte nog opviel aan Project Y, was dat bij het willekeurig selec-

teren van de goede en slechte gevallen er voor een significant deel van de wijzigingshistorie geen wijzigingen beschikbaar waren. Na onderzoek bleek dat het grootste deel van de commit logs van Project Y gefilterd werd door de stopwoordenlijst, omdat er een standaardtekst werd gebruikt. Uit navraag bij de betreffende ontwikkelaars bleek dit te liggen aan het feit dat er in de loop van het project een migratie had plaatsgevonden naar een ander versiebeheersysteem. Het ontbreken van duidelijke commit logs verklaart de grote leegte in de grafiek op figuur B.2 in bijlage B. Ook verklaart dit de lage recall-scores in tabel 5.1.

Afstand tussen de wijzigingen

In tabel 5.6 is een overzicht geplaatst van de gemiddelde afstand tussen de wijzigingen en de bijbehorende zoekresultaten. De afstand tussen twee wijzigingen is hier uitgedrukt in het aantal wijzigingen dat heeft plaatsgevonden tussen de twee wijzigingen.

Wat opvalt is dat de gemiddelde afstanden voor de goede gevallen voor alle projecten lager liggen dan de gemiddelde afstanden voor de slechte gevallen. Dit zou mogelijk veroorzaakt kunnen worden doordat gedurende de evolutie van een systeem de locatie en functie van bestanden kan veranderen (vanwege de dynamiek van software). Een mogelijke oplossing hiervoor is om bij het analyseren van de wijzigingshistorie ook rekening te houden met de verplaatsing van bestanden.

De gemiddelde afstand bij de goede gevallen is voor de beide commerciële projecten het laagst. Als de afstand erg laag is, zoals bij Project X, zal het vooral wijzigingen betreffen die vlak na elkaar uitgevoerd zijn, en dus nog vers in het geheugen zullen zitten van de betreffende ontwikkelaars. Voor dergelijke wijzigingen zal het minder nuttig zijn dat ze voorspeld kunnen worden dan voor wijzigingen waar een langere termijn tussen zit.

	Gemiddelde afstand:	
Project	**Goede gevallen**	**Slechte gevallen**
Project X	21	290
Project Y	116	6281
Mozilla	18371	31694
Trac	298	2085

Tabel 5.6: Gemiddelde afstand tussen de wijzigingen en de zoekresultaten

Frequentie van bestanden in de zoekresultaten

Er is geanalyseerd of het wellicht loont om bestanden die vaker in de zoekresultaten voorkomen een hogere 'betrouwbaarheidsscore' te geven, omdat voor deze bestanden de kans mogelijkerwijs hoger is dat ze relevant zijn.

Voor gemiddeld 6 op de 10 gevallen blijken bepaalde bestanden inderdaad vaker voor te komen. Echter gaat het hier even vaak om irrelevante als om relevante bestanden. Voor de onderzochte gevallen lijkt het dus niet te lonen om een betrouwbaarheidsfactor te introduceren.

Precision-score

Als de impact van een wijziging handmatig bepaald moet worden, moet in principe het gehele systeem geanalyseerd worden. Alhoewel een ontwikkelaar uiteraard logische overwegingen kan maken om bepaalde delen van het systeem uit te sluiten van de analyse, kan dat juist de oorzaak zijn van fouten (Lindvall & Sandahl, 1998). Om te bekijken in hoeverre het gebruik van het onderzochte zoeksysteem de hoeveelheid te analyseren bestanden vermindert, wordt de precision-score afgezet tegen een 'globale' precision-score. Deze globale precision is gedefinieerd als het percentage bestanden dat aangepast wordt in een bepaalde wijziging, ten opzichte van het totaal aantal bestanden in het systeem.

Zoals te zien is in tabel 5.7 zijn de precision-scores een grote verbetering ten opzichte van de globale precision. De totale set van bestanden die geanalyseerd moeten worden voor een bepaalde wijziging, wordt dus drastisch verminderd. De enige kanttekening die hierbij geplaatst moet worden, is dat de ontwikkelaar er wel zeker van moet zijn dat alle te wijzigen bestanden daadwerkelijk in de set van zoekresultaten voorkomen (oftewel: de recall moet 100% zijn).

Project	**Precision-score**	**Globale precision**
Project X	9,47%	0,58%
Project Y	27,61%	0,09%
Mozilla	15,58%	0,004%
Trac	23,26%	0,42%

Tabel 5.7: Vergelijking tussen de gemiddelde precision-score en globale precision

Hoofdstuk 6

Conclusies en aanbevelingen

In paragraaf 6.1 zullen de conclusies vermeld worden welke zijn genomen naar aanleiding van de resultaten uit het vorige hoofdstuk. Vervolgens worden in paragraaf 6.2 aanbevelingen gedaan voor verder onderzoek.

6.1 Conclusies

In dit hoofdstuk zal de onderzoeksvraag uit paragraaf 3.1 beantwoord worden. Ook zullen de hypothesen (zie paragraaf 3.5) gevalideerd worden.

Het nut van de onderzochte methode

De onderzoeksvraag, zoals deze gedefinieerd was in paragraaf 3.1, luidde:

> *In hoeverre kan de kennis over de wijzigingen aan een softwaresysteem hergebruikt worden bij latere wijzigingen, door wijzigingen te relateren op basis van tekstuele overeenkomsten in de commit logs?*

De onderzochte methode van het indexeren van de commit logs lijkt nog niet dermate goede resultaten op te leveren om toegepast te worden in een praktijksituatie. Voor een beperkt deel van de wijzigingen bleek het inderdaad mogelijk de impact te voorspellen, maar voor het overgrote deel was dit niet mogelijk.

Omdat voor slechts een laag aantal wijzigingen bepaald kon worden welke bestanden aangepast moesten worden, maar er nog geen manier is om 'goede'

resultaten van 'slechte' resultaten te onderscheiden, zou de betrouwbaarheid van een eventuele tool laag zijn.

De commit logs bleken vaak niet specifiek genoeg om wijzigingen te relateren die in dezelfde bestanden plaatsvinden. Een andere manier van het relateren van de commit logs, waarbij bijvoorbeeld ook semantische technieken gebruikt worden, zou mogelijk voor enigszins betere resultaten kunnen zorgen.

Hypothese 1: De invloed van de grootte van de wijzigingshistorie

De eerste hypothese betrof de invloed van de grootte van de wijzigingshistorie op de resultaten en luidde:

> *Naarmate een wijzigingshistorie groeit, zal de hoeveelheid nuttige informatie sneller groeien ten opzichte van de totale hoeveelheid informatie, en een grotere wijzigingshistorie zal dus beter presteren dan een kleinere.*

Uit het experiment zijn geen aanwijzingen gekomen dat een grotere wijzigingshistorie leidt tot betere resultaten dan kleinere wijzigingshistories. De resultaten lijken over het algemeen redelijk constant verdeeld te zijn over het gehele projectverloop. De hypothese kan dus *niet* op basis van deze resultaten valide verklaard worden.

Hypothese 2: Open source versus commerciële projecten

De tweede hypothese betrof het verschil tussen open source-projecten en commerciële projecten en luidde:

> *Open Source projecten zullen relatief beter presteren dan commerciële projecten vanwege kwalitatief betere commit logs.*

Uit de kwantitatieve analyse bleek dat het Trac-project inderdaad betere scores behaalt, maar het Mozilla-project presteerde niet beter dan Project X. De kwalitatieve analyse toonde echter aan dat Project X een aantal valse resultaten bevat, doordat in de commit logs lijsten van gewijzigde bestanden bijgehouden werden. Ook bleek dat Project X en Y relatief de laagste hoeveelheid 'ideale' gevallen bevatten (tabel 5.5).

Voor de onderzochte systemen geldt dus inderdaad dat de open source-projecten over het algemeen beter presteerden dan de commerciële (Logica-)

projecten en ook kwalitatief betere commit logs bevatten. Er kan echter niet worden gegarandeerd dat de twee gekozen Logica-projecten representatief zijn voor alle Logica-projecten, danwel voor alle commerciële projecten. De hypothese is dus voor de gekozen systemen valide, maar is niet generaliseerbaar naar andere projecten.

6.2 Aanbevelingen

De conclusies tonen aan dat het experiment niet tot dermate goede resultaten heeft geleid welke aanleiding zouden geven tot het bouwen van een tool. Er is vervolgonderzoek nodig om de resultaten te verbeteren, voordat een eventuele tool van enig nut zal zijn in de praktijk. Hieronder volgen enkele ideeën voor vervolgonderzoeken.

Slimmere tekstanalyse: In het uitgevoerde onderzoek werden teksten alleen gerelateerd op overeenkomstige woorden. Er zou onderzoek gedaan kunnen worden naar andere manieren van het indexeren van de wijzigingshistories. Mogelijke verbeterpunten zijn:

- Het opstellen van een synoniemenlijst, om meer tekstuele relaties te vinden.
- Een meer semantische aanpak, waarbij de context waarin woorden geplaatst zijn meegenomen worden in de beoordeling.

Codelocaties: Er is in dit onderzoek geen rekening gehouden met de verplaatsing van de bestanden gedurende de evolutie van het systeem. Dit zou mogelijkerwijs de resultaten negatief beïnvloed kunnen hebben. Een toekomstig onderzoek zou hier rekening mee kunnen houden. Ook zou een andere granulariteit gekozen kunnen worden, bijvoorbeeld op klasseniveau, procedureniveau of regelniveau.

Clone detection: In dit experiment werd alleen bekeken of wijzigingen met een tekstuele relatie zich ook in dezelfde delen van de source code bevonden. Een andere aanpak zou zijn om te analyseren of tekstueel overeenkomstige wijzigingen op een gelijke *manier* geïmplementeerd zijn (dus niet noodzakelijk op een identieke plek). Dit zou met clone detection technieken onderzocht kunnen worden.

Gebruikerstesten: Het uitvoeren van een experiment met een eventuele tool en een aantal potentiële gebruikers zou mogelijk inzichten kunnen geven in hoe de tool zo gebruikersvriendelijk en effectief mogelijk gemaakt kan worden. Voor een Information Retrieval-systeem is het niet voldoende om slechts de precision en recall te berekenen. De

bruikbaarheid van een dergelijk systeem is minstens net zo belangrijk (Manning et al., 2008).

Alternatieve eigenschappen: De wijzigingshistorie biedt meer informatie dan dat er in dit onderzoek gebruikt is. Er zou bijvoorbeeld geanalyseerd kunnen worden welke bestanden vaak tegelijkertijd zijn aangepast, of welke bestanden vaak door dezelfde ontwikkelaar worden bewerkt (Hassan & Holt, 2004). Een combinatie van deze gegevens kan wellicht tot betere relaties leiden.

Bibliografie

Abbattista, F. , Lanubile, F. , Mastelloni, G. , & Visaggio, G. (1994). An Experiment on the Effect of Design Recording on Impact Analysis. In *Proceedings of the International Conference on Software Maintenance (ICSM'94)*, (pp. 253–259). IEEE Computer Society Washington, DC, USA.

Abdelfettah, E. (2005). A tool support for traceability in software product families. Master's thesis, Vrije Universiteit, Amsterdam, the Netherlands.

Antoniol, G. , Canfora, G. , Casazza, G. , & De Lucia, A. (2001). Maintaining traceability links during object-oriented software evolution. *Software: Practice & Experience*, *31*, 331–355.

Antoniol, G. , Canfora, G. , Casazza, G. , De Lucia, A. , & Merlo, E. (2002). Recovering Traceability Links between Code and Documentation. *Software Engineering, IEEE Transactions on*, *28*, 970–983.

Arkley, P. , Manson, P. , & Riddle, S. (2002). Position Paper: Enabling Traceability. In *1st International Workshop on Traceability in Emerging Forms of Software Engineering, Edinburgh, UK. September 28th*, (pp. 61–65).

Arkley, P. & Riddle, S. (2005). Overcoming the Traceability Benefit Problem. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE'05)-Volume 00*, (pp. 385–389). IEEE Computer Society Washington, DC, USA.

Arnold, R. & Bohner, S. (1996). *Software Change Impact Analysis*. IEEE Computer Society Press Los Alamitos, CA, USA.

Asuncion, H. , François, F. , & Taylor, R. (2007). An End-To-End Industrial Software Traceability Tool. In *Proceedings of the 6th joint meeting of the european software engineering conference and the 14th ACM SIGSOFT symposium on Foundations of software engineering*, volume 3, (pp. 115–124). ACM Press New York, NY, USA.

Bratthall, L. , Johansson, E. , & Regnell, B. (2000). Is a Design Rationale Vital when Predicting Change Impact? A Controlled Experiment on Software Architecture Evolution. In *Proceedings of the Second International Conference on Product Focused Software Process Improvement*, (pp. 126–139). Springer-Verlag London, UK.

Brin, S. & Page, L. (1998). The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, *30*(1-7), 107–117.

Canfora, G. & Cerulo, L. (2006). Fine Grained Indexing of Software Repositories to Support Impact Analysis. In *Proceedings of the 2006 international workshop on Mining software repositories*, (pp. 105–111). ACM Press New York, NY, USA.

Chapin, N. , Hale, J. , Khan, K. , Ramil, J. , & Tan, W. (2001). Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution Research and Practice*, *13*(1), 3–30.

Chen, A. , Chou, E. , Wong, J. , Yao, A. , Zhang, Q. , Zhang, S. , & Michail, A. (2001). CVSSearch: Searching through Source Code using CVS Comments. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*. IEEE Computer Society Washington, DC, USA.

Cleland-Huang, J. (2006). Just Enough Requirements Traceability. In *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)-Volume 01*, (pp. 41–42). IEEE Computer Society Washington, DC, USA.

Cleland-Huang, J. , Settimi, R. , BenKhadra, O. , Berezhanskaya, E. , & Christina, S. (2005). Goal-Centric Traceability for Managing Non-Functional Requirements. In *Proceedings of the 27th international conference on Software engineering*, (pp. 362–371). ACM Press New York, NY, USA.

Cleland-Huang, J. , Zemont, G. , & Lukasik, W. (2004). A Heterogeneous Solution for Improving the Return on Investment of Requirements Traceability. In *Proceedings of the Requirements Engineering Conference, 12th IEEE International (RE'04) - Volume 00*, (pp. 230–239).

De Lucia, A. , Fasano, F. , Oliveto, R. , & Tortora, G. (2005). ADAMS Re-Trace: A Traceability Recovery Tool. In *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, (pp. 32–41). IEEE Computer Society Washington, DC, USA.

Dekhtyar, A. (2006). Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods. *IEEE Transactions on Software Engineering*, *32*, 4–19.

Egyed, A. (2001). A Scenario-Driven Approach to Traceability. In *Proceedings of the 23rd International Conference on Software Engineering*, (pp. 123–132). IEEE Computer Society Washington, DC, USA.

Egyed, A. & Grünbacher, P. (2002). Automating Requirements Traceability: Beyond the Record & Replay Paradigm. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering (ASE'02)*. IEEE Computer Society Washington, DC, USA.

Glass, R. (2002). *Facts and Fallacies of Software Engineering*. Addison-Wesley Professional.

Gotel, O. & Finkelstein, C. (1994). An Analysis of the Requirements Traceability Problem. In *Proceedings of the First International Conference on Requirements Engineering, 1994*, (pp. 94–101).

Gutwin, C. , Penner, R. , & Schneider, K. (2004). Group Awareness in Distributed Software Development. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, (pp. 72–81). ACM New York, NY, USA.

Harrington, G. & Rondeau, K. (1993). An Investigation of Requirements Traceability to Support Systems Development. Master's thesis, Naval Postgraduate School, Monterey, USA.

Hassan, A. & Holt, R. (2004). Predicting Change Propagation in Software Systems. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04)-Volume 00*, (pp. 284–293). IEEE Computer Society Washington, DC, USA.

Hassan, A. & Holt, R. (2006). Replaying development history to assess the effectiveness of change propagation tools. *Empirical Software Engineering*, *11*(3), 335–367.

Hayes, J. , Dekhtyar, A. , Sundaram, S. , & Howard, S. (2004). Helping Analysts Trace Requirements: An Objective Look. In *Proceedings of the Requirements Engineering Conference, 12th IEEE International (RE'04)-Volume 00*, (pp. 249–259). IEEE Computer Society Washington, DC, USA.

IEEE (1990). *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. New York: Institute of Electrical and Electronics Engineers Computer Society.

IEEE Std 830-1998 (1998). *IEEE Recommended Practice for Software Requirements Specification*. Piscataway, N.J.: IEEE Computer Society.

Järvelin, K. & Kekäläinen, J. (2002). Cumulated Gain-Based Evaluation of IR Techniques. *ACM Transactions on Information Systems (TOIS)*, *20*(4), 422–446.

Jönsson, P. (2005). *Impact Analysis - Organisational Views and Support Techniques*. PhD thesis, Blekinge Institute of Technology, Ronneby, Sweden.

Knethen, A. von (2002). Automatic Change Support based on a Trace Model. In *Proceedings of the 1st International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE02), UK*.

Knethen, A. von & Grund, M. (2003). QuaTrace: A Tool Environment for (Semi-) Automatic Impact Analysis Based on Traces. In *Proceedings of the International Conference on Software Maintenance*, (pp. 246–255). IEEE Computer Society Washington, DC, USA.

Kraus, T. U. (2007). Generating system documentation augmented with traceability information, using a central XML-based repository. Master's thesis, TU Delft.

Lindvall, M. & Sandahl, K. (1998). How Well do Experienced Software Developers Predict Software Change? *Journal of Systems and Software*, *43*, 19–27.

Manning, C. , Raghavan, P. , & Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.

Marcus, A. , Xie, X. , & Poshyvanyk, D. (2005). When and How to Visualize Traceability Links? In *Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*, (pp. 56–61). ACM Press New York, NY, USA.

Neumuller, C. & Grunbacher, P. (2006). Automating Software Traceability in Very Small Companies: A Case Study and Lessons Learned. In *Automated Software Engineering: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, volume 18, (pp. 145–156). IEEE Computer Society Washington, DC, USA.

Pohl, K. (1996). PRO-ART: Enabling Requirements Pre-Traceability. In *Proceedings of the 2nd International Conference on Requirements Engineering (ICRE'96)*, volume 96, (pp. 76–84). IEEE Computer Society Washington, DC, USA.

Ramesh, B. (1998). Factors Influencing Requirements Traceability Practice. *Communications of the ACM*, *41*, 37–44.

Ramesh, B. & Jarke, M. (2001). Towards Reference Models for Requirements Traceability. *Software Engineering, IEEE Transactions on*, *27*, 58–93.

Ramesh, B. , Stubbs, C. , Powers, T. , & Edwards, M. (1997). Requirements traceability: Theory and practice. *Annals of Software Engineering*, *3*, 397–415.

Rijsbergen, C. van (1979). *Information Retrieval*. Dept. of Computer Science, University of Glasgow.

Sakai, T. (2007). On the reliability of information retrieval metrics based on graded relevance. *Information Processing and Management*, *43*(2), 531–548.

Strens, M. & Sugden, R. (1996). Change Analysis: A Step towards Meeting the Challenge of Changing Requirements. In *Proceedings of the IEEE Symposium and Workshop on Engineering of Computer Based Systems*, (pp. 278). IEEE Computer Society Washington, DC, USA.

Watkins, R. & Neal, M. (1994). Why and How of Requirements Tracing. *Software, IEEE*, *11*, 104–106.

Wieringa, R. (1995). An Introduction to Requirements Traceability. Technical report, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, the Netherlands.

Ying, A. T. T. (2003). Predicting Source Code Changes by Mining Revision History. Master's thesis, The University of British Columbia.

Zimmermann, T. , Weissgerber, P. , Diehl, S. , & Zeller, A. (2004). Mining Version Histories to Guide Software Changes. In *Proceedings of the 26th International Conference on Software Engineering*, (pp. 563–572). IEEE Computer Society Washington, DC, USA.

Bijlagen

Bijlage A

Traceability tools

In deze bijlage wordt een overzicht gegeven van enkele traceability tools. Hierbij is een onderscheid gemaakt tussen tools die gevonden zijn in de literatuur (A.1) en commerciële tools (A.2).

A.1 Enkele tools uit de literatuur

De volgende lijst geeft een overzicht van enkele tools die in de literatuur zijn voorgesteld.

- **ADAMS Re-Trace:** Een tool gebaseerd op *Latent Semantic Indexing* (LSI), een *Information Retrieval* (IR) techniek. Het gebruikt een relationeel model voor het leggen van relaties tussen artefacts ('objecten'). Het is, naar eigen zeggen succesvol, getest in 7 studentenprojecten. (De Lucia et al., 2005)

- **FOREST Project:** Een Eclipse plugin ontwikkeld bij Chess, waarin gebruik wordt gemaakt van een metamodel voor het managen van documentatie in softwareprojecten. Dit heeft als doel om de documentatie consistent en controleerbaar te maken. Doordat documenten en artifacts uit verschillende fasen van een softwareontwikkelproces (van requirements tot implementatie en testen) aan elkaar kunnen worden gelinkt, bevordert dit model de traceability binnen het project. (Kraus, 2007)

- **PRO-ART:** Een requirements pre-traceability tool (traceability van requirements naar de rationale ervan) gebaseerd op een traceability model van Pohl (1996).

- **QuaTrace:** Een toolomgeving voor het combineren van een requirements management tool (b.v. RequisitePro) en een CASE tool, waarmee requirements aan design componenten gerelateerd kunnen worden. (von Knethen & Grund, 2003)
- **RETRO:** Een tool die gebruik maakt van IR-technieken voor het vinden van traces tussen high-level en low-level requirements. (Dekhtyar, 2006)
- **Trace Analyzer:** Een tool die tijdens het uitvoeren van scenario's (die gelinkt zijn aan requirements) bijhoudt welke source code classes aangesproken worden. Op basis hiervan wordt bepaald welke relaties er zijn tussen de scenario's en de classes in de source code. (Egyed & Grünbacher, 2002)
- **TraceViz:** Een Eclipse plugin, gebaseerd op een LSI-techniek. (Marcus et al., 2005)
- **TraCS:** Een tool die verschillende traceability-technieken heeft gecombineerd, om verschillende typen requirements te kunnen tracen. (Cleland-Huang et al., 2004)
- Abdelfettah (2005) presenteert een Eclipse plugin waarmee artifacts uit verschillende fasen van het ontwikkelproces aan elkaar gelinkt kunnen worden. Het onderzoek is gebaseerd op feature models, en besteedt speciaal aandacht aan product families.

A.2 Enkele commerciële tools

Enkele commerciële requirements traceability tools zijn:

- **IBM Rational RequisitePro:** Een tool voor het definiëren van relaties tussen (high- en low-level) requirements. Kan gecombineerd worden met bijvoorbeeld ClearCase (configuratiemanagement) en ClearQuest (change management) om traceability naar de implementatie mogelijk te maken. `http://www-306.ibm.com/software/awdtools/reqpro/`
- **Lighthouse**. Een requirements management tool waarin traceability mogelijk is van requirements naar onder andere change requests, bugs, testen en taken. `http://lighthouse.artifactsoftware.com/`
- **Telelogic DOORS:** Een tool, puur gericht op het linken van requirements. Door deze tool te combineren met Telelogic Synergy

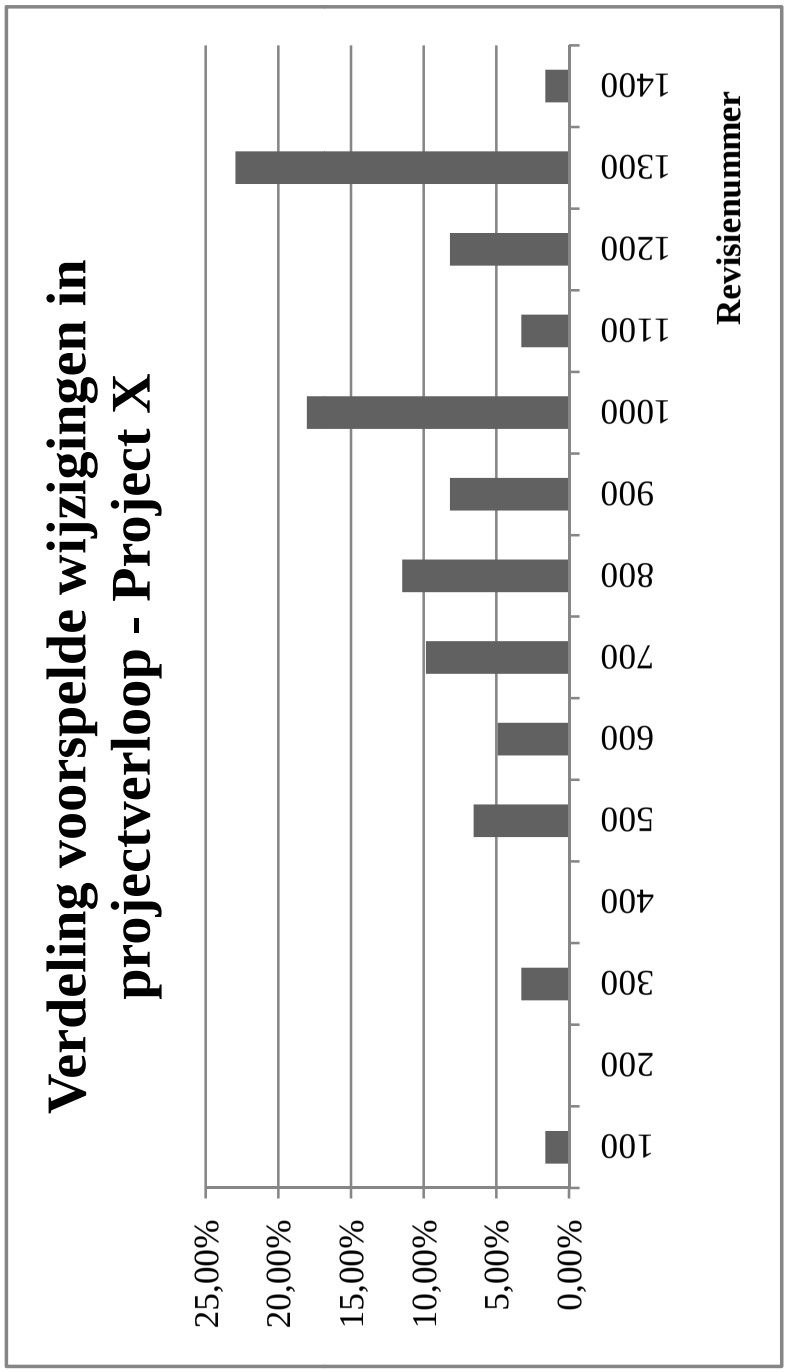
(configuratiemanagement) en Telelogic Change (change management), kan een volwaardige traceability oplossing worden gerealiseerd die op meerdere punten in het softwareontwikkelingsproces werkt. `http://www.telelogic.com/products/doors/index.cfm`

Bovenstaande lijst is niet volledig. Meer tools zijn bijvoorbeeld te vinden op de website van INCOSE (International Council on Systems Engineering)[1]. Daar is een overzicht te vinden van requirements management tools, waarvan een aantal tools ook ondersteuning bieden voor traceability.

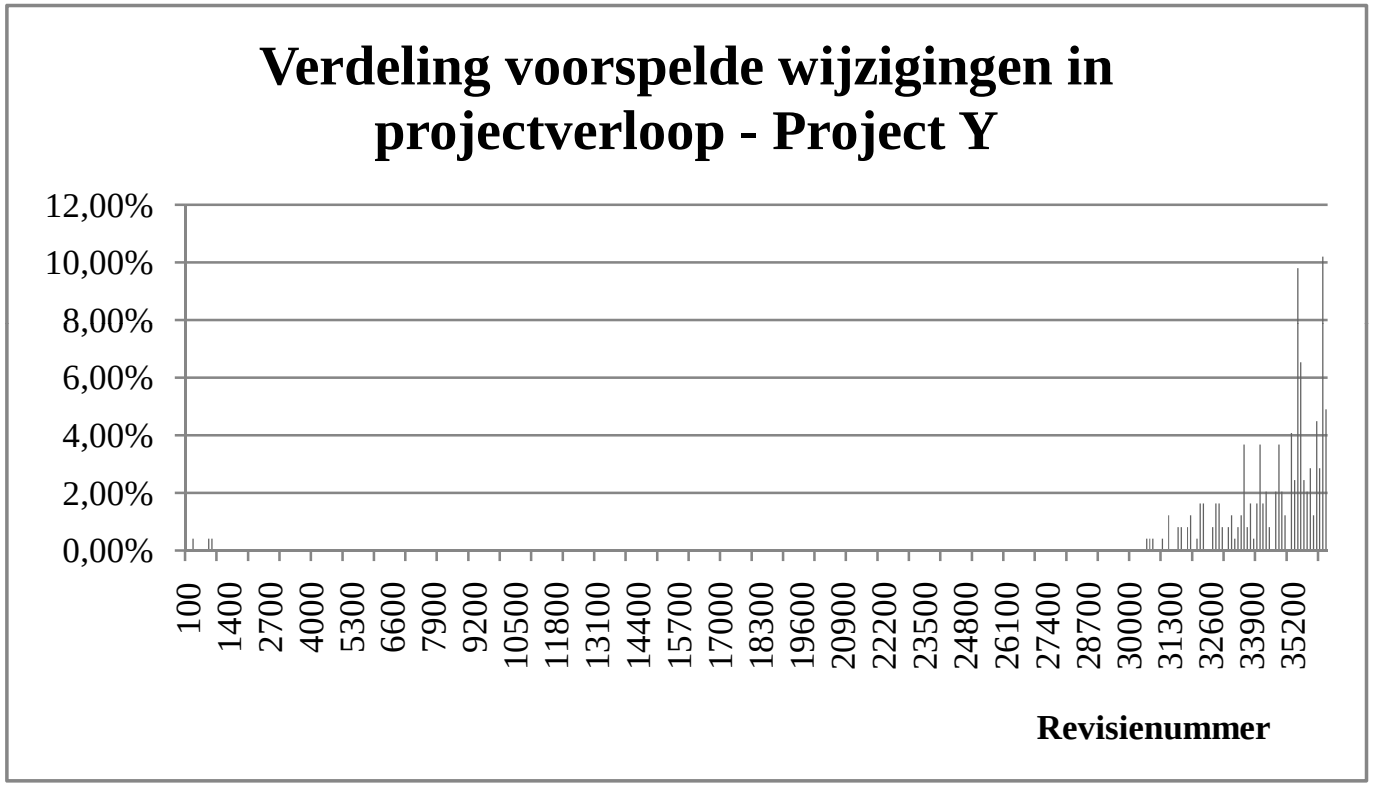
[1] `http://www.paper-review.com/tools/rms/read.php`

Bijlage B

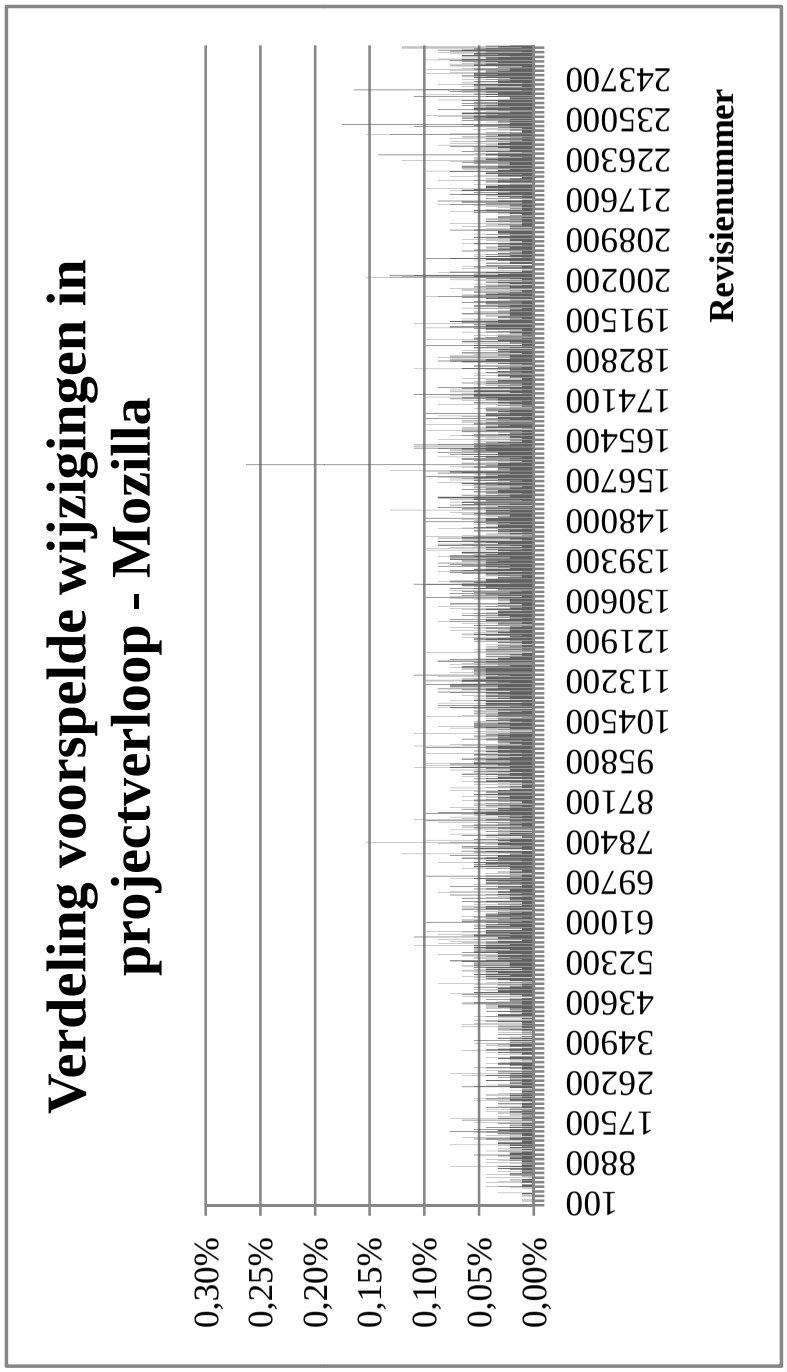
Grafieken



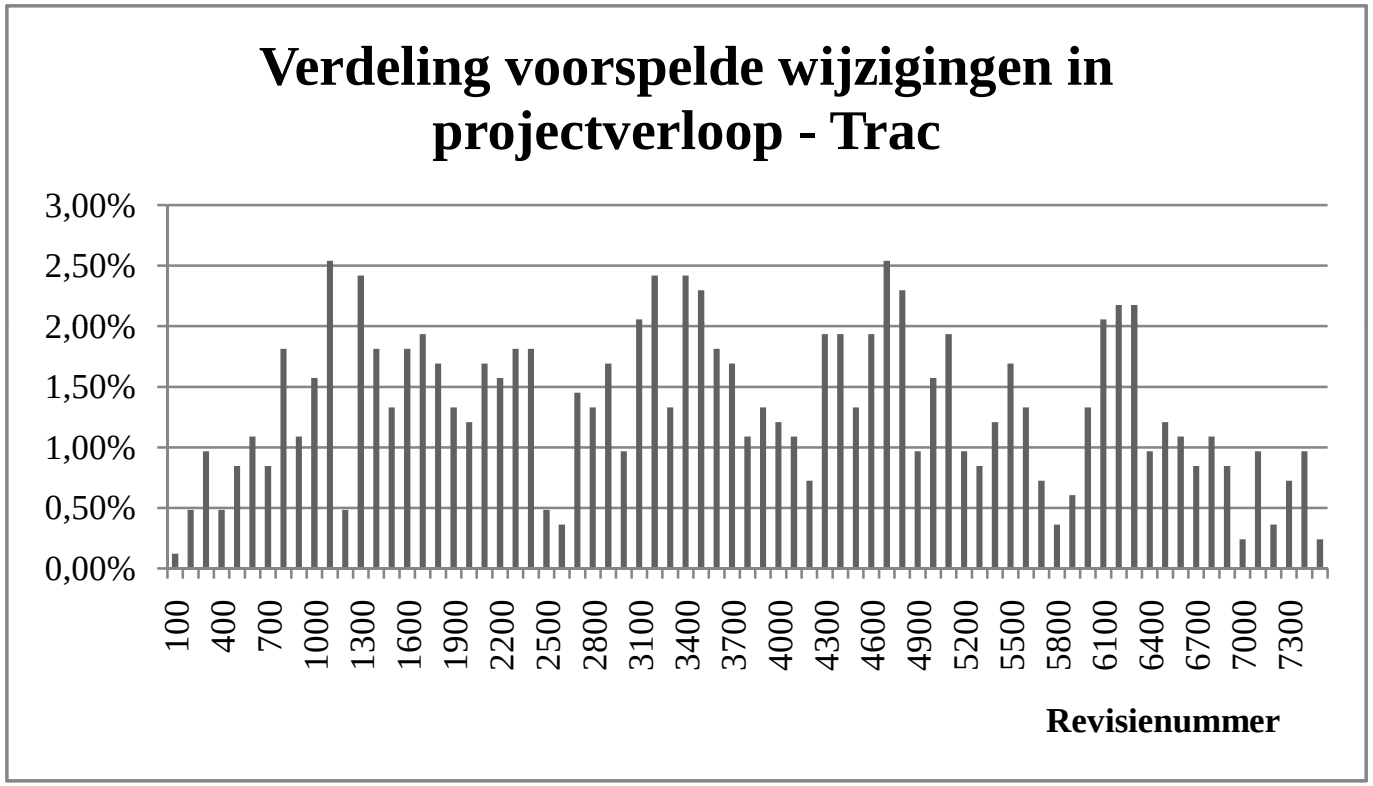
Figuur B.1: Verdeling van de voorspelde wijzigingen in het projectverloop van Project X



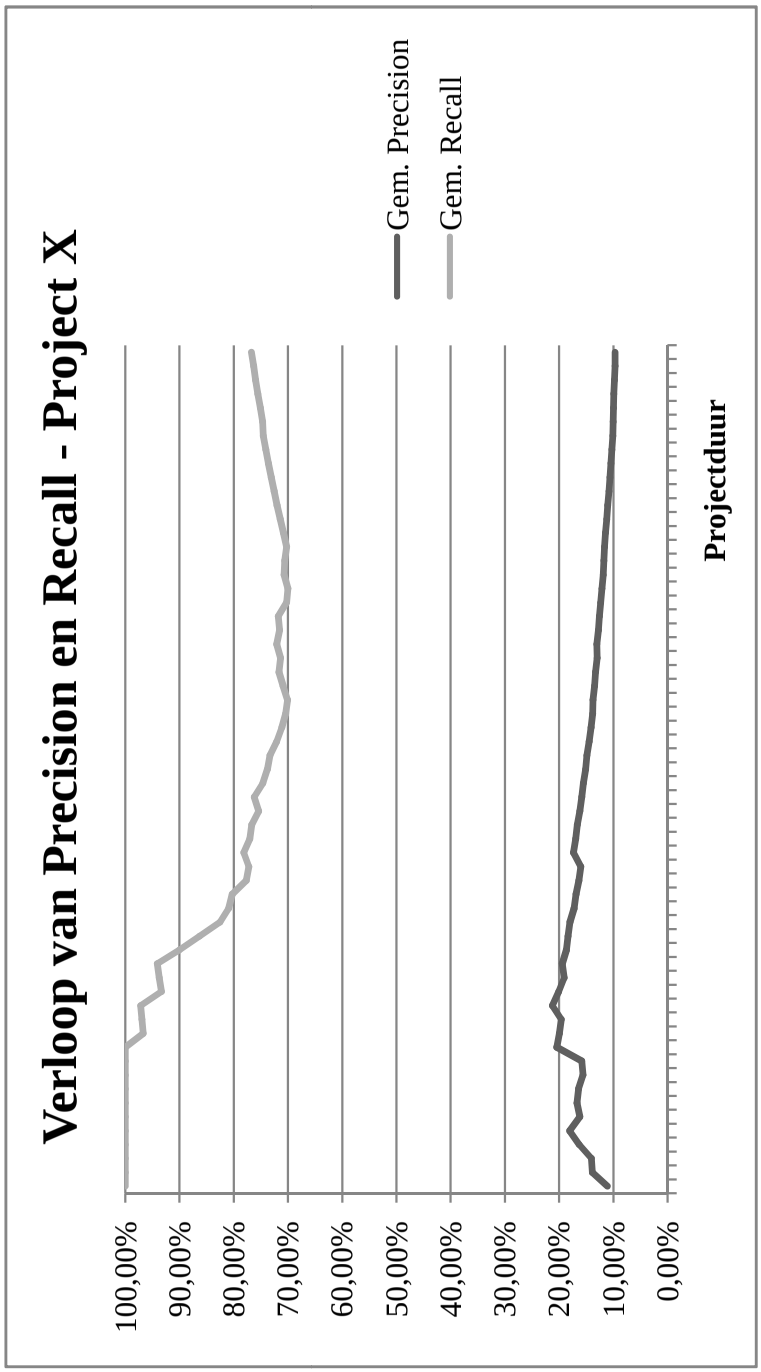
Figuur B.2: Verdeling van de voorspelde wijzigingen in het projectverloop van Project Y



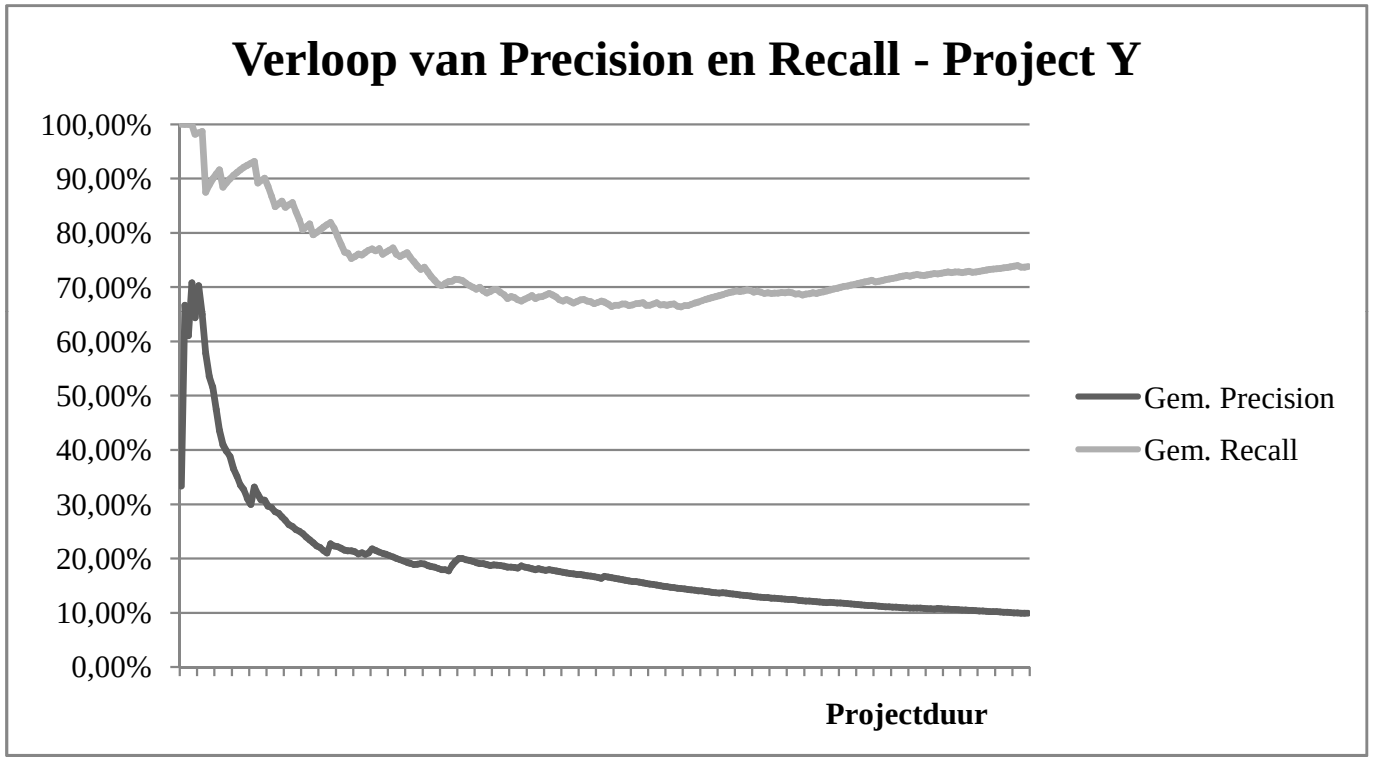
Figuur B.3: Verdeling van de voorspelde wijzigingen in het projectverloop van Mozilla



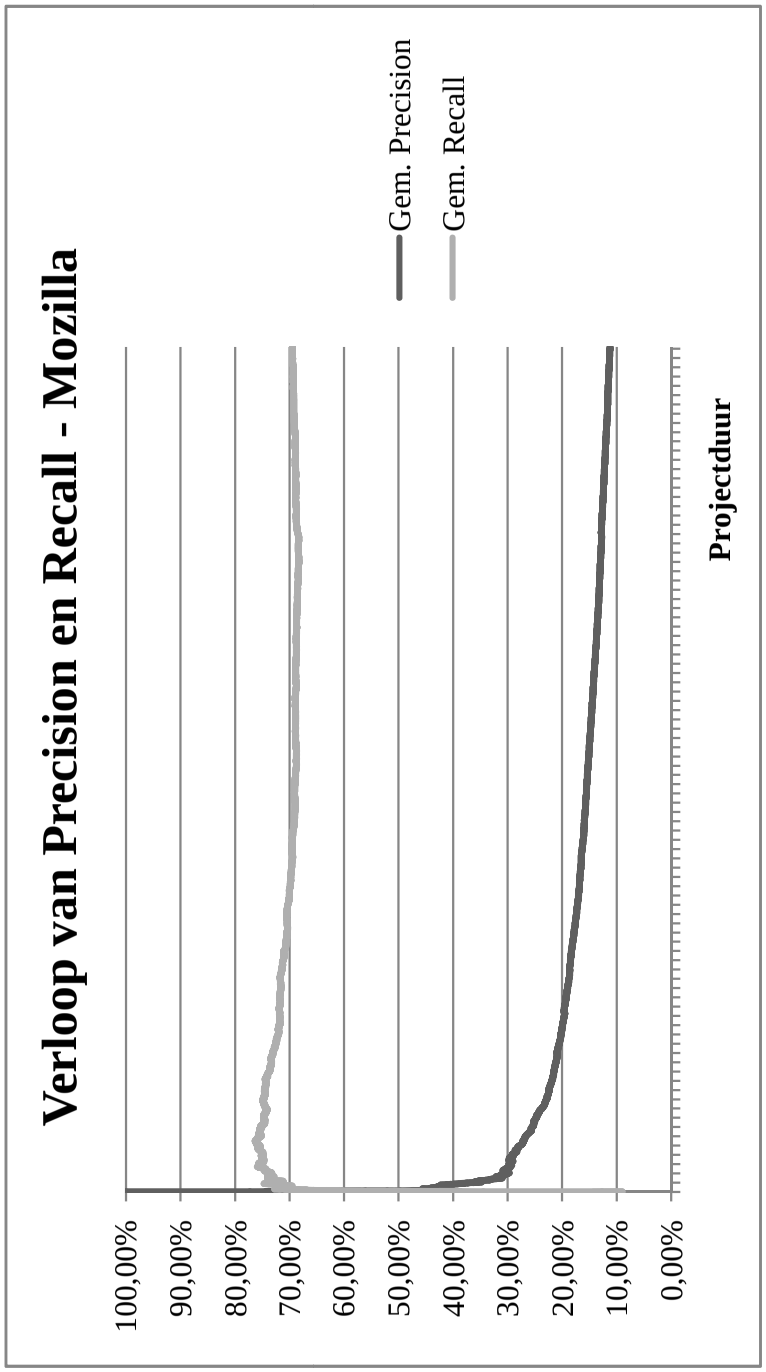
Figuur B.4: Verdeling van de voorspelde wijzigingen in het projectverloop van Trac



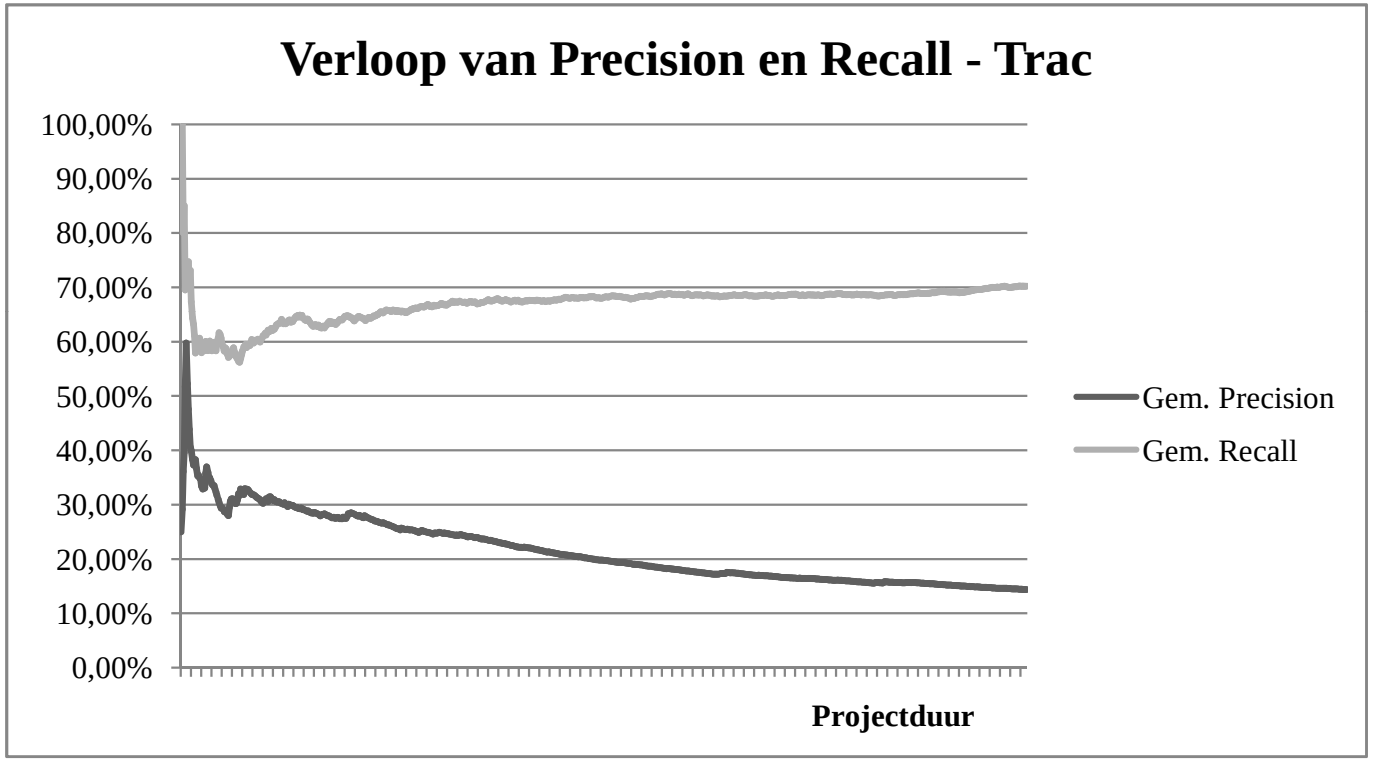
Figuur B.5: Verloop van de gemiddelde precision en recall in Project X



Figuur B.6: Verloop van de gemiddelde precision en recall in Project Y



Figuur B.7: Verloop van de gemiddelde precision en recall in Mozilla



Figuur B.8: Verloop van de gemiddelde precision en recall in Trac

Bijlage C

Stopwoorden

Hieronder volgt de stopwoordenlijst welke gebruikt is bij het indexeren van de wijzigingshistories. Om privacy-redenen zijn auteursnamen en bepaalde projectspecifieke namen uit de lijst verwijderd.

```
add, added, adding,
after,
all,
also,
author,
b,
back, backed, backing,
branch,
bug, bugs, bugzilla,
build, builds,
call,
can,
cause, causes, causing,
change, changed, changes, changing,
check, check-in, checkin, checked, checking,
class, classes,
clean, cleaned, cleaning,
close, closed, closes, closing,
code, coding,
comment, comments, commented,
commit, commits,
config,
correct, corrected, correcting, correction,
crash, crashes, crashing,
create, created, creates,
```

```
data,
date,
de,
do, don't, doesn't,
empty,
error, errors,
exp,
file, files,
first,
fix, fixed, fixes, fixing,
follow, follow-up,
framework,
from,
function, functional, functionality, functionalities,
get, getting,
have, has,
i, ic,
implementation,
implement, implemented, implementing, implements,
in,
initial,
instead,
jar, jars,
latest,
log, logging, logs,
made, make, making,
manufactured,
me,
merge, merged,
message,
method, methods,
modified,
module, modules,
more,
move, moved, moving,
mozilla,
name, names,
need, needed, needing, needs,
new,
now,
observation,
one,
only,
oops,
```

```
out,
p,
part,
patch, patches,
port, ported,
problem, problems,
procedure, procedures,
project,
qc,
r,
refactor, refactored, refactoring,
removal, remove, removed, removing,
review, reviewed,
revision, revisions,
script, scripts,
see,
set,
should,
so,
solved,
some,
source,
sr, src,
state,
string, strings,
support,
svn, svnmerge,
t,
tag,
test, tests,
thanks,
through,
ticket,
trac,
trunk,
up, upping, ups,
update, updated, updates, updating,
use, used, using,
version,
warning, warnings,
we,
when,
which,
work, works, working
```