Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

CONSTANTIN HASLER

MASTER THESIS

IMPLEMENTIERUNG UND PERFORMANCE-OPTIMIERUNG VON SCM ADAPTERN

Eingereicht am 29. Mai 2017

Betreuer: Prof. Dr. Dirk Riehle, M.B.A., M.Sc. Maximilian Capraro
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 29. Mai 2017

License

Erlangen, 29. Mai 2017

Abstract

Inner source (IS) is the use of open source software development practices and the establishment of an open source-like culture within organizations. To create metrics about the usage of IS within a specific corporation, data about the software development need to be extracted from source code management (SCM) systems. A developed crawl process retrieves the data over specially implemented adapters. To date adapters for git and manually exported CSV files from Microsoft Team Foundation Server (TFS) are in use. To automate data extraction from TFS a new adapter must be developed. Furthermore, the poor performance of the existing git adapter along with the crawl process needs to be improved. To validate the performance increase execution time and resource metrics are measured and compared. The result of this work is a newly developed TFS adapter and a performance-optimized git adapter and crawl process.

Zusammenfassung

Inner Source (IS) ist die Verwendung von Open Source Entwicklungspraktiken und die Einführung einer Open Source ähnlichen Kultur innerhalb eines Unternehmens. Um Metriken über die Anwendung von IS in einer speziellen Organisation erstellen zu können, müssen Daten über die Software-Entwicklung aus existierenden Source-Code-Management (SCM) Systemen extrahiert werden. Ein entwickelter Crawl-Prozess nutzt speziell implementierte Adapter, um diese Daten abzufragen. Bislang werden Adapter für Git und manuell exportierte CSV-Dateien von Microsoft Team Foundation Servern (TFS) eingesetzt. Zur Automation der Datenextraktion von TFS Systemen muss ein neuer Adapter entwickelt werden. Zudem soll die mangelhafte Performance des bestehenden Git Adapter sowie des Crawl-Prozesses verbessert werden. Um die Performance-Optimierung überprüfen zu können, werden die Ausführungszeit und verschiedene Ressourcenmetriken aufgezeichnet und verglichen. Das Ergebnis dieser Arbeit ist ein neu implementierter TFS Adapter sowie ein Performance-optimierter Git Adapter und Crawl-Prozess.

Inhaltsverzeichnis

1 Einleitung

Inner Source ist die Verwendung von Open Source Entwicklungsmethoden und die Einführung einer Open Source ähnlichen Kultur innerhalb eines Unternehmens. Dies ist mittlerweile bei großen Organisationen zunehmend verbreitet (Capraro & Riehle, 2016). Die Kooperation verschiedener Abteilungen an gemeinsamen Projekten sowie die Wiederverwendung von Programmcode bieten neue Möglichkeiten für die kommerzielle Softwareentwicklung.
Im Rahmen der Forschungsarbeit des Open-Source Lehrstuhls der Friedrich-Alexander-Universität Erlangen-Nürnberg werden Metriken entwickelt, mit deren Hilfe die Anwendung von Inner Source innerhalb von Unternehmen gemessen werden kann. Um die Qualität der Zusammenarbeit und weitere Faktoren bestimmen zu können, müssen Daten aus den Source-Code-Management (SCM) Systemen der Unternehmen extrahiert werden. Hierfür wurde ein Crawl-Prozess entwickelt, welcher alle benötigten Daten sammelt, transformiert und in eine externe Datenbank überträgt. Da nicht alle SCM Systeme über eine einheitliche Schnittstelle verfügen, werden verschiedene Adapter im Crawl-Prozess eingesetzt, die je nach SCM System individuell implementiert sind.
Aktuell existieren Adapter für Git und von Microsoft Team Foundation Server (TFS) exportierte CSV-Dateien. Da der CSV-Export von TFS Daten umständlich ist und nicht voll automatisiert durchgeführt werden kann, soll ein Adapter entwickelt werden, welcher die mit TFS Server 2013 neu eingeführte REST API zur Abfrage der benötigten Daten nutzt. Zudem soll der bestehende Git Adapter als auch der neu entwickelte TFS Adapter in Bezug auf verschiedene Performance-Aspekte optimiert werden. Der Crawl-Prozess selbst wird in die Performance-Messung und die nachfolgende Optimierung mit eingebunden.

Der Beitrag dieser Arbeit besteht aus drei Punkten:

- Erstellung eins objektorientierten Entwurfs für den TFS Adapter sowie die anschließende Implementierung
- Anlage eines Performance-Messkonzepts und die Durchführung von initialen Messungen an den Adaptern und dem Crawl-Prozess
- Optimierung des Git und TFS Adapters aufgrund der gesammelten Messdaten

Im folgenden Kapitel 2 werden zunächst die Artefakte der Arbeit beschrieben und die zu erfüllenden Anforderungen definiert. In Kapitel 3 wird das bestehende System sowie der aktuell eingesetzte Crawl-Prozess vorgestellt und mit bestehenden Lösungen verglichen. Anschließend wird in Kapitel 4 der objektorientierte Entwurf des TFS Adapters erläutert. Im darauffolgenden Kapitel 5 werden hervorzuhebende Implementierungsabschnitte präsentiert und gegen alternative Lösungen abgewägt. Die Evaluation und Optimierung der Software wird in Kapitel 6 durchgeführt. Abschließend wird das Ergebnis der Arbeit in Kapitel 7 zusammengefasst.

2 Artefakte der Arbeit

Die Artefakte dieser Arbeit sind der zu entwickelnde TFS SCM Adapter sowie der optimierte Crawl-Prozess und Git Adapter. Um den Nutzen der Software aufzuzeigen, wird im folgenden Abschnitt 2.1 der Zweck der Artefakte erläutert. Die zu erfüllenden Anforderungen werden anschließend in Abschnitt 2.2 definiert.

2.1 Zweck der Artefakte

Im Rahmen dieser Arbeit wird ein Adapter entwickelt, der zur Extraktion ausgewählter SCM Daten von TFS Servern eingesetzt werden kann. Die Daten werden in bestehende Datenmodelle transformiert und als Iterator zurückgegeben. Eingebunden wird der Adapter in einen existierenden Crawl-Prozess, der in Abschnitt 3.2 genauer beschrieben wird.
Die Performance-Optimierung des Crawl-Prozesses und Git-Adapters soll das Laufzeitverhalten sowie die Ressourcennutzung während der Durchführung des Crawl-Prozesses verbessern, da diese Faktoren bei der bisherigen Entwicklung nicht priorisiert wurden und Verbesserungspotential bieten.

2.2 Anforderungen an die Artefakte

Da bislang keine automatisierte Möglichkeit besteht Daten aus TFS SCM Systemen auszulesen, ist die Entwicklung eines Adapters zur automatisierten Datenextraktion notwendig. Mit der Einführung einer REST API für TFS Server 2013 und nachfolgende Versionen besteht die Möglichkeit, die Daten automatisiert abzufragen. Deswegen soll die API in der Entwicklung genutzt werden. Um eine problemlose Integration in das bestehende System zu ermöglichen, soll der entwickelte Adapter das definierte SCM Adapter-Interface implementieren.

Die Performance-Optimierung der Adapter sowie des Crawl-Prozesses ist nötig, da diese sehr lange Ausführungszeiten aufweisen. Um die Problematiken identifi-

zieren zu können, soll zunächst eine initiale Analyse durchgeführt werden. In der nachfolgenden Optimierung soll eine iterative Verbesserung der Performance für die Adapter sowie den Crawl-Prozess erfolgen.

Abschließend können die folgenden Anforderungen festgehalten werden:

- Entwicklung eines Adapters zur Datenextraktion von TFS SCM Systemen
- Verwendung der TFS REST API
- Implementierung des bestehenden Adapter Interfaces
- Performance-Analyse und Optimierung des entwickelten TFS Adapters, des Git Adapters und des Crawl-Prozesses

3 Bestehendes System

Die Entwicklungsaufgaben für die SCM Adapter werden in einem bestehenden Projekt namens „Collaboration Metric Suite" (CMSuite) durchgeführt. Das Ziel des Projekts ist die Erstellung von Metriken über die Zusammenarbeit in Unternehmen. Ein Faktor ist hierbei der „Patch-Flow". Dieser beschreibt den Strom an Patches über interne Organisationsgrenzen, wie Organisationseinheits-, Projekt- oder Profit-Center-Grenzen, hinweg. Das Projekt enthält entwickelten Sourcecode als auch benötigte Softwarewerkzeuge, um den „Patch-Flow" zu vermessen und die Metriken über die Verwendung von Inner Source zu erstellen.
In Abschnitt 3.1 werden relevante Komponenten des Systems sowie deren Abhängigkeiten genauer erläutert. Anschließend werden in Abschnitt 3.2 die Aufgaben und Funktionen des bestehenden Crawl-Prozesses beschrieben. Abschließend wird der Crawl-Prozess mit bestehenden Lösungen zur SCM Datenextraktion in Abschnitt 3.3 verglichen.

3.1 Komponenten des Systems

Das Projekt *CMSuite* ist in 5 Komponenten unterteilt, die in insgesamt 22 Unterkomponenten untergliedert sind. Relevant für diese Arbeit sind insbesondere die Komponenten *Commons* und *PfCrawler*, die im Folgenden genauer beschrieben werden. Die Komponenten *Accountancy*, *Datamanager* und *Webclient* verarbeiten und übertragen die in der Datenbank persistierten Daten. Da diese Funktionalitäten nicht weiter wichtig für die Adapterimplementierung sind, werden diese nicht genauer erläutert.

Commons

Die Komponente *Commons* umfasst allgemein genutzte Funktionalitäten, beschreibt die Datenmodelle und regelt die Persistierung sowie die Anbindung zur Datenbank.
Zu den Funktionalitäten zählen häufig genutzte Logging-, Datums- und Dateiimplementierungen. Diese besitzen keine Abhängigkeiten mit anderen Modulen und

können somit von allen übrigen Komponenten genutzt werden, ohne zyklische Abhängigkeiten zu generieren.
Die für diese Arbeit relevanten Datenmodelle beziehen sich, wie in Abbildung 3.1 dargestellt, zum einen auf die Repräsentation von Patches und die zugehörigen *FileChanges* sowie zum anderen auf die übergeordneten Abhängigkeiten. Ein *FileChange*-Objekt repräsentiert hierbei eine konkrete Änderung an einer Datei, während ein Patch eine Sammlung an *FileChange*-Objekten sowie zugehörige Metadaten umfasst. Zugewiesen ist ein Patch einem Repsoitory, einem *Crawl-Run*

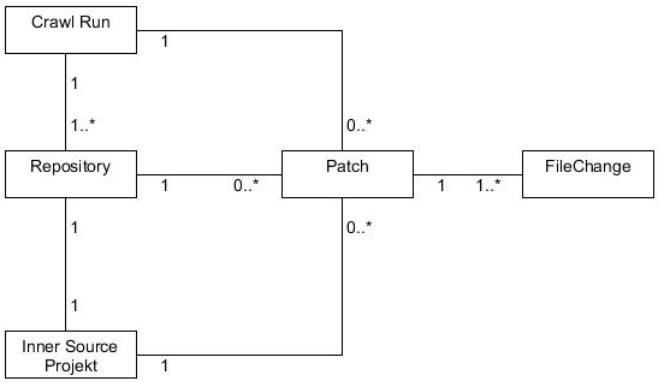

Abbildung 3.1: Datenmodelle und Assoziationen der CMSuite

sowie einem Inner Source Projekt. Diese Modelle sind wiederum untereinander abhängig. Ein Repository beschreibt einen konkreten Ablageort eines SCM Systems für Daten, Dokumente und Programmcode.
Um die Kommunikation mit der Datenbank zu kapseln, werden für die einzelnen Modelle Datenzugriffsobjekte, sogenannte DAOs (DataAccessObjects) im Persistenzmodul der Komponente erstellt. Diese greifen auf Funktionalitäten von Hibernate[1] zurück und bieten verschiedene Speicher- oder Ladeoperationen für die Modelle an.

PfCrawler

Die Komponente *Patch-Flow Crawler* (PfCrawler) implementiert alle Funktionalitäten, die für die Ausführung des Crawl-Prozesses benötigt werden und bietet einen Einstiegspunkt zur Ausführung eines *Crawl-Runs*. Von besonderer Relevanz für diese Arbeit ist die Definition der Interfaces *Plugin* und *Adapter*, welche im Rahmen dieser Arbeit implementiert werden.
Das *Adapter* Interface stellt eine Methode zur Verfügung, die alle Patches von einem übergebenen Repository in einem definierten Zeitintervall abfragt und als Iterator zurückgibt.
Die Konfiguration eines *Crawl-Runs* wird über das Interface *Plugin* spezifiziert und kann je nach eingesetzter Umgebung individuell implementiert werden. So

[1] Hibernate ist ein Open Source Java Persistenz-Framework - http://hibernate.org/.

können beliebig viele Schritte für vor als auch nach dem *Crawl-Run* sowie für die Patch-Verarbeitung angegeben werden. Zudem muss der zu verwendende SCM Adapter definiert werden.

3.2 Crawl-Prozess

Da der Crawl-Prozesses von zentraler Bedeutung für die Entwicklung und Optimierung neuer Adapter ist, wird er im Folgenden genauer beschrieben. Die Aufgabe des Crawl-Prozesses ist die Extraktion von Daten für definierte Inner Source Projekte, die Transformation der Daten in Modelle und die Persistierung dieser in einer konfigurierten Datenbank.

Parametrisierung

Um den Crawl-Prozess für wechselnde Umgebungen konfigurieren zu können, müssen verschiedene Einstellungen vor Prozessbeginn definiert werden. Neben einem *Plugin* muss zudem ein Datum angegeben werden, bis zu dem alle Patches geladen werden sollen. So können bestimmte Zeitabschnitte speziell untersucht oder sequenzielle Durchläufe ohne Überschneidung durchgeführt werden.

Prozessschritte

Zu Beginn des Prozesses werden alle im *Plugin* definierten Pre-Schritte ausgeführt. Hierbei können initiale Datenbestände für Projekte und Repositories oder Organisationselemente festgelegt werden.
Im nächsten Schritt werden alle verfügbaren Repositories aus der Datenbank geladen und für jedes Element ein eigenständiger *Crawl-Job* gestartet. Im *Crawl-Job* werden zunächst für das jeweilige Repository alle Patches für das definierte Zeitfenster aus dem SCM System extrahiert und transformiert. Hierfür wird der im *Plugin* definierte Adapter eingesetzt. Für jeden Patch werden im Anschluss die im *Plugin* festgelegten Patch-Verarbeitungsschritte durchgeführt und die Persistierung in der Datenbank eingeleitet. Nach Abschluss aller *Crawl-Jobs* werden im Prozess alle Post-Schritte des *Plugins* ausgeführt und der Crawl-Prozess in der Datenbank als abgeschlossen gekennzeichnet.

Endzustand

Nach erfolgreichem Durchlauf des Crawl-Prozesses sind alle Patches der Repositories für das Zeitfenster ermittelt und mit den benötigten Verarbeitungsschritten in die Organisationsstrukturen der Umgebung eingebunden worden. Die gesammelten Daten können daraufhin von anderen Komponenten des Projekts zur Erstellung von Metriken verwendet werden.

3.3 Vergleich mit anderen Lösungen

Um den Prozess mit bestehenden Lösungen vergleichen zu können, wurde eine Literaturrecherche zu den Themen „Repository Mining" sowie „SCM System Mining" durchgeführt.

Zunächst wurde nach wissenschaftlichen Arbeiten gesucht, die „Repository Mining" in ihrer Arbeit einsetzen und Strategien zur Datenextraktion beschreiben. Ein dazu veröffentlichter Artikel von German et al. (German, Adams, & Hassan, 2016) über das kontinuierliche Abfragen von Daten aus verteilten Versionskontrollsystemen (Distributed version control systems, D-VCS), wie Git oder SVN, beschreibt einen Crawl-Algorithmus für D-VCS Repositories. Dieser gleicht dem in Abschnitt 3.2 definierten Crawl-Prozess in einigen Punkten. Unterschiede finden sich hauptsächlich in der detaillierten Verarbeitung von Patches. Im Ansatz von German et al. wird das Überprüfen von nachträglichen Veränderungen an Patches in einem Repository durch löschen, „rebasing" oder „cherry-picking" in den Crawl-Prozess mit einbezogen. Dies kann auf die unterschiedlichen Ziele der Datenextraktion zurückgeführt werden. Während German et al. den Fokus auf die Entstehung und Verknüpfung von Patches legt, ist das Ziel des *CMSuite*-Crawl-Prozesses konkrete Daten über neu erstellte Patches und deren Inhalt und Ersteller zu sammeln. Die Umsetzung des Algorithmus wird von German et al. mittels eines Perl Skrips durchgeführt, der Quellcode des Skripts wurde jedoch nicht angegeben und kann somit nicht genauer analysiert werden.
Eine fertig implementierte Lösung zur Datenextraktion aus Repositories und zum Erstellen von Metriken ist die von Sokol et al. (Sokol, Aniche, & Gerosa, 2013) vorgestellte Webapplikation MetricMiner. Diese kann zur Analyse von Git oder SVN Repositories eingesetzt werden und soll die Abfrageausführung bei wissenschaftlichen Arbeiten vereinfachen. Eine genaue Analyse der Webapplikation ist nicht möglich, da zum Zeitpunkt dieser Arbeit die Webapplikation nicht online angeboten wird. Der zugrundeliegende Quellcode wurde als Java-Framework unter dem Namen „Metric Miner 2" veröffentlicht und ist auf Github[2] hinterlegt. Durch den Vergleich des Sourcecodes mit der bestehenden Implementierung

[2]https://github.com/mauricioaniche/repodriller

konnte festgestellt werden, dass die erstellten Abläufe, Implementierungen sowie verwendeten Werkzeuge sehr ähnlich sind. Eine Integration des Frameworks in das Projekt *CMSuite* wurde nicht durchgeführt, da alle benötigten Funktionalitäten bereits implementiert wurden und eine Umstellung demnach keinen erkennbaren Mehrwert bietet. Zudem bietet das Framework keine Erweiterbarkeit für weitere SCM Systeme wie den TFS Server, die im Rahmen dieser Arbeit nötig ist. Ein weiterer Grund gegen die Integration ist, dass nur Metadaten über Commits persistiert werden und keine Organisations-spezifischen Daten, die jedoch für das *CMSuite*-Projekt von Bedeutung sind.

Da bei den Ergebnissen der Recherche zu „Repository Mining" häufig keine Angaben zu verwendeten Extraktionsstrategien gemacht werden, wurde die weitere Suche auf „Repository Mining Tools" fokussiert. Eine ausführliche Übersicht über in Forschungsarbeiten verwendete *Repository Mining Tools* bietet hierbei der Artikel von Chaturvedi et al. (Chaturvedi, Sing, & Singh, 2013). Dieser analysiert verschiedene Veröffentlichungen zu dem Thema „Mining Software Repositories" und klassifiziert die verwendeten Werkzeuge. Die resultierende Tabelle wurde auf Programme untersucht, die mit dem Crawl-Prozess der *CMSuite* verglichen werden können. Hierbei wurde lediglich ein Programm gefunden, das zur Datenextraktion aus Git Repositories genutzt werden kann. Der von Bird et al. (Bird u. a., 2009) auf Github veröffentlichte Quellcode[3] für das Programm *GitMiningTools* wurde hierfür analysiert. Da der Funktionsumfang des verwendeten Perl-Skripts auf das Abspeichern von Patch-Daten aus einem lokalen Repository in einer Datenbank beschränkt ist, ist ein sinnvoller Einsatz in den bestehenden Crawl-Prozess nicht möglich.
Die Literaturrecherche zu dem Thema „SCM System Mining" hat keine verwendbaren Ergebnisse vorgebracht. Dies ist auf die etablierte Bezeichnung „Mining Software Repositories" zurückzuführen, die andere Begriffe in der Fachliteratur ersetzt hat.

Der Vergleich mit anderen Lösungen hat zusammenfassend gezeigt, dass bestehende Ansätze ähnlich zu dem implementierten Prozess sind. Jedoch erfüllt wie Tabelle 3.1 abgebildet keine der Lösungen die nötigen Voraussetzungen, um in das Projekt integriert zu werden. Zudem wurde keine Lösung für die Abfrage von Daten aus TFS SCM Systemen gefunden. Dies bestätigt die Notwendigkeit der Implementierung eines TFS Adapters.

[3] https://github.com/cabird/git_mining_tools

	CMSuite	German	MetricMiner	GitMiningTools
Repository-Download	X	X	X	-
Datenextraktion	X	X	X	X
Datentransformation	X	X	X	-
Organisationsdaten	X	-	-	-
Persistierung	X	X	X	X
Erweiterbarkeit	X	-	-	-
Code vorhanden	X	-	X	X

Tabelle 3.1: Vergleich einzelner Faktoren, die zur Integration in das Projekt erfüllt sein müssen. Verglichen wird die Implementierung der CMSuite mit dem Algorithmus von German et al., dem Framework *MetricMiner* und dem Programm *GitMiningTools*. „X" steht für erfüllt, „-" steht für nicht erfüllt.

4 Objektorientierter Entwurf

Im nachfolgenden Kapitel wird der objektorientierte Entwurf für den im Rahmen dieser Arbeit zu entwickelnden TFS Adapter vorgestellt. Zunächst werden in Unterabschnitt 4.1 die Komponenten vorgestellt und auf die die jeweiligen Klassen eingegangen. Anschließend werden die verwendeten Entwurfsstrategien erläutert und Beispiele für die Anwendung aufgezeigt.

4.1 Komponentenübersicht

Da die Ziele des TFS Adapter die Datenextraktion sowie die Datenverarbeitung sind, wurden zwei Komponenten entwickelt, die diese Funktionalitäten getrennt implementieren. Wie in Abbildung 4.1 dargestellt, verwendet die Komponente *TFS Adapter* den *Client*, der wiederum mit der TFS REST API des SCM Servers kommuniziert. Der Client kapselt hierbei die Datenextraktion und bietet eine übersichtliche Schnittstelle, die vom Adapter genutzt werden kann. Im Folgenden wird der Aufbau der einzelnen Komponenten genauer beschrieben.

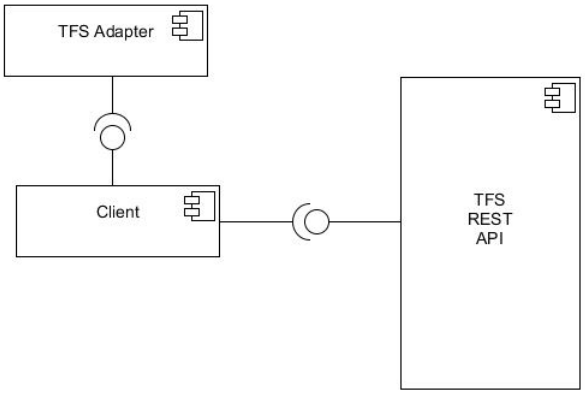

Abbildung 4.1: Adapter und verwendete Komponenten

4.1.1 TFS Adapter

Die Aufgabe der Komponente *TFS Adapter* ist, die vom Client abgefragten Daten in bestehende Datenmodelle zu transformieren und an den Aufrufer zurückzugeben. Dazu wurden, wie in Abbildung 4.2 dargestellt, verschiedene Klassen erstellt, die eigene Aufgabenbereiche besitzen. Im nachfolgenden Abschnitt werden die einzelnen Klassen und deren Funktionalitäten genauer erläutert.

TFS SCM Adapter

Die Klasse *TfsScmAdapter* beschreibt den Einstiegspunkt für den TFS Adapter und implementiert das bestehende *Adapter* Interface. Dieses Interface definiert eine Methode *fetch()*, der ein Repository sowie ein Start- und Enddatum übergeben wird und einen Iterator über Patches zurückgibt. Zur Implementierung der Methode wird eine Instanz der *PatchIterator* Klasse erstellt und an den Aufrufer zurückgegeben. Zum Erstellen des Adapters sind Objekt-Instanzen für *BranchResolver* und *PatchTransformer* nötig, welche dem *PatchIterator* übergeben werden. Zudem wird ein *FilterIterator* erstellt und dem *PatchIterator* als Iterator bereitgestellt. Das *FilterIterator*-Objekt erweitert wiederum eine Instanz der *ChangeSetIterator*-Klasse für das angegebene Respository.

Patch Iterator

Der *PatchIterator* implementiert das Interface *Iterator* für Patch-Objekte. Hierfür nutzt er Funktionalitäten der Klassen *BranchResolver* sowie *PatchTransformer* und verfügt über einen Iterator über alle verfügbaren *ChangeSet*-Objekte der TFS Umgebung. Die angegebenen Klassen werden alle zur Konstruktion eines *PatchIterator*-Objekts benötigt. Der *PatchIterator* fungiert hierbei als Dekorator für den übergebenen *ChangeSet*-Objekt-Iterator. Auf Anfrage, im Anwendungsfall durch Ausführen der *next()* Methode, wird ein *ChangeSet*-Objekt von dem internen *Iterator* abgefragt, durch den *PatchTransformer* in einen Patch transformiert, mit dem *BranchResolver* verarbeitet und als fertiges Patch-Objekt zurückgegeben. Wenn alle Elemente des internen Iterators durchlaufen sind, ist die Aufgabe des *PatchIterators* abgeschlossen und gibt auf Anfrage keine weiteren Elemente zurück . Eine genaue Erläuterung der Implementierung der Klasse und der Abhängigkeiten mit weiteren Iterator-Klassen wird in Abschnitt 5.1 durchgeführt.

Filter Iterator

Der *FilterIterator* dient als Erweiterung eines übergebenen Iterators über *ChangeSet*-Objekte um eine Filter-Funktionalität. Es wird weiterhin das Iterator-Interface implementiert, wodurch eine flexible Modifikation eines anderen Iterator-Objekts möglich ist. Die Klasse prüft vor Rückgabe eines Elements, ob alle geforderten Bedingungen von dem nächsten abzufragenden Objekt des internen Iterators erfüllt werden. Falls nicht, werden von dem zu erweiternden Iterator so lange Elemente abgefragt, bis ein valides Objekt identifiziert werden kann oder keine weiteren Elemente mehr vorhanden sind. In der TFS Adapter Komponente wird der *FilterIterator* als Dekorator für die *ChangeSetIterator*-Klasse eingesetzt und wird von dem *PatchIterator* genutzt. Es werden hierbei nur *ChangeSet*-Objekte mit mindestens einer Dateiänderung weitergeleitet und Dateisystemänderungen sowie *Merge*- und *Branching*-bezogene Änderungen ausgeschlossen.

ChangeSet Iterator

Die Klasse *ChangesetIterator* implementiert das Iterator Interface für *ChangeSet*-Objekte und hat das Ziel, alle verfügbaren TFS *ChangeSet*-Objekte über die Iterator-Schnittstelle zurückzugeben. Intern kommuniziert die Klasse mit dem Client, um die benötigten TFS Daten in Blöcken abzufragen und speichert diese als Iterator-Objekt zwischen. Auf Anfrage werden die Daten-Objekte einzeln von dem internen Iterator abgeladen und zurückgegeben. Wenn dieser keine Elemente mehr besitzt, werden neue Daten vom Client abgefragt. Da die Daten seitenweise vom Client angefordert werden müssen, speichert der *ChangesetIterator* intern, welche Seite als nächstes benötigt wird. Somit werden in sequenziellen Aufrufen alle verfügbaren Daten von dem Client abgefragt. Die Kommunikation mit dem Client sowie die Strategie zur Seitenabfrage werden hierbei gekapselt und lediglich über die Iterator-Schnittstelle werden die *ChangeSet*-Objekte nach außen angeboten.

Patch/FileChange Transformer

Die Aufgabe des *PatchTransformers* ist die Transformation von TFS Daten in Patches. Dazu werden die nötigen Informationen aus dem übergebenen Datenobjekt ausgelesen und in ein neues Patch-Objekt eingefügt. Zur Transformation der einzelnen *FileChange*-Objekte des Patches wird die Klasse *FileChangeTransformer* genutzt. Diese führt neben der Datentransformation eine Filterung von nicht relevanten Änderungen durch.

Branch Resolver

Um ein Patch einem Branch[1] der Entwicklungsumgebung zuordnen zu können, wird der *BranchResolver* eingesetzt. Dieser fragt von der Client Komponente alle verfügbaren Branches ab und speichert diese intern als *Map*-Object. Auf Anfrage wird ein übergebenes Patch-Objekt einem der verfügbaren Branches zugeordnet und zurückgegeben.

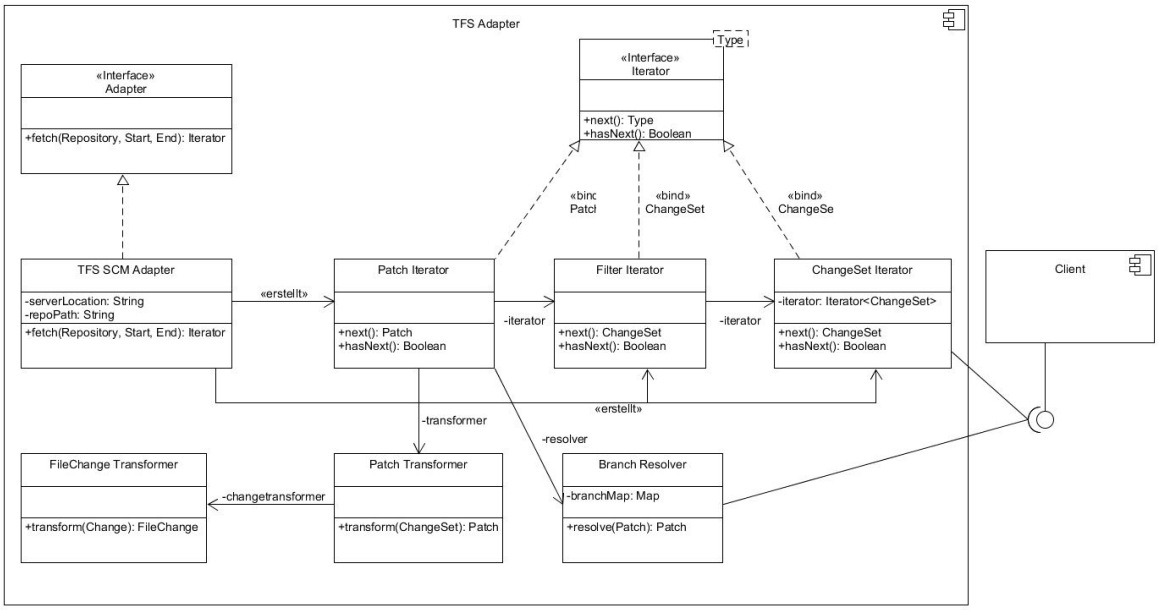

Abbildung 4.2: Klassen der TFS Adapter Komponente

4.1.2 Client

Die Aufgabe der Komponente *Client* ist es, benötigte Daten auf Anfrage von einem TFS Server zu laden und in einem verarbeitbaren Datenformat zurückzugeben. Die Client Komponente ist unabhängig von dem *TfsScmAdapter* und dient zur Kapselung der Kommunikation zwischen dem Adapter und der TFS REST API. Eine wichtige Rolle in der Kommunikation spielt das eingesetzte HTTP-Client Framework Retrofit[2]. Dieses und die in Abbildung 4.3 dargestellten Klassen der Komponente werden im Folgenden genauer beschrieben.

[1] Entwicklungspfad eines Repositories
[2] http://square.github.io/retrofit/

Retrofit

Für die Kommunikation mit der REST API wird das Open Source Framework Retrofit verwendet, da dieses eine übersichtliche Implementierung der benötigten Funktionalitäten bereitstellt. Zudem können über die angebotene Schnittstelle existierende Objekt-Parser integriert werden, die eine manuelle Datentransformation ersetzen. Da die abgefragten Daten im JSON[3]-Format von dem TFS Server zurückgegeben werden, ist die Transformation in temporäre Datenobjekte nötig. Hierfür wird der von Google veröffentlichte GSON-Converter[4] in den Retrofit-Client integriert. Zur Definition der Kommunikationsschnittstelle mit dem TFS Server muss dem Retrofit-Client ein Interface übergeben werden, das alle Pfade und Parameter der benötigten Aufrufe festlegt. Retrofit erstellt daraufhin eine Objekt-Instanz, welche die angegebene Schnittstelle implementiert. Dieses Objekt kann zur Durchführung der Aufrufe genutzt werden.

Service

Das Service Interface stellt den definierten Endpunkt zu der TFS REST API dar. Die genutzten Adresspfade sowie die dafür benötigten Parameter werden hierbei angegeben. Implementiert wird das Interface durch eine Retrofit-Instanz.

Client

Die Klasse *Client* ist der Einstiegspunkt der Komponente. Sie bietet die geforderten Funktionalitäten nach außen an und führt die Abfragen über eine Instanz der Retrofit Klasse aus, welche das Service Interface implementiert. Die Client Klasse selbst ist zustandslos und erhält alle benötigten Informationen zum Aufruf über die Methodenschnittstelle. Die Instanz der Retrofit-Klasse wird für jeden Aufruf neu über die statische Methode des *RetrofitBuilders* abgefragt. Nach dem erfolgreichen Ausführen einer Datenabfrage, werden die von Retrofit und GSON erstellten Datenobjekte zurückgegeben.

Retrofit Builder

Die Aufgabe des *RetrofitBuilders* ist es, für eine übergebene Serveradresse eine konfigurierte Retrofit-Instanz zurückzugeben. Hierfür wird von dem *Builder* eine statische Methode öffentlich angeboten. Um für gleiche Server-Adressen nicht

[3]JSON (JavaScript Object Notation) ist ein textbasiertes Datenformat zum Datenaustausch zwischen Anwendungen.

[4]https://github.com/google/gson

jedes Mal eine neue Klasse erstellen zu müssen, werden bereits erzeugte Objekte mit der zugehörigen Adresse in einem statischen *Map*-Objekt abgespeichert. Bei wiederholten Aufrufen der selben Adresse wird somit nur eine Retrofit-Instanz benötigt. Neben der Objektverwaltung ist die *RetrofitBuilder* Klasse für die korrekte Konfiguration des Retrofit-Objekts verantwortlich. Dazu zählen die Übergabe des Datenkonvertierers sowie die Eingliederung von Authentifizierungsmechanismen.

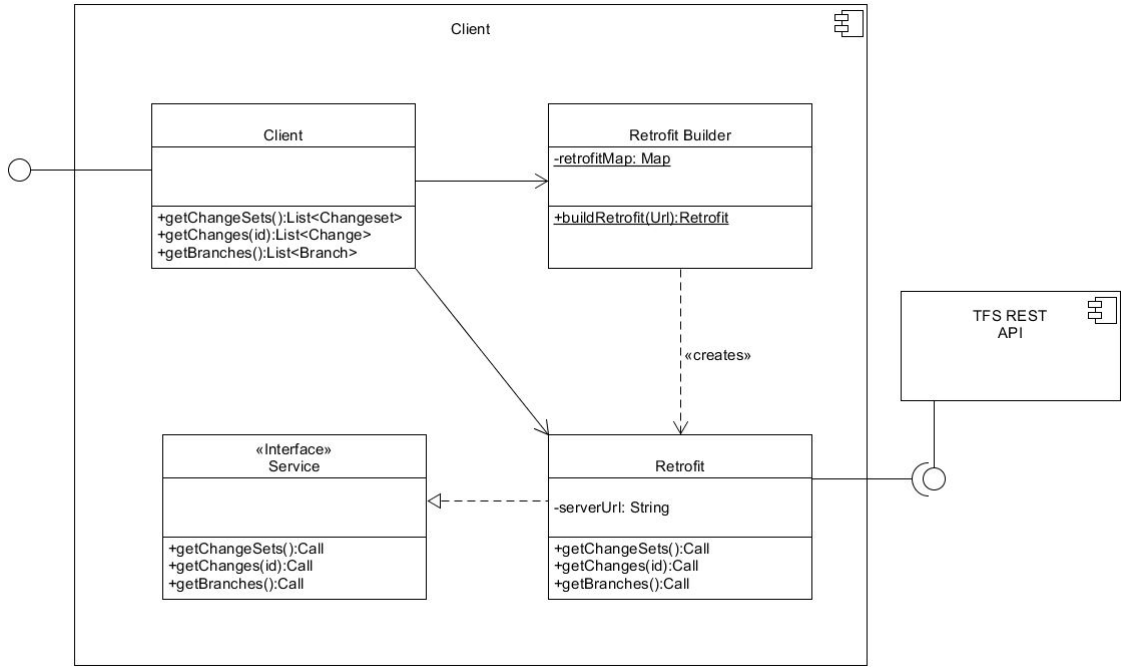

Abbildung 4.3: Klassen der Client Komponente

4.2 Verwendete Entwurfsstrategien

Während des Entwurfs neuer Softwarekomponenten entstehen häufig Probleme, die durch die Anwendung bewährter Strategien vermieden werden können. Bekannte Prinzipien für objektorientierte Entwürfe fördern bei richtiger Anwendung die Erweiterbarkeit sowie Wartbarkeit der Software. Im Folgenden wird das Prinzip der *Dependency Injection* sowie relevante Strategien von *S.O.L.I.D.* vorgestellt und die Umsetzung in der Entwicklung beschrieben.

Dependency Injection

Das Prinzip der *Dependency Injection* beschreibt die Übergabe einer Abhängigkeit an eine Klasse. Hierbei wird die Abhängigkeit per Konstruktor oder Setter-Methode in die Klasse eingefügt und nicht innerhalb der Klasse selbst erstellt. Dies fördert die Erweiterbarkeit, Testbarkeit, Modifizierbarkeit und Wiederverwendbarkeit des kompletten Systems (Yang, Tempero, & Melton, 2008).
Dieses Prinzip wurde während dem Entwurf des TFS Adapter und der zugehörigen Klassen berücksichtigt. So muss beispielsweise bei der Erstellung der TFS Adapter-Klasse der zu verwendende Client im Konstruktor angegeben werden. Falls Änderungen am Client durchgeführt werden müssen, ist der Adapter davon nicht betroffen, da die Initialisierung nicht in seinem Aufgabenbereich liegt.

S.O.L.I.D.

Die Abkürzung *S.O.L.I.D.* steht für 5 grundlegende Prinzipien bei dem objektorientierten Entwurf von Software Systemen, deren Ziel die Minimierung der Kopplung zwischen Klasssen sowie die Maximierung der Abstraktion ist (Wampler, 2007).

Single Responsibility Prinzip

Das Prinzip besagt, dass eine Klasse oder ein Modul nur für eine Funktionalität einer Software verantwortlich sein sollte. Das bedeutet, dass nur eine Änderung der Funktionalität ein Grund für die Anpassung einer Klasse ist (Martin, 2003). Angewendet wurde es beispielsweise bei der Transformation der Patch-Daten. Hierbei existiert ein Transformator für Patches und einer für Dateiänderungen. Somit sind die Funktionalitäten Patch-Transformation und Datei-Transformation unabhängig voneinander und bei Änderungen muss nur eine der Klassen angepasst werden. Dies fördert zudem die Übersichtlichkeit der einzelnen Klassen und ermöglicht Erweiterungen einfach einfügen zu können.

Open/Closed Prinzip

Das nächste Prinzip beschreibt die Eigenschaft einer Klasse, dass sie offen für Erweiterungen, aber verschlossen für Veränderungen sein sollte. Dies vermeidet eine Verkettung von Änderungen an abhängigen Modulen(Martin, 2003).
Während dem Entwurf des Clients wurde das Prinzip grundlegend berücksichtigt, da die bestehenden Funktionalitäten nicht verändert, jedoch eine Erweiterung in einzelnen Fällen durchgeführt wurde.

Liskovsche Substitutionsprinzip

Das *Liskovsche Substitutionsprinzip* besagt, dass eine Instanz einer abgeleiteten Klasse sich so verhalten muss, wie es von der Basisklasse erwartet wird. Dies garantiert, dass Operationen der Basisklasse, die auf abgeleiteten Klassen angewendet werden, korrekt durchgeführt werden können(Martin, 2003).
Zur Anwendung kommt das Prinzip bei der Implementierung des TFS Adapters, da dieser das vorgeschriebene Interface wie erwartet implementiert. Die Erstellung und Rückgabe eines eigenständigen Iterators, der zu Laufzeit die Objekte lädt und transformiert erfüllt hierbei die Vorgabe in selber Weise, wie ein fertig geladener Iterator der zurückgegeben wird. Diese Umsetzung ermöglicht eine problemlose Integration des Adapters in den bestehenden Crawl-Prozess, aufgrund des definierten Verhaltens. Ein nicht erwartetes Verhalten, wie eine fehlerhafte Sortierung der Elemente könnte hierbei zu Fehlerzuständen führen.

Interface Segregation Prinzip

Das *Interface-Segregation-Prinzip* dient der Aufteilung von zu großen Interfaces(Martin, 2003). Da keine Interfaces im Rahmen der Entwicklungsaufgaben erstellt oder verändert wurden, kam das Prinzip nicht zum Einsatz.

Dependency Inversion Prinzip

Das *Dependency-Inversion-Prinzip* unterscheidet sich trotz des ähnlichen Namens von dem bereits vorgestellten Prinzip der *Dependency Injection*. Ein zentraler Punkt ist die Einführung von Ebenen. So soll ein Modul einer höheren Ebene nicht abhängig von einem Modul einer niedrigeren Ebene sein, beide sollen jedoch von einer Abstraktion abhängen. Zudem soll eine Abstraktion nicht abhängig von Details sein, sondern die Details abhängig von der Abstraktion(Martin, 2003).
Eine konkrete Anwendung findet das Prinzip bei der Implementierung des *PatchIterators*, da diesem ein Iterator mit *FileChange*-Objekten bei Konstruktion übergeben wird. Bei der Abfrage der Objekte nutzt der *PatchIterator* die Funktionalitäten des Iterator Interfaces, wodurch er unabhängig von der konkreten Implementierung des Iterators ist und lediglich Bezug auf die fest definierte Abstraktion nimmt. Dies ermöglicht einen flexiblen Austausch der Iterator-Implementierung ohne die *PatchIterator*-Klasse verändern zu müssen. Im Anwendungsbeispiel ist es möglich einen *FileChangeIterator* oder einen *FilterIterator* zu übergeben.

5 Implementierungsdetails

Im nachfolgenden Kapitel werden spezielle Implementierungsentscheidungen vorgestellt und gegen alternative Umsetzungsmöglichkeiten abgewägt. Zudem wird kurz auf die Relevanz für einzelne Performanceaspekte eingegangen.

5.1 Iteratorimplementierungen

Bei der Entwicklung des TFS Adapters und der Optimierung des Git Adapters wurden in beiden Fällen Iteratoren entwickelt, welche auf die zugrundeliegende Patch-Verarbeitung angepasst sind. Zunächst wird auf die Implementierung des TFS *PatchIterators* und der abhängigen Module eingegangen. Anschließend wird die Optimierung des bestehenden *PatchIterators* für den Git Adapter vorgestellt.

TFS

Im ersten Entwurf des TFS Adapters wurden die Funktionalitäten der in Abschnitt 4.1 vorgestellten Klassen *ChangeSetIterator*, *FilterIterator* und *PatchIterator* von der Klasse *TfsScmAdapter* implementiert. Eine Problematik dieser Umsetzung ist die nicht abgegrenzte Aufgabenverteilung der Module. Während bei anderen Klassen eine eindeutige Funktionalität implementiert wird, ist der *TfsScmAdapter* für eine Vielzahl an Aufgaben verantwortlich. Hierzu zählt die Initialisierung benötigter Objekte, das Laden der SCM Daten von der *Client* Komponente, das Anstoßen der Transformation sowie die Filterung und Sortierung der Patch-Objekte. Dies widerspricht auch dem in Abschnitt 4.2 vorgestellten Prinzip der *Seperation of Concerns*, da hierbei keine klare Aufgabenabgrenzung stattfindet. Eine weitere Problematik der Umsetzung ist die Zwischenspeicherung der Patch-Objekte. Da keine Iteratorlogik vorhanden ist, jedoch ein Iterator zurückgegeben werden muss, werden alle verfügbaren Daten geladen und in einer Liste zwischengespeichert. Diese wird abschließend in einen Iterator transformiert und zurückgegeben. Somit können unkontrollierbar viele Objekte in die Liste ge-

laden werden, was im Worst-Case-Szenario zu einer *Out-Of-Memory-Exception* der *Java Virtual Machine* (JVM) führen kann.

Die überarbeitete Umsetzung besitzt die in Abschnitt 4.1 vorgestellte Aufteilung der Funktionalitäten auf einzelne Klassen. Das Laden der SCM Daten von der *Client* Komponente wird auf den *ChangeSetIterator* ausgelagert. Das Filtern der *ChangeSet*-Objekte wird nun von der *FilterIterator*-Klasse implementiert und die Transformation der Daten wird von dem *PatchIterator* auf Anfrage angestoßen. Zudem implementiert der *PatchIterator* das Iterator Interface, wodurch er als Rückgabe-Objekt für die *fetch()* Methode des TFS Adapters eingesetzt werden kann. Lediglich die Initialisierung der benötigten Iterator-Objekte ist weiterhin die Aufgabe des *TfsScmAdapters*. Besonders hervorzuheben an dieser Lösung sind die kaskadierenden Iteratoren. Der *PatchIterator* dekoriert den *FilterIterator* und der *FilterIterator* dekoriert wiederum den *FileChangeIterator*. Durch diese Neuverteilung konnten klare Aufgabenbereiche für die einzelnen Klassen definiert und die Struktur der Software optimiert werden. Ein exemplarischer Ablauf der Aufrufe wird in Abbildung 5.1 angegeben.
Ein weiterer Vorteil der Iteratorimplementierung ist das dynamische Laden der SCM Daten und die damit optimierte Arbeitsspeicherauslastung. Die SCM Daten werden in Blöcken einer fester Größe n von dem *ChangeSetIterator* geladen und intern als Iterator zwischengespeichert. Die Daten werden auf Anfrage einzeln vom *ChangeSetIterator* abgefragt, gefiltert, transformiert und an den Aufrufer zurückgegeben.

Wenn alle Objekte im *ChangeSetIterator* abgearbeitet sind, werden neue Daten vom Client geladen und alte freigegeben. Durch diesen Ansatz werden maximal n Objekte zwischengespeichert, wodurch eine gleichbleibende Arbeitsspeicherauslastung garantiert werden kann, unabhängig von der Anzahl der verfügbaren Objekte. Als Blockgröße n wurde die von der TFS REST API als Standardwert angegebene Zahl 100 festgelegt. Die genauen Ergebnisse auf die Programm-Performance werden in Abschnitt 6.3 vorgestellt.

Git

Die Optimierung der bestehenden *PatchIterator*-Klasse des Git Adapters wurde nach der initialen Performanceanalyse durchgeführt, da hierbei festgestellt wurde, dass ein großer Teil der Programm-Laufzeit für die Datentransformation von Git benötigt wird. Der Aufbau des Iterators ist ähnlich zu dem im vorangehenden Absatz vorgestellten TFS *PatchIterators*. Es wird ein interner Iterator mit *Commit*-Objekten verwendet, die auf Anfrage zu Patch-Objekten verarbeitet und zurückgegeben werden. Der interne Iterator wird einmalig initialisiert und enthält alle verfügbaren *Commit*-Objekte. Um eine abgegrenzte Aufgabenverteilung zu fördern, wurde die Initialisierung des Iterators in ein eigenes Objekt verlagert.

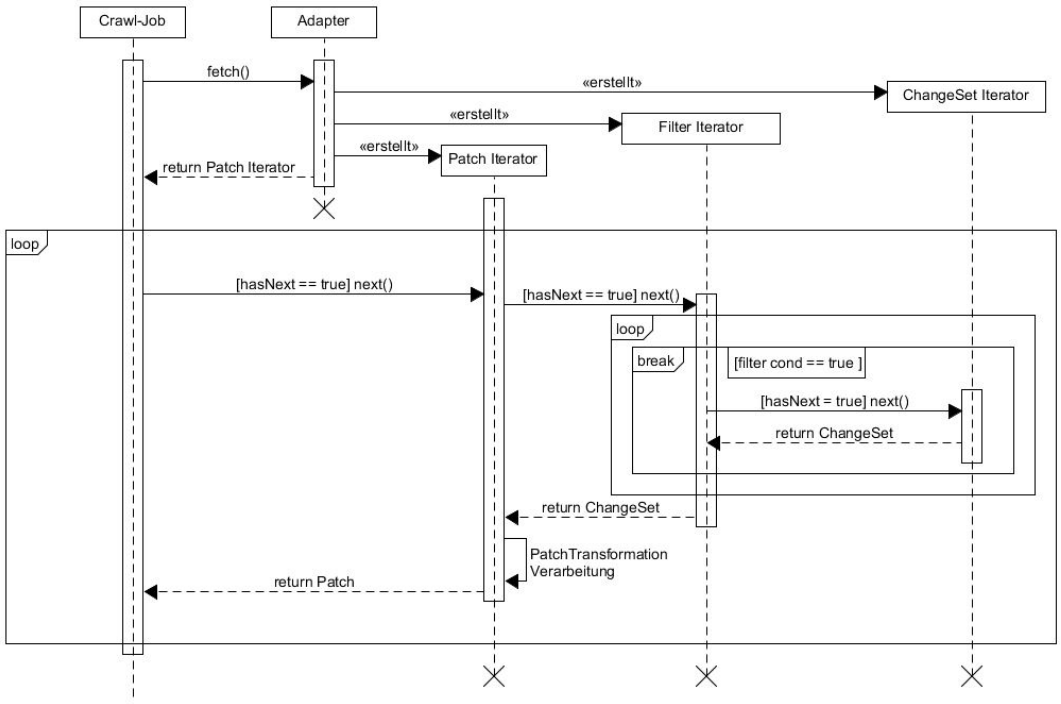

Abbildung 5.1: Abbildung der exemplarischen Aufrufabfolge zwischen den Klassen der *TFS Adapter* Komponente. Zur übersichtlicheren Darstellung wird der Kontrollaufruf *hasNext()* nicht mit angegeben, sondern als Bedingung definiert. Die Patch-Transformation durch den *PatchTransformer* sowie die Patch-Verarbeitung durch den *BranchResolver* werden zur Vereinfachung der Darstellung als eine Operation des *PatchIterators* abgebildet. Die Schleife für die Elementabfragen des *FilterIterators* von dem *ChangeSetIterator* wird abgebrochen, falls die Filter-Bedingung (filter-cond) erfüllt ist.

Damit wurde eine identische Aufrufstruktur wie bei dem TFS *PatchIterator* erzeugt, da der interne Iterator ebenfalls von einem Filter-Iterator dekoriert wird. Die Abfragedauer für ein Patch-Objekt von dem Git *PatchIterator* ist in der Ausgangssituation die Zeit, die zur Transformation des *Commit*-Objekts zu einem Patch-Objekt benötigt wird. Die Rückgabezeit des *Commit*-Objekts von dem internen Iterator kann hierbei vernachlässigt werden. Da keine Optimierungsmöglichkeit für den Transformationsalgorithmus identifiziert werden konnte, wurde nach anderen Optionen gesucht, um die Abfragedauer zu verkürzen. Der entwickelte Ansatz hierfür nutzt eine Queue[1] mit Arbeitselementen sowie *Future*-Objekte des *java.util.concurrent* Pakets, um eine dynamische Aufgabenverteilung zu ermöglichen. *Future*-Objekte dienen als definierter Rückgabeort für die Ergebnisse einer asynchron ausgeführten Aufgabe. Auf diese Weise können eine Vielzahl an Aufträgen an eine Ausführungsinstanz übergeben und auf die

[1] *Queue* - englische Bezeichnung für Warteschlange.

Ergebnisrückgabe gewartet werden. Im Anwendungsfall der Arbeit wird für jedes *Commit*-Objekt ein *Future*-Objekt mit dem Auftrag der Transformation zu einem Patch erstellt. Das erwartete Ergebnis des *Future*-Objekts ist somit ein Patch-Objekt, das von der *PatchIterator*-Klasse zurückgegeben werden kann. Zur Lastverteilung werden die *Future*-Objekte in einer Queue mit einer begrenzten Größe gespeichert, die von einem *Executer Service* mit integriertem *Thread-Pool* asynchron abgearbeitet werden. Der Main-Thread greift auf Anfrage auf das erste Element der Queue zu und gibt es, wenn fertig verarbeitet, zurück. Damit wird es aus der Queue gelöscht und ein neues Element rückt an die erste Stelle. Wenn die Queue zur Hälfte geleert ist, wird sie mit neuen *Future*-Objekten gefüllt und zur Bearbeitung durch den *Executer-Service* freigegeben. Die Aufgabe des *PatchIterators* ist abgeschlossen, wenn alle Elemente des *Commmit*-Iterators in Patch-Objekte transformiert und zurückgegeben wurden. Ein exemplarischer Ablauf wird in Abbildung 5.2 für eine Queue mit der Größe 4 und einem *Thread Pool* mit zwei Threads dargestellt.

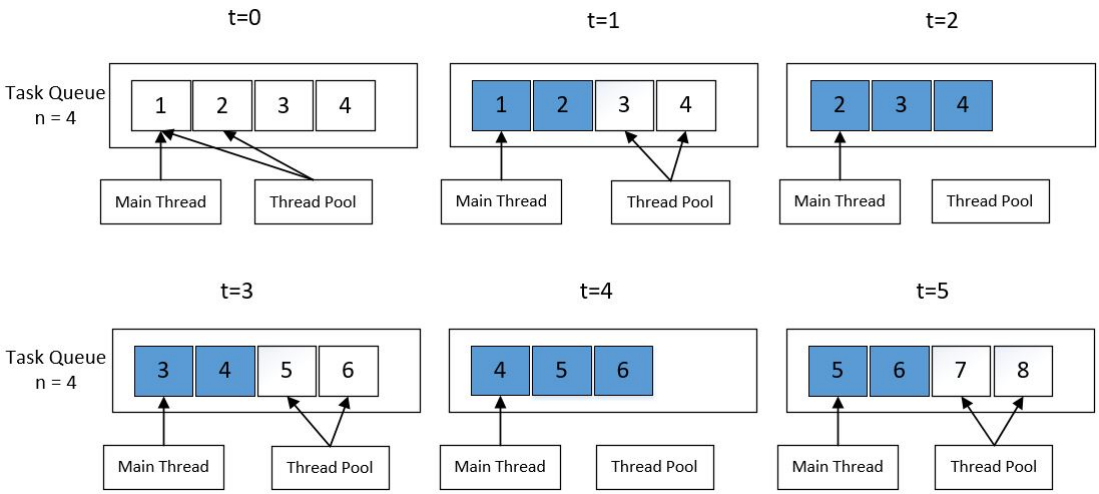

Abbildung 5.2: Darstellung des Verhaltens der entwickelten Queue für den *PatchIterator* des Git Adapters. Farblose Queue-Elemente stehen für erstellte Aufgabenobjekte für den *Thread Pool*. Farbige Elemente stehen für verarbeitete Elemente, die von dem *Main-Thread* genutzt und damit aus der Queue entfernt werden können. Das Zeitintervall repräsentiert die Dauer, die zur Verarbeitung eines einzelnen Elements benötigt wird. Durch die Abbildung soll verdeutlicht werden, dass der Main-Thread nur im ersten Zeitabschnitt auf ein Element warten muss und in den nachfolgenden Schritten keine weitere Wartezeiten benötigt werden.

Der Performancegewinn der Implementierung entsteht durch die Parallelisierung der zeitintensiven Transformation von *Commit*- zu Patch-Objekt. Für die Anwendung wurde die Größe der Queue auf 100 Elemente festgelegt und zwei Threads

dem *Executer Service* zu Verfügung gestellt. Mit dieser Konfiguration werden die Elemente schneller von dem *Executer Service* transformiert, als sie vom Main-Thread verarbeitet werden können. Somit wird die Wartezeit für die Transformation auf die Erstellung der *Future*-Objekte sowie die Verarbeitung des ersten Objekts reduziert. Der zugrundeliegende Zweck der Queue ist die Kontrolle der Arbeitsspeicher- sowie der CPU-Auslastung. Über die feste Größe der Queue wird die Anzahl an zwischengespeicherten Elementen limitiert und aufgrund der blockweisen Auffüllung die Arbeitslast verteilt. Die gemessenen Auswirkungen auf die Performance des Crawl-Prozesses wird in Abschnitt 6.3 genauer erläutert.

5.2 Paginationimplementierung

Für die Kommunikation mit der TFS Rest API wird die in Abschnitt 4.1 vorgestellte Komponente *Client* eingesetzt. Die Schnittstellendefinition der REST API wird auf einer öffentlichen Microsft Webseite ("Microsoft Team Foundation Server REST API Documentation", 2017) angegeben. Eine häufig eingesetzte Methode zur Aufteilung von großen Anfragen auf Listenelemente ist eine seitenweise Rückgabe der Elemente, die auch Pagiantion[2] genannt wird. Hierbei werden die Parameter *top* und *skip* bei dem HTTP-Aufruf mitgegeben. *Top* steht für die Anzahl an Elementen die zurückgegeben werden sollen und *skip* steht für die Anzahl an zu überspringenden Elementen. Durch die Kombination der Parameter können beliebig viele Elemente an jeder Stelle der Elementliste abgefragt werden. Im Praxisfall wird ein fester *top* Wert definiert und der mit Null initialisierte *skip* Wert nach jedem Aufruf um den *top* Wert erhöht, bis alle Elemente abgefragt sind. Die entwickelte Client-Klasse nutzt diese Praktik, um eine Lastverteilung für das Programm sowie den angesprochenen TFS Server zu bieten. Die Seitengröße *top* wird intern fest definiert und die Schnittstelle benötigt beim Aufruf lediglich die Seitenzahl als Parameter. Die Umwandlung in *skip* und *top* Werte wird hierbei nach außen gekapselt.

5.3 Parallelisierung des Crawlers

Die Änderung mit den größten Auswirkungen auf die Performance des Crawl-Prozesses, ist die Parallelisierung der *Crawl-Jobs*. Bislang wurden die einzelnen *Crawl-Jobs* sequenziell gestartet, was bei großen Mengen an Repositories zu langen Laufzeiten geführt hat. Dies liegt unter anderem daran, dass die durch Ein- und Ausgabe bedingten Wartezeiten nicht genutzt wurden und somit Optimierungspotential besteht. Für die Parallelisierung wurden die bestehenden

[2]Pagination - englisch für Nummerierung

Funktionalitäten des *java.util.concurrent* Pakets genutzt, da diese übersichtliche Schnittstellen zur Durchführung von Parallelisierungen anbieten. In der neuen Umsetzung werden zu verarbeitende *Crawl-Job* Aufrufe an einen existierenden *Thread Pool* übergeben, der diese auf interne Arbeiter-Threads verteilt. Als konkrete *Thread Pool* Implementierung wurde der standardmäßige *ForkJoinPool* verwendet, da dieser die Anzahl an verwendeten Threads an die Anzahl der vorhandenen logischen Prozessoren des Host-Systems anpasst. Somit wird die Leistungsfähigkeit des Systems bei wechselnden Hosts dynamisch genutzt. Bei der Implementierung der Parallelisierung mussten neben der Änderung an dem *Crawl-Job* Aufruf keine weiteren Änderungen an anderen Modulen durchgeführt werden, da diese bereits threadsicher programmiert wurden. Das Ergebnis der Parallelisierung aus Sicht der Performanceanalyse ist eine Reduktion der Laufzeit, jedoch auch eine Erhöhung der CPU- und Arbeitsspeicherauslastung. Diese Veränderungen werden in Abschnitt 6.3 genauer erläutert.

5.4 Optimierung des Speichervorgangs

Die Persistierung der Patch-Objekte ist eine der zentralen Aufgaben des Crawl-Prozesses, die von der Klasse *Crawl-Job* implementiert wird. Diese nutzt die in der *Common* Komponente definierten Datenzugriffsobjekte (DAO), um die einzelnen Daten-Objekte in der Datenbank zu speichern. Diese wiederum greifen auf das Persitenz-Framework Hibernate[3] zurück, das vordefinierte Persistierungs-Funktionalitäten bereitstellt. Der bislang verwendete Speicheralgorithmus in der *Crawl-Job* Klasse wird in Algorithmus 1 vereinfacht dargestellt.

Algorithmus 1 : Der bislang verwendete Speicheralgorithmus

input : Iterator *Patches* über alle verfügbaren Patch-Objekte
output : -

```
begin
    foreach patch ∈ Patches do
        fileChanges := patch.fileChanges;
        foreach fileChange ∈ fileChanges do
            /* Eine Transaktion pro Element                    */
            speicher(fileChange);
        end
        speicher(patch);
    end
end
```

[3] urlhttp://hibernate.org/

Von besonderer Relevanz ist hierbei, dass für jedes *FileChange*-Objekt von jedem Patch-Objekt die Methode *speicher()* des zugehörigen DAOs aufgerufen wird. Da jedem Patch beliebig viele *FileChange*-Objekte zugeordnet werden können, kann hierbei eine hohe Anzahl an Speicheraufrufen erzeugt werden. Die Implementierung der Speichermethode in den zugrundeliegenden Klassen öffnet für jeden Aufruf eine eigene Transaktion, speichert das übergebene Objekt und schließt die Transaktion. Für eine lokale Datenbank ist die Übertragungsdauer sehr gering. Wird jedoch eine Datenbank auf einem externen Server innerhalb eines Netzwerks angesprochen erhöht sich die Latenz stark und vervielfacht die Übertragungsdauer. Somit können hohe Ausführungszeiten für das Abspeichern der Datensätze entstehen.

Zur Reduktion der benötigten Transaktionen wird im Persistenzmodul der *Common* Komponente eine neue Funktionalität implementiert. Dieser wird statt einem einzelnen Objekt eine Liste mit zu persistierenden Objekten übergeben. Diese werden zu Blöcken einer definierten Größe zusammengefasst und an die Datenbank gesendet. Zudem wird in der Hibernate Konfigurationsdatei das Senden in Blöcken aktiviert und die Blockgröße definiert. Nach Anpassung der DAOs kann der überarbeitete Algorithmus, wie in Algorithmus 2 dargestellt, ausgeführt werden.

Algorithmus 2 : Der optimierte Speicheralgorithmus

input : Iterator *Patches* über alle verfügbaren Patch-Objekte
input : Integer n der die Blockgröße angibt
output : -

```
begin
    foreach patch ∈ Patches do
        /* Eine Transaktion pro n Elementen                    */
        speicherAlle(patch.fileChanges);
        speicher(patch);
    end
end
```

Als Standardwert für die Blockgröße der Übertragung wurde nach Recherche der Wert 50 festgelegt, da hierbei die Speicherauslastung nicht zu Fehlern führen kann und für den Regelfall nur eine Transaktion zum Speichern alle *FileChanges* nötig ist. Die Auswirkungen auf die Laufzeit während der Crawl-Run Ausführung wird in Kapitel 6.3 präsentiert.

Eine weitere Optimierungsmöglichkeit für den Speicheralgorithmus bietet sich durch die Aufhebung von typsicherem Speichern. So könnte die Anzahl an nötigen Transaktionen im Vergleich zu der Ausgangsanzahl bis um das n-fache reduziert

werden, wobei n die definierte Blockgröße ist. Jedoch würde damit die Kontrolle über die Speichervorgänge abgegeben werden und bei Fehlerfällen könnte ein invalider Datenbankzustand erreicht werden. Dies kann bei dem Durchführen eines neuen *Crawl-Runs* zu Überschneidungen führen und somit zu Redundanzen in der Datenbank. Da dies vermieden werden muss, wurde der Ansatz nicht umgesetzt.

6 Evaluation und Optimierung der Software

Um die entwickelte Software gegenüber den Anforderungen evaluieren zu können, wurde eine Evaluationsstrategie festgelegt, die in Unterabschnitt 6.1 erläutert wird. Im Anschluss werden in 6.2 die Messziele der Performanceevaluation vorgestellt. Abschließend werden die Optimierungsergebnisse in Unterabschnitt 6.3 für die jeweiligen Adapter präsentiert.

6.1 Evaluationsstrategie

6.1.1 Evaluierungsziele

Das Ziel der Evaluation der Software ist unter anderem die Sicherstellung einer korrekten Implementierung aller benötigten Funktionalitäten. Die verwendete Strategie hierfür wird bei dem Evaluierungsvorgehen in Unterabschnitt 6.1.2 genauer erläutert.

Das zweite Evaluierungsziel ist die Überprüfung von Performanceaspekten während der Ausführung der Software. Da das Ziel der Arbeit eine Optimierung dieser Aspekte ist, wird das Programm iterativ verbessert und nach jeder Überarbeitung erneut gemessen. Das Vorgehen bei der Messung, die Erstellung der Testszenarien sowie der Aufbau des Messumgebung werden in den nachfolgenden Kapiteln ausgeführt.

6.1.2 Vorgehen

Die Evaluation der Software gegenüber den funktionalen Anforderungen für den TFS Adapter, wird über die Implementierung von Tests sichergestellt. Eine grundlegende Unterscheidung besteht zwischen Modul-Tests, die eine konkrete Funktionalität einer Klasse überprüfen, und Integrationstests, welche die korrekte Zu-

sammenarbeit der entwickelten Klassen und Komponenten in einem System validieren.
Modul-Tests, auch *Unit*-Tests genannt, fokussieren sich auf die Überprüfung einer speziellen Klasse. Abhängigkeit sowie Randbedingungen werden mit *Mock*-Objekten[1] ersetzt oder mit Standardwerten angegeben. Da für alle entwickelten Klassen Modul-Tests erstellt und diese fehlerfrei abgeschlossen wurden, kann die Erfüllung aller funktionalen Anforderungen bestätigt werden.
Die Entwicklung von Integrationstests benötigt ein beispielhaftes System, an der die korrekte Zusammenarbeit aller eingesetzten Komponenten überprüft werden kann. Hierfür wurde eine Testumgebung erstellt, die eine konfigurierte Datenbank sowie eine TFS Komponente mit angelegten Datensätzen und zugänglicher REST-API enthält. Die durch einen ausgeführten *Craw-Run* erhaltenen Daten werden mit erwarteten Werten verglichen. Somit kann eine fehlerfreie Integration der einzelnen Komponenten sichergestellt werden. Da diese Tests ebenfalls erfolgreich durchlaufen wurden, können die folgenden Anforderung an die Entwicklung des TFS Adapters als erfüllt betrachtet werden.

- Entwicklung eines Adapters zur Datenextraktion von TFS SCM Systemen
- Verwendung der TFS REST API
- Implementierung des bestehenden Adapter Interfaces

Die letzte zu erfüllende Anforderung ist die „Performanceanalyse und Optimierung des entwickelten TFS Adapters, des Git Adapters und des Crawl-Prozesses". Hierfür werden zunächst initiale Messungen erstellt, die für beide Adapter getrennt durchgeführt werden. Anschließend wird mit den in Anhang C beschriebenen Programmen nach Optimierungsoptionen gesucht. Diese können „Flaschenhälse" im Crawl-Prozess oder sehr ressourcenintensive Codestellen sein, welche verbessert werden müssen. Die Optimierung des Codes wird iterativ durchgeführt, um für jeden Zustand neue Messungen erstellen und mit den vorangegangen Messungen vergleichen zu können.

Zur Durchführung der benötigten *Crawl-Runs* und um die relevanten Stellen des Crawl-Prozesses auf verschiedene Performanceaspekte vermessen zu können, wurde eine Mess-Applikation entwickelt. Diese Anwendung erstellt ein zur Performancemessung angepasstes *Plugin* und startet den *Crawl-Run*. Das verwendete *Plugin* wird je nach gewünschtem Adapter und Szenario ausgetauscht. Getestet wird hierbei der Crawl-Prozess, ausgewählte Prozessorschritte des *Plugins* und der Adapter. Ausgeschlossen werden bei den Prozessorschritten die Personenauflösung über LDAP sowie die Generierung von Organisations-Elementen, da für diese die Erstellung einer umfangreichen Testumgebung notwendig ist.

[1] Als Mock-Objekt wird ein Platzhalter-Objekt bezeichnet, das alle benötigten Funktionalitäten und Eigenschaften einer Klasse bietet.

6.1.3 Evaluationsszenarien

Bei der Erstellung der Evaluationsszenarien wurde darauf geachtet, dass gleichbleibende Bedingungen für wiederholte Messungen bestehen, um vergleichbare Messwerte zu erhalten. Das grundlegende Szenario ist die Durchführung eines *Crawl-Runs* auf einem Computer mit lokaler Datenbank und externen Repositories, die abgefragt werden sollen.

Szenario 1

Das erste Szenario stellt eine übliche Situation bei der Durchführung eines *Crawl-Runs* dar. Es werden 10 Repositories von kleiner bis mittlerer Größe abfragt, die akkumuliert 12.769 *Commits* und 78.541 *FileChanges* besitzen. Diese Anzahl der Commits liegt zwischen 303 und 2.279 pro Repository. Da dieses Szenario in einem Zeitrahmen von wenigen Minuten durchgeführt werden kann, wird es hauptsächlich zur Analyse der Metriken verwendet.

Szenario 2

Das zweite Szenario dient als Lasttest für die einzelnen Adapter. Hierbei werden lediglich drei Repositories eingesetzt, jedoch besitzen diese eine erhöhte Datenmenge. Zudem werden für ein *Repsootory* 22.000 Commits zurückgegeben. Dieses Szenario wird zu Überprüfung der Ressourcenauslastung während der Durchführung verwendet und stellt nicht den erwarteten Fall dar. Deshalb wird es in Abschnitt 6.3 zur Überprüfung einzelner Problematiken herangezogen.

6.1.4 Aufbau der Messumgebung

Ausgeführt werden die Performance-Tests auf einem Acer Aspire VN7-591G-50UG mit Windows 10 als Betriebssystem. Die Eigenschaften der verwendeten Hardware werden in Tabelle 6.1 ausgelistet. Um während der Laufzeit eine minimale Auslastung durch das Host-System und dadurch gleichbleibende Bedingungen für nachfolgende Tests zu ermöglichen, wurden verschiedene Hintergrunddienste während der Messungen deaktiviert. Zu diesen zählen der Virenscanner, der in Windows integrierte Indizierungsdienst als auch automatische Systemupdates. Durch diese Maßnahmen konnte eine durchschnittliche CPU-Auslastung von unter 2% durch nicht test-relevante Prozesse erreicht werden. Die für Downloads relevante Netzwerkbandbreite beträgt maximale 32,7 MBit pro Sekunde.

Kategorie	Hardwareeigenschaft
Prozessor	Intel Core i5-4210H
Prozessorkerne	2
Logische Prozessoren	4
Max. Taktfrequenz	3,5 GHz
Festplatte	Samsung SSD 850 EVO 500 GB
Arbeitsspeicher	8GB DDR3

Tabelle 6.1: Übersicht der verwendeten Hardware

Für Durchläufe mit dem Git Adapter werden 13 öffentliche Repositories von Github[2] verwendet, um eine repräsentative Last zu erzeugen und vergleichbare prozentuale Laufzeiten für die Methodenausführung zu erhalten. Diese werden in Anhang B aufgelistet. Bei der Auswahl der Repositories wurde darauf geachtet, dass die Dateigröße nicht 20MB übersteigt, um die Downloaddauer gering zu halten sowie, dass genug Commits vorhanden sind, die zu verarbeiten sind. Ausnahme sind hierbei drei Repositories, die zu Lasttests in Szenario 2 eingesetzt werden, welche eine Datengröße von mehr als 200 MB besitzen. Zudem wurde überprüft, ob das Erstelldatum vor 2015 liegt, da das *CrawlRun* Abfragedatum auf den 11.06.2016 gesetzt wird, um die Abfrage der relevanten Commits einheitlich zu halten.

Da keine öffentlichen Repositories für den TFS Adapter Test zur Verfügung stehen, werden über einen simulierten TFS Server Daten per HTTP deterministisch zurückgegeben. Somit können auch für diesen Adapter realistische Nutzdaten in dem benötigten Umfang generiert werden. Um vergleichbare Szenarien wie für den Git Adapter zu erzeugen, wird der Server so eingestellt, dass für die verfügbaren Repositories ähnliche *Commit* und *FileChange* Mengen bereitgestellt werden. Ausgeführt wird der simulierte Server auf einem anderen System im gleichen Netzwerk wie der Laptop, um die Performance bei der Durchführung von *Crawl-Runs* nicht zu beeinflussen.

6.2 Metriken der Performanceevaluation

Vor der Durchführung der Performance-Optimierung werden Messgrößen benötigt an denen aufgezeigt werden kann, wie die Änderungen am Programmcode das Verhalten des zu optimierenden Systems beeinflusst haben. Hierfür werden Metriken genutzt, die für jeden Programmzustand neu erstellt werden. Zunächst

[2]https://github.com/

werden in Abschnitt 6.2.1 die Ergebnisse der Literaturrecherche nach bekannten Performancemetriken für Java Applikationen präsentiert. Die gefundenen Metriken, welche in primäre und sekundäre Metriken untergliedert wurden, werden im Anschluss in Abschnitt 6.2.2 und 6.2.3 genauer erläutert. Abschließend werden die Metriken in Tabelle 6.2 dargestellt.

6.2.1 Bekannte Performancemetriken

Um auf etablierte Performancemetriken zurückgreifen zu können, wurde eine Literaturrecherche zu dem Thema „java application performance metrics" durchgeführt. Ein Großteil der hierzu relevanten Veröffentlichungen geben Metriken an, die auf eine spezielle Problemsituation angepasst und demnach nicht für die Anwendung in dieser Arbeit geeignet sind. Durch weitere Recherche konnten Metriken identifiziert werden, die häufig bei Performance-Überprüfungen eingesetzt werden und im Rahmen dieser Arbeit verwendet werden können.
Die erste Größe ist die durchschnittliche Laufzeit eines Programms. Diese wird von Georges, Buytaert, und Eeckhout (2007) in der Veröffentlichung über Untersuchungsmethoden von Java Programmen ausführlich diskutiert. Hierbei werden die verschiedenen Ansätze zur Auswertung von Performancemessungen gegenübergestellt und an einem Anwendungsbeispiel verdeutlicht. Grundsätzlich wird zwischen der Messung von „Startup performance" und „Steady-state performance" unterschieden. Das Ziel der „Startup performance" Messung ist die Ermittlung der Zeitdauer, wie schnell ein kurzläufiges Programm von einer Java Virtual Machine (JVM) ausgeführt werden kann. Bei „Steady-state performance" wird dagegen ein langläufiges Programm untersucht, das nach einer gewissen Zeitspanne einen gleichbleibenden Zustand erreicht. Auf diesem Zustand werden anschließend die Messungen zu verschiedenen Metriken durchgeführt. Da die Crawl-Applikation der CMSuite der „Startup performance" zuzuordnen ist, wurden die Ausführungsbeschreibungen hierzu genauer untersucht. Zur Ermittlung der durchschnittlichen Laufzeit werden mehrere Iterationen der Messung durchgeführt und nach jeder Iteration das zugehörige Konfidenzintervall berechnet, bis dieses maximal 2% vom Durchschnittswert abweicht. Zudem wird angegeben, das die erste Ausführung ignoriert wurde, da hierbei verschiedene Ladeoperationen vollzogen werden, die bei den nachfolgenden Ausführungen nicht auftreten.
Eine weitere Messgröße ist die Arbeitsspeicherauslastung eines Programms während der Ausführung. Diese wird in verschiedenen Veröffentlichen (Ryan & Rossi, 2005) (Czajkowski & von Eicken, 1998) (Dufour, Driesen, Hendren, & Verbrugge, 2003) aufgegriffen, um die Performance eines Systems zu bewerten. Bei Dufour u. a. (2003) wird die durchschnittliche Speicherallokation pro 1000 Bytecode Ausführungen gemessen. Ryan und Rossi (2005) verwenden den aggregierten Speicherverbrauch des Host-Systems als Metrik.
Die letzte identifizierte Metrik, die für diese Arbeit anwendbar ist, ist der CPU

Verbrauch während der Programmausführung. Diese wird von Ryan und Rossi (2005) als aggregierter Prozessorverbrauch angegeben. In der Veröffentlichung von Wood, Cherkasova, Ozonat, und Shenoy (2008) über das Profiling des Ressourcenverbrauchs bei virtualisierten Anwendungen wird der CPU Verbrauch des Kernels sowie des „User Space" in Prozent vermessen.
Die Anwendung der Metriken bei der Performance-Optimierung des Crawl-Prozesses wird in den folgenden Abschnitten erläutert.

6.2.2 Primäre Metriken

Als primäre Metriken werden in dieser Arbeit Metriken bezeichnet, die eine direkte Auswirkung auf den Nutzer während der Ausführung des Crawl-Prozesses haben. Dies ist insbesondere die Ausführungsdauer, da der Nutzer weniger Zeit zur Datenextraktion benötigt. Das Ziel bei der Optimierung des Crawl-Prozesses sowie der Adapter ist eine Minimierung dieser.

Laufzeit

Um die Metrik Laufzeit zu bestimmen, werden die Messmethoden der in Abschnitt 6.2.1 vorgestellten Veröffentlichung zur Laufzeitmessung berücksichtigt. Zur Ermittlung eines Durchschnittswertes wurden 10 Durchläufe der Messapplikation durchgeführt und die Ergebnisswerte in Millisekunden abgespeichert. Zur Überprüfung der Qualität des Durchschnitts wurde das in (Georges u. a., 2007) beschriebene 95%-Konfidenzintervall berechnet und die Abweichung zum Durchschnittswert ermittelt. Da die Abweichung bei 10 Iterationen immer unter 5% lag, wurde die Anzahl an Iterationen nicht erhöht. Zudem wurde wie in Abschnitt 6.2.1 erläutert die erste Iteration ausgeschlossen, um vergleichbare Messungen zu erhalten.

6.2.3 Sekundäre Metriken

Als sekundäre Metriken werden in dieser Arbeit Metriken bezeichnet, die keine direkte Auswirkung auf den Nutzer während der Ausführung haben, sondern auf die Ressourcen des ausführenden Systems. Zu diesen zählen die in Abschnitt 6.2.1 beschrieben Metriken Arbeitsspeicherauslastung und CPU-Auslastung. Zudem wird die Systemressource Festplattenspeicher mit einbezogen. Das Optimierungsziel der sekundären Metriken ist keine unbedingte Minimierung, sondern ein kontrollierter Einsatz der zugehörigen Ressource. Das bedeutet, dass eine höhere Ressourcenauslastung bei geringerer Laufzeit nicht unbedingt negativ ist, jedoch eine unkontrolliert steigende Ressourcenauslastung mit steigender Anzahl

an Patches problematisch ist. Im Folgenden werden die einzelnen Metriken sowie die Messstrategien genauer erläutert.

Arbeitsspeicherauslastung

Zur Überprüfung der Arbeitsspeicherauslastung während der Programmausführung wurden verschiedene Werkzeuge genutzt, die unterschiedliche Messungen durchführen können. Um vergleichbare Daten für die Optimierung zu sichern, wird für jeden Entwicklungsstand der Arbeitsspeicherverbrauch von drei Durchführungen aufgezeichnet. Aus den gesammelten Daten, die in einem Intervall von einer Sekunde aufgezeichnet werden, lässt sich der durchschnittliche Verbrauch als auch der maximale Verbrauch ermitteln. Zudem können kritische Tendenzen, wie eine fehlende Speicherfreigabe über eine Visualisierung identifiziert werden. Die verwendeten Werkzeuge werden in Anhang C vorgestellt. Da durch die Aufzeichnung der Werte ein Overhead von bis zu 5% entstehen kann, wurden die Durchläufe zur Arbeitsspeichermessung nicht mit in die Laufzeitmessung eingeschlossen.

CPU-Auslastung

Eine weitere verwendete Metrik ist die CPU-Last während der Programmausführung. Um diese kontrollieren und vergleichen zu können, wurden ähnlich zur Arbeitsspeicherauslastung Werkzeuge zur Aufzeichnung von Messdaten genutzt, welche in Anhang C vorgestellt werden. Die für drei Durchführungen gemessenen Werte werden in einem Intervall von einer Sekunde aufgezeichnet und können zur Analyse visualisiert werden. Zudem kann mit den Daten die durchschnittliche und maximale CPU-Last ermittelt werden. Der Wert der CPU Auslastung ist stark von dem System abhängig auf dem der Prozess ausgeführt wird und muss deshalb sorgfältig interpretiert werden. Eine sehr niedrige Auslastung kann beispielsweise auf hohe Wartezeiten für Ein- und Ausgabeoperationen hindeuten, die durch Parallelisierung genutzt werden können. Sehr hohe Werte über längere Zeit können wiederum auf eine ungeeignete Implementierungsstrategie, fehlerhafte Ressourcennutzung oder ein leistungsschwaches Host-System hindeuten.

Festplattenspeicherauslastung

Die letzte Metrik ist die akkumulierte Festplattenspeichernutzung über die Ausführungsdauer des Programms. Um Messungen durchführen zu könne, wurde eine Applikation entwickelt, die in regelmäßigen Abständen die Datengröße von einem definierten Speicherort abfragt. Von den Messungen wird die initiale Datengröße abgezogen, um die Differenz zu erhalten.

Kategorie	Metrik	Einheit
Laufzeit	Durchschnittliche Laufzeit (L-AVG)	Millisekunden
Arbeitsspeicher	Durchschnittlicher Verbrauch (A-AVG)	MB
	Maximaler Verbrauch (A-MAX)	MB
CPU	Durchschnittlicher Verbrauch (C-AVG)	%
	Maximaler Verbrauch (C-MAX)	%
Festplattenspeicher	Akkumulierter Verbrauch (F-Akku)	MB

Tabelle 6.2: Übersicht der ermittelten Metriken

6.3 Optimierungsergebnisse

Im folgenden Abschnitt werden die Ergebnisse der Performancemessungen für die Crawl-Durchläufe mit den Adaptern angegeben und verglichen. Zudem werden die Optimierungsoptionen für die verschiedenen Bereiche vorgestellt und angewendet. Nicht adapterspezifische Änderungen werden bei beiden Adaptern vermessen. Für die Performanceanalyse und zur Ermittlung der Durchschnittlichen Laufzeit, wird hauptsächlich auf das in Abschnitt 6.1.3 angegebene Szenario 1 eingegangen. Zur Veranschaulichung von einzelnen Ressourcenproblematiken wird Szenario 2 angewendet.

6.3.1 TFS Adapter

Ausgangssituation

Als Ausgangsversion für die Vermessung der Performance des TFS Adapters wird der erste komplett lauffähige Programmzustand verwendet, der alle entwickelten Tests erfolgreich durchläuft. Die initialen Werte für die ermittelten Metriken für Szenario 1 werden in Tabelle 6.3 präsentiert. Da für die Durchführung keine Daten dauerhaft auf der Festplatte persistiert werden, wird eine Festplattenspeicher-Auslastung von 0 MB festgesetzt.

Optimierungsoptionen

Neben den initialen Metriken werden zur Identifizierung von Optimierungsoptionen für den TFS Adapter die Funktionalitäten der in Anhang C vorgestellten Performance-Werkzeuge genutzt. Über die Metriken der Tabelle 6.3 kann zunächst ein niedriger durchschnittlicher CPU- und Arbeitsspeicherverbrauch für

Kategorie	Metrik	Wert
Laufzeit	Durchschnittliche Laufzeit (L-AVG)	139742 ms
Arbeitsspeicher	Durchschnittlicher Verbrauch (A-AVG)	37,1 MB
	Maximaler Verbrauch (A-MAX)	72,63 MB
CPU	Durchschnittlicher Verbrauch (C-AVG)	10,75%
	Maximaler Verbrauch (C-MAX)	56,2 %
Festplattenspeicher	Akkumulierter Verbrauch (F-Akku)	0 MB

Tabelle 6.3: Initiale Metriken des TFS Adapters für Szenario 1

das Szenario 1 festgestellt werden. Bei dem Durchlauf für Szenario 2 wird jedoch ersichtlich, dass für eine große Anzahl an Patch-Objekten der Arbeitsspeicherverbrauch stark ansteigt. Um dies zu reduzieren ist eine Optimierung des Ladealgorithmus nötig. Der CPU-Verbrauch für Szenario 2 hingegen zeigt keine Optimierungsnotwendigkeit.
Über die Profiling Funktionalität des Performance-Programms Java Visual VM können zwei Faktoren identifiziert werden, für die eine hohe prozentuale Ausführungsdauer benötigt wird. Der zeitintensivste Abschnitt ist das Laden der SCM Daten von der TFS REST API mit 62% der Gesamtlaufzeit. Für das Transformieren der Daten wird lediglich 1% der Laufzeit benötigt. Somit sind die Ziele der Adapter Optimierung eine Einschränkung der Arbeitsspeicherauslastung sowie eine Reduzierung der Ladedauer für SCM Daten. Das Persistieren der Patch-Objekte und der dazugehörigen *FileChanges* benötigt 27,8% der Zeit und bietet damit Optimierungspotential für die Crawl-Applikation.
Eine weitere Möglichkeit zur Reduktion der Ausführungsdauer ist die Parallelisierung der Crawl-Job Ausführung mit mehreren Threads, da somit die Wartezeiten für Ein- und Ausgabe dynamisch genutzt werden können. Da diese Änderung den kompletten Crawl-Prozess betrifft, müssen zunächst beide Adapter überprüft werden, ob keine Performance-Problematiken dadurch entstehen können. Zusammenfassend können vier primäre Optimierungsziele festgehalten werden.

- Optimierung des Arbeitsspeicherverbrauchs für große Patch-Mengen
- Reduzierung der benötigten Ladedauer für SCM Daten
- Optimierung des Speichervorgangs von Patch- und *FileChange*-Objekten
- Parallelisierung der *Crawl-Job* Ausführung

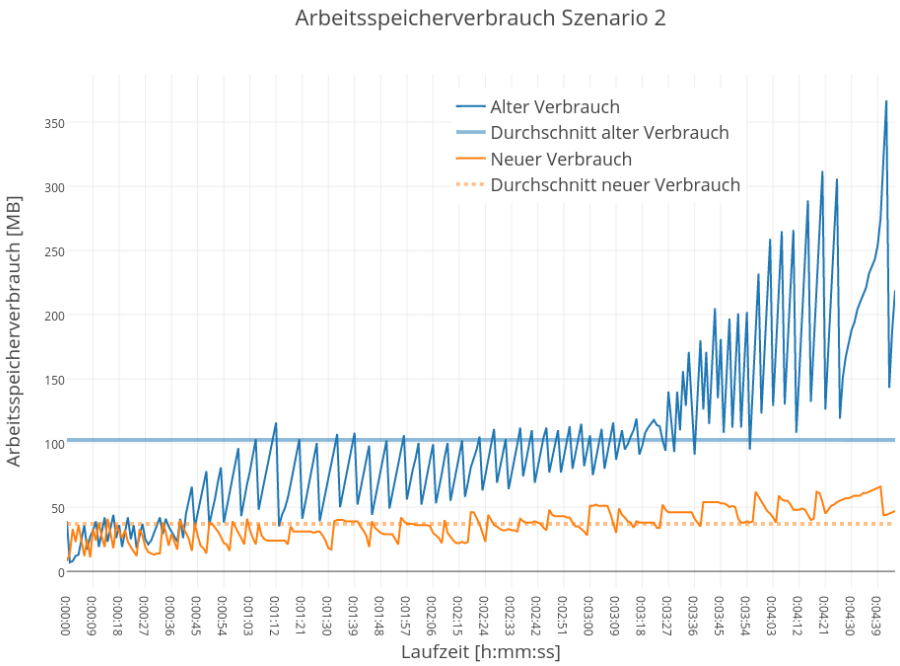

Abbildung 6.1: Vergleich des Arbeitsspeicherverbrauchs über die Laufzeit für vor und nach der Ressourcen-Optimierung des TFS Adapters

Optimierung des TFS Adapter

Das erste Ziel der Optimierung des TFS Adapters ist die Reduktion des Arbeitsspeicherverbrauchs für eine große Anzahl an Patch-Objekten. Über das Performance-Programm Java Mission Control konnte die Abspeicherung aller Patch-Objekte eines Crawl-Jobs in einer Liste als Ursache für den konstanten Anstieg der Arbeitsspeicherauslastung identifiziert werden. Die nötige Umstrukturierung des TFS Adapters sowie die überarbeitete Umsetzung werden in Abschnitt 5.1 genauer erläutert. Um Auswirkungen auf die Arbeitsspeicherauslastung vergleichen zu können, werden in Abbildung 6.1 die aufgezeichneten Arbeitsspeicherverläufe für Szenario 2 visualisiert. Hierbei kann eindeutig das geänderte Verhalten bei der Ressourcennutzung identifiziert werden. Während der alte Verbrauch über die Laufzeit stark ansteigt, ist nur eine leichte Erhöhung bei der neuen Version ersichtlich. Der durchschnittliche Verbrauch konnte von 102,9 MB um 64% auf 36,9 MB reduziert werden. Aufgrund des kontrollierten Ressourcenverbrauchs ist der Adapter in der neuen Version für eine parallele Ausführung einsetzbar. Ein weiterer Effekt der Änderung ist eine leichte Minderung der CPU-Auslastung. Dies ist durch die dynamische Lastverteilung zu erklären. Da die Ressourcenauslastung für das ausführende System reduziert wird und keine Änderung des Laufzeitverhaltens auftritt, kann die Optimierung angewendet werden.

Kategorie	Metrik	Startwert	Aktueller Wert	Änderung
Laufzeit	L-AVG	139742 ms	118333 ms	-15,3%
Arbeitsspeicher	A-AVG	37,1 MB	26,56 MB	-28,4%
	A-MAX	72,63 MB	41,94 MB	-42,3%
CPU	C-AVG	10,75%	11,79%	+9,7%
	C-MAX	56,2%	47,6%	-15.3%
Festplattenspeicher	F-AKKU	0 MB	0 MB	-0%

Tabelle 6.4: Metriken nach Optimierung des TFS Adapters

Das zweite Optimierungsziel für den TFS Adapter ist die Reduktion der Ladezeit für die SCM Daten. Hierfür wurde der Ladealgorithmus überarbeitet und unnötige Abfragen an die TFS REST API entfernt. In Tabelle 6.4 werden die Messwerte im Vergleich zu den Ausgangswerten aufgelistet. Über jeweils 10 Durchläufe konnte die durchschnittliche Laufzeit von 139742 ms um 15,3% auf 118333 ms reduziert werden. Besonders zu betonen ist, dass in der Testumgebung optimierte Bedingungen für einen effizienten Datentransfer bestehen. In einer Umgebung mit längeren Antwortzeiten für Ein- und Ausgabeoperationen würde der Performancegewinn noch deutlicher ausfallen. Durch die erhöhte Ausführungsrate, wird ein Anstieg der durchschnittlichen CPU-Auslastung von 9,92% auf 11,79% vermessen. Da die Änderung nur sehr geringe Auswirkungen auf die Arbeitsspeicherauslastung hat, wird diese nicht weiter ausgeführt. In Abbildung 6.3 werden die prozentualen Änderungen der Durchschnittswerte zum Ausgangszustand abgebildet. Der Zustand nach Optimierung des Adapters wird als Änderung 3 abgebildet. Die Reduzierung der Ausführungszeit um 15,3% für eine Erhöhung der CPU-Auslastung um 9,7% zum Ausgangszustand ist aus Sicht der Performance-Optimierung als positiv zu bewerten und kann folglich angewendet werden.

Optimierung des Crawl-Prozesses

Das erste nicht adaterspezifische Optimierungsziel ist die Reduzierung der Speicherdauer. Die durchgeführten Änderungen am Speicherprozess werden in Abschnitt 5.4 im Detail erläutert. Der Vergleich der optimierten Laufzeit zu der vorangegangen Version wird in Abbildung 6.3 dargestellt. Die durchschnittliche Ausführungszeit für das Szenario 1 konnte durch die Anpassung von 137937 ms um 14,22% auf 118333 ms reduziert werden. Ähnlich wie bei der Optimierung des TFS Ladealgorithmus ist diese Optimierung von der eingesetzten Umgebung abhängig. Da in der Testumgebung eine lokale Datenbank eingesetzt wird, ist es zu erwarten, dass die Optimierung für Umgebungen mit externer Datenbank und höheren Antwortzeiten einen noch deutlicheren Performancegewinn liefert. Da

sich für die Ressourcenauslastung keine deutlichen Änderungen ergeben und eine Laufzeitreduzierung vermessen werden kann, ist die Änderung als rein positiv zu bewerten.

Die letzte Optimierung die am TFS Adapter vermessen wird, ist die Parallelisierung der *Craw-Job*-Aufrufe. Da nach Behebung der Arbeitsspeicherproblematik der TFS Adapter eine kontrollierte Ressourcennutzung bietet, kann er problemlos parallel eingesetzt werden. Die Implementierungsdetails der Änderungen werden in Abschnitt 5.3 genauer vorgestellt. Die Auswirkung der Parallelisierung auf die zu vermessenden Performance-Aspekte ist zunächst eine deutliche Reduzierung der Ausführungsdauer. Die Werte für CPU- und Arbeitsspeicherverbrauch steigen durch die komprimierte Ausführungsspanne deutlich an. Die Änderung der durchschnittlichen Werte wird in Abbildung 6.3 für Änderung 5 im Vergleich zur Ausgangsversion dargestellt. Für die durchschnittliche Ausführungszeit des Endzustands kann eine Reduzierung von 76,9% vermessen werden, die jedoch einen Ansteig der Ressourcenauslastung um bis zu 170% mit sich zieht. Um die Änderungen der Ressourcenauslastung weiter zu verdeutlichen, werden in Abbildung 6.2 die Messwerte eines Durchlaufs von Ausgangs- und Endzustand gegenübergestellt. Da in Szenario 2 nur drei Repositories vermessen werden und damit keine volle Parallelisierung über vier Threads genutzt werden kann, werden die 10 Repositories von Szenario 1 verwendet. Hierbei kann ein deutlicher Anstieg der Werte über eine kurze Zeitspanne festgestellt werden. Die abschließenden Metriken für den SCM Adapter werden in Tabelle 6.5 aufgelistet. Die Auswirkungen auf das System müssen bei dieser Änderung abgewogen werden, da eine deutliche Steigerung der Ressourcennutzung auftritt. Aus der Optimierungs-Perspektive ist die Reduzierung der Ausführungsdauer der Ressourcenreduzierung vorzuziehen. Zudem sind die absoluten Performance-Werte trotz starker Erhöhung in keinem Bereich, der für das auszuführende System kritisch wird. Deshalb kann die Änderung problemlos angewendet werden.

Kategorie	Metrik	Startwert	Endwerte	Änderung
Laufzeit	L-AVG	139742 ms	32285 ms	-76,9%
Arbeitsspeicher	A-AVG	37,1 MB	99,96 MB	+169,5%
	A-MAX	72,63 MB	369,08 MB	+408,2%
CPU	C-AVG	10,75%	28,79%	+167,8%
	C-MAX	56,2%	90%	+60,1%
Festplattenspeicher	F-AKKU	0 MB	0 MB	-0%

Tabelle 6.5: Ermittelte Metriken nach der Optimierung des Crawl-Prozesses und des TFS Adapters

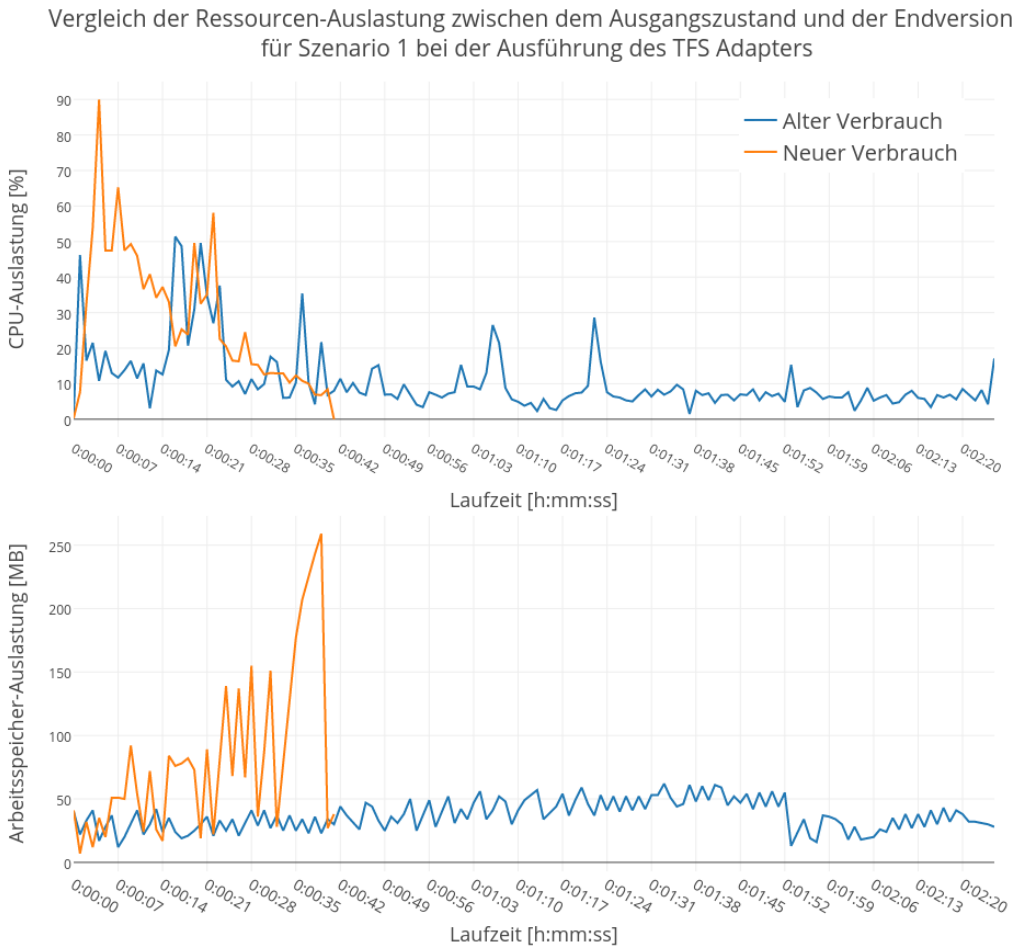

Abbildung 6.2: Vergleich des Ressourcen-Verbrauchs bei einer Ausführung des Crawl-Runs für Szenario 1 für vor und nach Optimierung des TFS Adapters und des Crawl-Prozesses

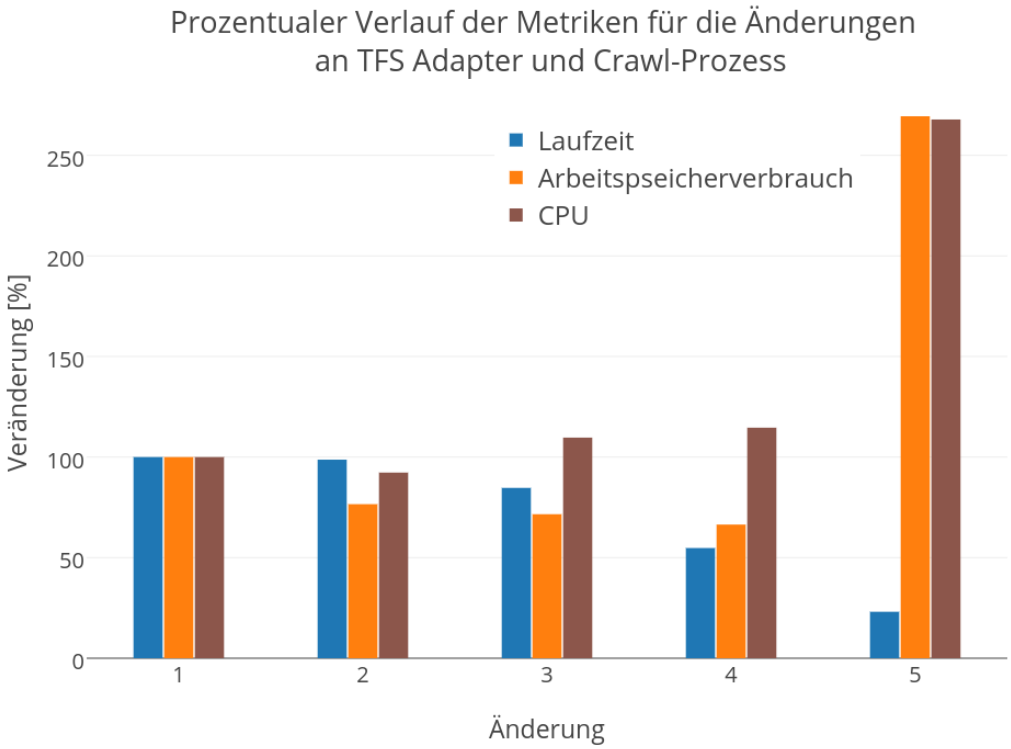

Abbildung 6.3: Vergleich der Durchschnittsmetriken (L-AVG, A-AVG, C-AVG) für die einzelnen Programmzustände zum Ausgangszustand. Änderung 1 ist der Ausgangszustand, Änderung 2 die Arbeitsspeicheroptimierung, Änderung 3 die Optimierung des Ladealgorithmus, Änderung 4 die Optimierung des Speichervorgangs und Änderung 5 die Parallelisierung der Crawl-Jobs.

6.3.2 Git Adapter

Ausgangssituation

Als Ausgangszustand für die Vermessung des Git Adapter wird die ursprüngliche Version aus dem Projekt *CMSuite* verwendet. Die bereits in Abschnitt 6.3.1 für den TFS Adapter vermessenen Änderungen am Crawl-Prozess, werden ebenfalls bei den Git Adapter angewendet und vermessen. Die initialen Messwerte für die in Szenario 1 ermittelten Metriken werden in Tabelle 6.6 präsentiert. Im Vergleich zu den Ausgangswerten des TFS Adapters kann ein deutlich erhöhter Arbeitsspeicherverbrauch festgestellt werden. Dies ist auf die Zwischenspeicherung der Repository-Dateien für die Datenextraktion zurückzuführend.

Kategorie	Metrik	Wert
Laufzeit	Durchschnittliche Laufzeit (L-AVG)	206066 ms
Arbeitsspeicher	Durchschnittlicher Verbrauch (A-AVG)	70,99 MB
	Maximaler Verbrauch (A-MAX)	224,61 MB
CPU	Durchschnittlicher Verbrauch (C-AVG)	17,95%
	Maximaler Verbrauch (C-MAX)	62,4%
Festplattenspeicher	Akkumulierter Verbrauch (F-AKKU)	207 MB

Tabelle 6.6: Initiale Metriken des Git Adapters für Szenario 1

Optimierungsoptionen

Vor der Optimierung wird der bestehende Code mithilfe von Performance-Kontrollwerkzeugen auf Optimierungsmöglichkeiten untersucht. Über die Profiling-Funktionalität von Java Visual VM wird ersichtlich, dass bis zu 40% der Laufzeit für den Download der Projekte verwendet wird. Hierfür kann keine direkte Optimierung vorgenommen werden, da die Ausführungsdauer von der Bandbreite der Netzwerkverbindung abhängig ist. Jedoch kann durch Parallelisierung der auszuführenden Crawl-Jobs die Wartezeit von anderen Threads genutzt werden. Weitere 27% der Laufzeit werden für das Transformieren von geladenen *Commit*-Objekten zu Patch-Objekten benötigt. Für diese Funktionalität kann eine Optimierungsoption für den Git Adapter festgestellt werden. Ebenfalls 27% der Ausführungszeit werden für das Abspeichern der Patch-Objekte benötigt. Da dies im Aufgabenbereich des Crawl-Prozesses liegt, werden die bereits vorgestellten Änderungen erneut mit dem Git Adapter vermessen. Zusammenfassend können drei Optimierungsziele festgehalten werden.

- Optimierung der Patch-Transformation

- Optimierung des Speichervorgangs von Patch- und *FileChange*-Objekten
- Parallelisierung der Crawl-Job Ausführung

Optimierung des Git Adapter

Das Ziel für die Optimierung des Git Adapters ist die Reduzierung der benötigten Zeit für die Patch-Transformation. Für die Durchführung der Aufgabe wird in der Implementierung des Git Adapters das Framework JGit eingesetzt. Da keine performanteren Alternativen zu JGit ermittelt werden konnten, sind die Optimierungsmöglichkeiten auf die Parallelisierung der Transformation beschränkt. Diese wird in Abschnitt 5.1 im Rahmen der Überarbeitung des Git *PatchIterators* beschrieben. Die Auswirkung auf die Performance des Programms ist zunächst die Reduzierung der durchschnittlichen Laufzeit von 206066 ms um 23,3% auf 157976 ms. Die Ressourcenauslastung steigt durch die parallele Durchführung wie in Abbildung 6.4 dargestellt stark an. Zudem kann festgestellt werden, dass trotz großer Daten- und Dateimengen die Ressourcenauslastung beschränkt ist, was für weitere Parallelisierungen von Bedeutung ist. Die erneute Überprüfung der Transformationsdauer mithilfe der Performance-Kontrollwerkzeuge zeigt, dass der prozentuale Zeitanteil zur Gesamtlaufzeit von 27% auf 0,5% reduziert werden konnte. Ein Nebeneffekt hierbei ist die Erhöhung der Ausführungsdauer der Patch-Objekt-Persistierung um 9%. Diese Veränderung ist auf die erhöhte Prozessorauslastung zurückzuführen. Die Werte der einzelnen Metriken sowie der prozentuale Unterschied zu den Werten des Ausgangszustands werden in Tabelle 6.7 präsentiert. Die prozentualen Unterschiede der durchschnittlichen Messwerte zur Ausgangssituation werden zudem in Abbildung 6.5 für Änderung 2 visualisiert. Diese Anpassung ist aus Performance-Sicht positiv, da trotz erhöhter Ressourcenauslastung und hoher Lastspitzen keine kritische Auslastung dauerhaft verursacht wird. Somit kann die Optimierung für den Git Adapter angewendet werden.

Kategorie	Metrik	Startwert	Aktueller Wert	Änderung
Laufzeit	L-AVG	206066 ms	157976 ms	-23,34%
Arbeitsspeicher	A-AVG	70,99 MB	293,57 MB	+313,5%
	A-MAX	224,61 MB	687,14 MB	+205,9%
CPU	C-AVG	17,95%	30,03 %	+70,1%
	C-MAX	62,4%	88,2 %	+41,3%
Festplattenspeicher	F-AKKU	207 MB	207 MB	-0%

Tabelle 6.7: Metriken nach Optimierung des Git Adapters für Szenario 1

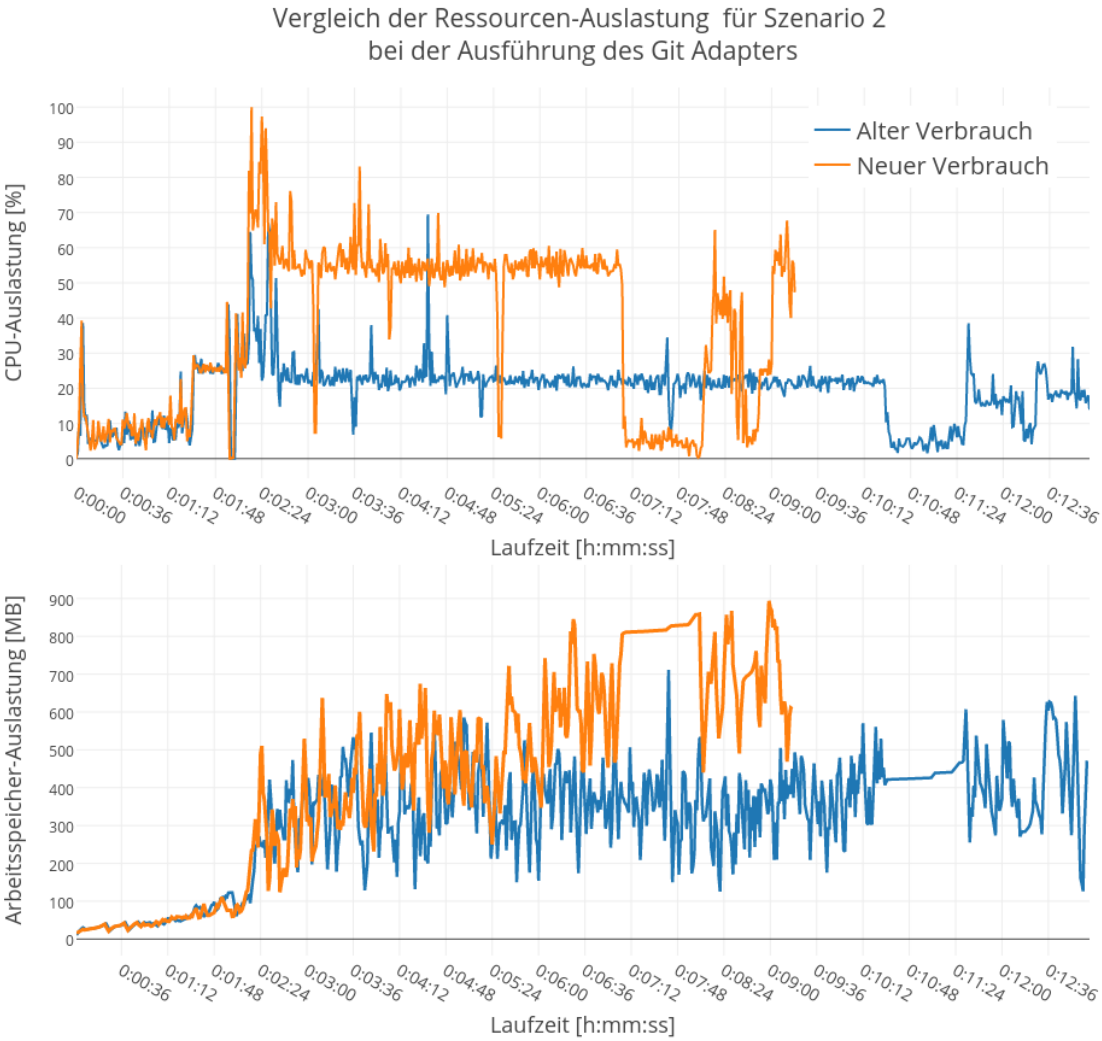

Abbildung 6.4: Vergleich der Ressourcennutzung zwischen der Ausgangsversion und der Version nach Optimierung des Git Adapters für Szeanrio 2.

Optimierung des Crawl-Prozesses

Wie bereits erläutert, werden auch die nicht-adapterspezifischen Änderungen für den Git Adapter vermessen. Die erste Änderung ist die Optimierung des Speichervorgangs, die in Abschnitt 5.4 im Detail beschrieben wird. Diese hat primär relevante Auswirkungen auf die Laufzeit des Programms. So kann die durchschnittliche Ausführungsdauer für einen Crawl-Run mit Git Adapter von 157976 ms um 16,34% auf 132178 ms reduziert werden. Da keine relevanten Änderungen in der Ressourcenauslastung auftreten, ist die Anwendung der Optimierung problemlos durchführbar.

Die letzte Änderung ist die Parallelisierung des Crawl-Prozesses, die in Abschnitt 5.3 erläutert wird. Die abschließenden Metriken und der prozentuale Unterschied zur Ausgangssituation werden in Tabelle 6.7 aufgelistet. Die Auswirkungen auf die Laufzeit des Programms ist eine Reduzierung von 132178 ms um 43,9% auf 74072 ms im Vergleich zur Vorversion. Ein nicht erwartetes Verhalten ist die Reduzierung der durchschnittlichen Arbeitsspeicherauslastung von 293,57 MB auf 252,13 MB, obwohl leicht erhöhte Maximalwerte auftreten. Dies ist auf die optimierte Lastverteilung sowie die erhöhte Aktivitätsfrequenz des *Garbage Collectors* bei multiplen Threads zurückzuführen. Eine weitere Auswirkung ist der deutliche Anstieg der durchschnittlichen CPU-Last von 30,03% auf 56,81%. Während der Programmausführung wurde zudem kurzzeitig eine CPU-Lastspitze von 100% vermessen. Da dieser Wert nicht über eine längere Zeitspanne anhält, ist er nicht als problematisch einzustufen. Über die Darstellung der Ressourcenauslastung in Abbildung 6.6 wird die Komprimierung der Last ersichtlich. Hierbei werden die Werte für durchschnittliche Ausführungen der optimierten Version und der Ausgangsversion für Szenario 1 verglichen. Über einen weiteren Lasttest mit Szenario 2 wurde zudem sichergestellt, dass für große Repositories keine kritische Ressourcenauslastung verursacht wird. Da keine Alternative zu dem kompletten Download der benötigten Repositories gefunden wurde, kann keine Optimierung für den akkumulierten Festplattenverbrauch durchgeführt werden.
In Abbildung 6.5 wird unter anderem die prozentuale Veränderung der abschließenden Durchschnittsmetriken im Vergleich zur Ausgangssituation abgebildet. Hierbei werden die bereits Vorgestellten Änderungen verdeutlicht. Abschließend steht eine Verbesserung der Laufzeit um 64% einer Erhöhung der Ressourcenauslastung um bis zu 255% gegenüber. Da jedoch die in Tabelle 6.8 aufgeführten absoluten Werte für CPU- und Arbeitsspeicherauslastung nicht als kritisch zu bewerten sind und das Programm unter Lasttests keine Probleme aufweist, überwiegt die Optimierung der Ausführungsdauer und die Änderung kann aus Performance-Sicht als positiv betrachtet und angewendet werden.

Kategorie	Metrik	Startwert	Aktueller Wert	Änderung
Laufzeit	L-AVG	206066 ms	74072 ms	-64,1%
Arbeitsspeicher	A-AVG	70,99 MB	252,13 MB	+255,1%
	A-MAX	224,61 MB	701,67 MB	+212,4%
CPU	C-AVG	17,95%	56,81 %	+216,5%
	C-MAX	62,4%	100 %	+60,3%
Festplattenspeicher	F-AKKU	207 MB	207 MB	-0%

Tabelle 6.8: Abschließende Metriken nach Optimierung des Git Adapters und des Crawl-Prozesses für Szenario 1

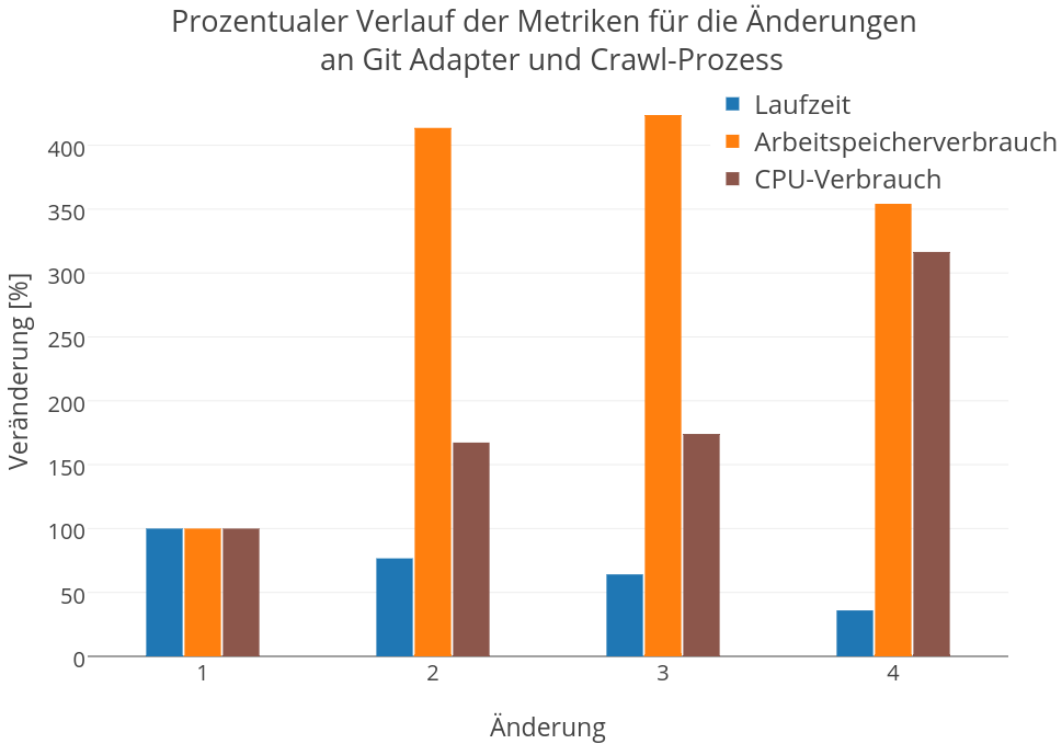

Abbildung 6.5: Vergleich der Durchschnittssmetriken (L-AVG, A-AVG, C-AVG) für die einzelnen Programmzustände. Änderung 1 ist der Ausgangszustand, Änderung 2 die Transformationsoptimierung, Änderung 3 die Optimierung des Speichervorgangs und Änderung 4 die Parallelisierung der Crawl-Jobs.

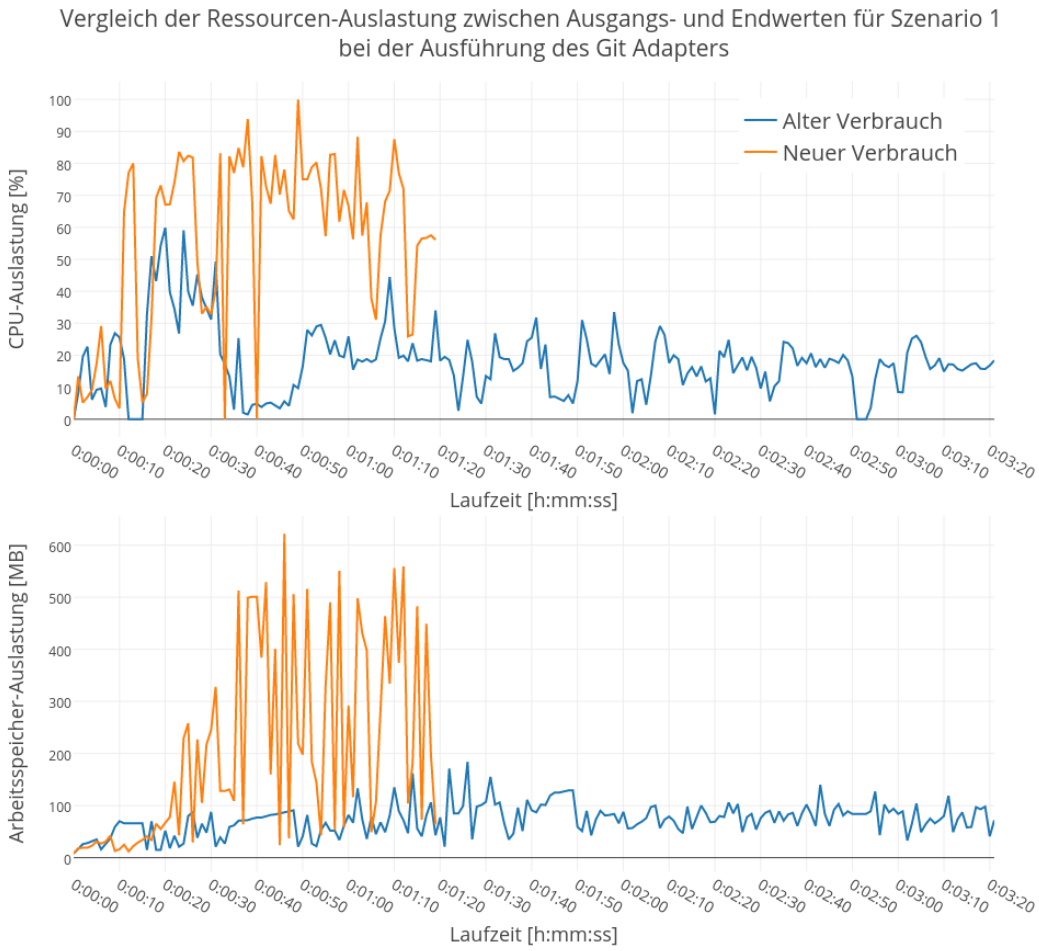

Abbildung 6.6: Vergleich des Ressourcen-Verbrauchs bei einer Ausführung des Crawl-Runs für Szenario 1 nach Optimierung des Git Adapter und des Crawl-Prozesses.

7 Fazit

Zusammenfassend kann festgehalten werden, dass der entwickelte TFS Adapter alle funktionalen Anforderungen erfüllt. Die Anforderungen an die Performance-Optimierung der beiden Adapter sowie des Crawl-Prozesses werden ebenfalls abgedeckt. Für den TFS Adapter kann eine Reduzierung der Ausführungszeit um 76,8 % und für Durchläufe mit dem Git Adapter um 64,1% erzielt werden. In beiden Fällen wird ein Anstieg der Ressourcennutzung vermessen, der jedoch mit Lasttests überprüft wird. Hierbei ist die Ressourcenauslastung trotz Vervielfachung der Daten- sowie Dateimenge begrenzt und somit unproblematisch. Eine Reduzierung der Festplattenspeicherauslastung für den Git Adapter kann nicht erreicht werden, da keine Alternativen zu dem Download der kompletten Datenmenge identifiziert werden konnte. Zudem ist für beide Fälle durch Verbesserung der Hardware eine weitere Performance-Steigerung möglich, was zu einer einfachen Skalierbarkeit führt. Die Optimierung der Adapter kann somit als erfüllt betrachtet werden.

Anhang A Fremdsoftware

Im Folgenden werden die verwendeten Fremdsoftware-Komponenten und die dazugehörigen Lizenzierungen aufgeführt. Um lizenzrechtliche Komplikationen zu vermeiden, wurde bei der Auswahl der Software auf die Verwendung spezieller Open Source Lizenzen geachtet. Zu diesen zählen Apache-2.0, MIT oder EDL. Da die verwendete Software von verbreiteten Projekten oder großen Unternehmen stammt und eine sorgfältige Open-Source-Governance somit gewährleistet ist, wurde keine weitere Lizenzüberprüfung aller Subkomponenten durchgeführt.

Software	Version	Lizenz	URL
Retrofit	2.1.0	Apache-2.0	http://square.github.io/retrofit/
GSON	2.7	Apache-2.0	https://github.com/google/gson
Mockito	1.10.19	MIT	http://site.mockito.org/
Wiremock	2.5.1	Apache-2.0	http://wiremock.org/
JGit	3.7.1	EDL	https://eclipse.org/jgit/

Tabelle 7.1: Übersicht der verwendeten Fremdsoftware-Komponenten

Anhang B GIT Repositories

Um die zu Testzwecken verwendeten GitHub Repositories statisch zu halten, wurde mit einem Funktionsaccount des CMSuite Projekts ein Projekt-Fork für jedes Repository erstellt. Auf diese Weise kann sichergestellt werden, dass über die Dauer der Performance-Tests keine Änderungen an der Projektgröße auftreten. Die erstellten Fork-Projekte werden in Tabelle 7.2 aufgelistet. Die ersten 10 Repositories wurden für Szenario 1 und die weiteren 3 wurden für die Lasttests von Szenario 2 verwendet.

Software	URL
JUnit	https://github.com/cmsuite-integration/it-perf-junit4
Kanban	https://github.com/cmsuite-integration/it-perf-kanboard
GoldenOrb	https://github.com/cmsuite-integration/it-perf-goldenorb
Storm	https://github.com/cmsuite-integration/it-perf-storm
Lets-Chat	https://github.com/cmsuite-integration/it-perf-lets-chat
Java Design Patterns	https://github.com/cmsuite-integration/it-perf-java-design-patterns
Retrofit	https://github.com/cmsuite-integration/it-perf-retrofit
OkHTTP Client	https://github.com/cmsuite-integration/it-perf-okhttp
Asynchronous HTTP	https://github.com/cmsuite-integration/it-perf-android-async-http
Dubbo	https://github.com/cmsuite-integration/it-perf-dubbo
Elastic Search	https://github.com/cmsuite-integration/it-perf-elasticsearch
ZXing	https://github.com/cmsuite-integration/it-perf-zxing
RxJava	https://github.com/cmsuite-integration/it-perf-RxJava

Tabelle 7.2: Übersicht der verwendeten Git Repositories

Anhang C Performance-Kontrollwerkzeuge

Im nachfolgenden Abschnitt werden die Werkzeuge zur Performance-Analyse vorgestellt. Diese wurden zur Identifikation von Optimierungsoptionen sowie zur Aufzeichnung von Performance-Metriken verwendet. Es wurden im Rahmen dieser Arbeit nicht kommerzielle Programme untersucht, welche die benötigten Funktionalitäten anbieten. Zum Vergleich der Daten wurden Probe-Versionen von den kommerziellen Performance-Kontrollprogrammen JProfiler[1] sowie YourKit[2] verwendet. Da diese identisch zu den Ergebnissen der Open Source Werkzeuge waren, wurden für die Performance-Analyse die nicht kommerziellen Programme verwendet.

C.1 Java Visual VM

Das Programm Java Visual VM ist Bestandteil des *Java Development Kits* und kann zum Profiling und Monitoring von *Java Virtual Machines* (JVM) eingesetzt werden. Die grafische Oberfläche des Programms dient zur vereinfachten Interpretation der aufgezeichneten Performance-Daten. Über Plugins kann es zudem

[1] https://www.ej-technologies.com/products/jprofiler/overview.html
[2] https://www.yourkit.com/

um verschiedene Funktionalitäten erweitert werden.

Übersicht

Die Grundfunktion des Programms zeichnet Performance-Werte einer laufenden JVM auf und visualisiert diese in Liniendiagrammen. Standardmäßig werden CPU und RAM-Verbrauch, die Anzahl der geladenen Klassen und aktiven Threads angezeigt. Über das Plugin *Tracer* können diese Werte auch als CSV-Datei exportiert werden. Im Folgenden wird das *Tracer* Plugin sowie die Profiling Funktionalität genauer vorgestellt.

Profiler

Eine wichtige Methode zur Performance-Analyse von entwickelter Software ist das Profiling. Hierbei wird die Software auf verschiedene Aspekte detailliert untersucht und die Ursachen von identifizierten Problematiken recherchiert. Zu häufig untersuchten Faktoren zählt die prozentuale Aufteilung der Ausführungszeit auf einzelne Methoden, die Anzahl und Dauer von Datenbankverbindungen sowie die Verteilung von Objekt-Allokationen auf Methoden (Liang & Viswanathan, 1999).

Der Profiler des Programms Java Visual VM wurde zur Aufzeichnung der prozentualen Laufzeit von Methoden genutzt. Hierfür bietet die Anwendung die Optionen Sampling und Instrumentierung zur Aufzeichnung an. Sampling nutzt den von der JVM angelegten *Stack* der Aufrufe, um in regelmäßigen Zeitintervallen abzufragen, welche Methoden ausgeführt werden. Durch die erhaltenen Werte kann eine statistische Wahrscheinlichkeit angegeben werden, wie lang ein Programm mit der Ausführung einer Methode beschäftigt ist, ohne die Codeausführung zu beeinflussen (Liang & Viswanathan, 1999).
Bei der Instrumentierung wird der zu kontrollierende Code vom Profiler abgeändert und eigener Code zur Messung der Aufrufdauer eingefügt. Auf diese Weise können exakte Ausführungszeiten sowie die genau Anzahle an Methodenaufrufen ermittelt werden. Problematisch ist jedoch das invasive Verändern des ursprünglichen Programmcodes sowie die zusätzliche Zeit, die zur Ausführung des eingefügten Codes benötigt wird. Bei falscher Konfiguration kann somit ein unkontrollierbarer Overhead entstehen (Liang & Viswanathan, 1999). Das Programm Java Visual VM bietet zur Reduzierung des Overheads die Möglichkeit speziell ausgewählte Stellen des auszuführenden Codes zu vermessen und den restlichen Code unverändert zu lassen.

Um exakte Werte bei der Vermessung der Laufzeiten der Methoden zu erhalten, wird im Rahmen dieser Arbeit die Instrumentierungs-Methode verwendet und der zu kontrollierende Code stark eingeschränkt. Somit wird der Overhead

auf 2 - 4% eingeschränkt und es können genaue Ausführungszeiten sowie Aufrufanzahlen für die einzelnen Methoden ermittelt werden. In Abbildung 7.1 wird eine exemplarische Ausgabe des Profilers angegeben. Besonders nützlich in der Anwendung ist die Baumstruktur sowie die graphische Repräsentation der prozentualen Ausführungsdauer, über die zeitintensive Methoden effizient ermittelt werden können.

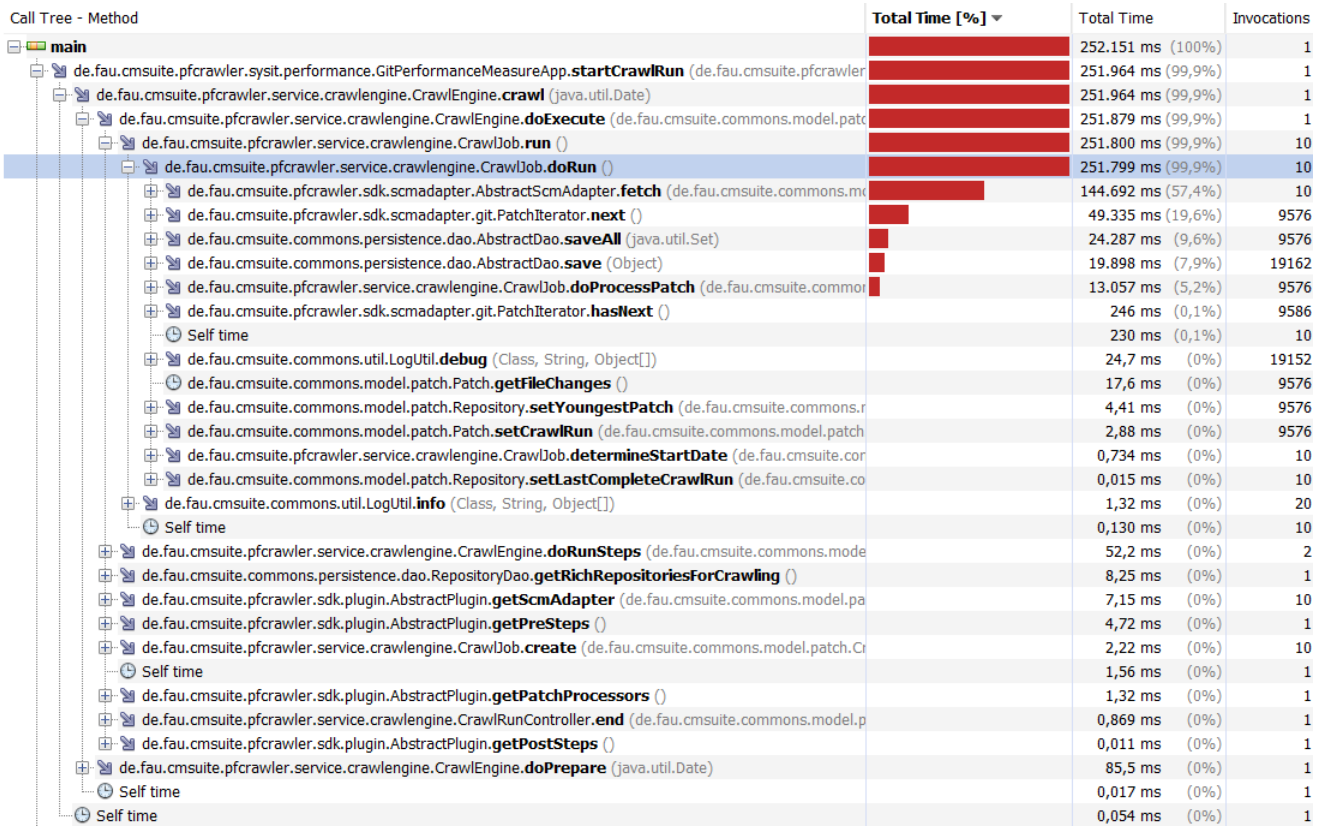

Abbildung 7.1: Beispielausgabe des Profilers

Tracer

Das Plugin *Tracer* von Java Visual VM dient zur Aufzeichnung von verschiedenen Performance-Metriken einer Applikation während der Programmausführung. Neben CPU- und Arbeitsspeicherauslastung können Systemevents sowie JVM-spezifische Werte abgefragt werden. Für diese Arbeit werden die CPU- und RAM-Werte benötigt. Diese werden in einem Intervall von einer Sekunde aufgezeichnet und können nach Programmabschluss zur weiteren Analyse als CSV-Datei exportiert werden.

C.2 Java Mission Control

Das zweite verwendete Werkzeug zur Performance-Kontrolle ist das ebenfalls im Java Development Kit enthaltene Programm Java Mission Control. Dieses bietet eine graphische Oberfläche, über die aktuelle Performance-Werte kontrolliert sowie detaillierte Messungen für verschiedene Metriken initiiert werden können.

Übersicht

Die angegebenen Grundfunktionalitäten sind ähnlich zu dem bereits vorgestellten Programm Java Visual VM. In dem Übersichtsfenster der Applikation werden die aktuellen Werte für CPU- und Arbeitsspeicherauslastung sowie die Thread-Werte für ein ausgeführtes Programm angezeigt. Der volle Funktionsumfang kann genutzt werden, wenn für eine Ausführung eine Werte-Aufzeichnung, *Flight-Recording* genannt, gestartet wird. Vor Beginn der Messung können die Metriken sowie die Abfragefrequenz für das Methoden-Sampling festgelegt werden. Eine hilfreiche Funktionalität bei der Auswertung der gemessenen Daten ist die Selektierung von Zeitfenstern. So kann ein Zeitabschnitt mit hoher CPU-Auslastung ausgewählt werden und der in diesem Bereich ausgeführte Code über das Methoden Sampling nachvollzogen werden. Dadurch lassen sich ressourcenintensive sowie ressourcenschonende Programmstellen einfach identifizieren.

Eine weitere Funktionalität ist die Anzeige des akkumulierten Arbeitsspeicherverbrauchs für einzelne Methoden des Programms. Wie in Abbildung 7.2 präsentiert, wird in einer Baumstruktur die Gesamtgröße der Objekt-Allokationen angegeben. Dies kann zur Optimierung der Arbeitsspeicherauslastung für einzelne Methoden verwendet werden.

Allocation Profile

Stack Trace	Average Object ...	Total TLAB size	Pressure
de.fau.cmsuite.pfcrawler.service.performance.GitPerformanceMeasureApp.main(String[])	11,78 kB	39,47 GB	99,77%
de.fau.cmsuite.pfcrawler.service.performance.AbstractPerformanceMeasureApp.startMeasure(Stri	11,78 kB	39,47 GB	99,77%
de.fau.cmsuite.pfcrawler.service.performance.AbstractPerformanceMeasureApp.measureCraw	11,78 kB	39,47 GB	99,77%
de.fau.cmsuite.pfcrawler.service.performance.AbstractPerformanceMeasureApp.startCrawl	11,78 kB	39,47 GB	99,77%
de.fau.cmsuite.pfcrawler.service.crawlengine.CrawlEngine.crawl(Date)	11,78 kB	39,47 GB	99,77%
de.fau.cmsuite.pfcrawler.service.crawlengine.CrawlEngine.execute(CrawlRun)	11,79 kB	39,47 GB	99,77%
de.fau.cmsuite.pfcrawler.service.crawlengine.CrawlEngine.doExecute(CrawlRun)	11,79 kB	39,47 GB	99,77%
de.fau.cmsuite.pfcrawler.service.crawlengine.CrawlJob.run()	11,79 kB	39,47 GB	99,77%
de.fau.cmsuite.pfcrawler.service.crawlengine.CrawlJob.doRun()	11,79 kB	39,47 GB	99,77%
de.fau.cmsuite.pfcrawler.sdk.scmadapter.git.PatchIterator.next()	12,01 kB	33,44 GB	84,52%
de.fau.cmsuite.pfcrawler.sdk.scmadapter.AbstractScmAdapter.fetch(I	18,06 kB	3,14 GB	7,95%
de.fau.cmsuite.commons.persistence.dao.AbstractDao.saveAll(Set)	69 bytes	1,30 GB	3,28%
de.fau.cmsuite.pfcrawler.service.crawlengine.CrawlJob.doProcessPatc	76 bytes	868,49 MB	2,14%
de.fau.cmsuite.commons.persistence.dao.AbstractDao.save(Object)	66 bytes	580,96 MB	1,43%
de.fau.cmsuite.commons.persistence.dao.AbstractDao.update(Object	64 bytes	164,30 MB	0,41%
de.fau.cmsuite.pfcrawler.sdk.scmadapter.git.PatchIterator.hasNext()	4,07 kB	17,21 MB	0,04%
de.fau.cmsuite.pfcrawler.service.crawlengine.CrawlEngine.doRunSteps(Craw	48 bytes	651,20 kB	0,00%
de.fau.cmsuite.pfcrawler.service.crawlengine.CrawlEngine.prepare(Date)	24 bytes	1,27 MB	0,00%

Abbildung 7.2: Beispielausgabe der Arbeitsspeicher-Allokation für einzelne Methoden

Literaturverzeichnis

Bird, C., Rigby, P. C., Barr, E. T., Hamilton, D. J., German, D. M., & Devanbu, P. (2009 Mai). The promises and perils of mining git. In *2009 6th ieee international working conference on mining software repositories* (S. 1–10). doi:10.1109/MSR.2009.5069475

Capraro, M. & Riehle, D. (2016 Dezember). Inner source definition, benefits, and challenges. *ACM Comput. Surv. 49*(4), 67:1–67:36. doi:10.1145/2856821

Chaturvedi, K. K., Sing, V. B., & Singh, P. (2013 Juni). Tools in mining software repositories. In *2013 13th international conference on computational science and its applications* (S. 89–98). doi:10.1109/ICCSA.2013.22

Czajkowski, G. & von Eicken, T. (1998 Oktober). Jres: a resource accounting interface for java. *SIGPLAN Not. 33*(10), 21–35. doi:10.1145/286942.286944

Dufour, B., Driesen, K., Hendren, L., & Verbrugge, C. (2003 Oktober). Dynamic metrics for java. *SIGPLAN Not. 38*(11), 149–168. doi:10.1145/949343.949320

Georges, A., Buytaert, D., & Eeckhout, L. (2007 Oktober). Statistically rigorous java performance evaluation. *SIGPLAN Not. 42*(10), 57–76. doi:10.1145/1297105.1297033

German, D. M., Adams, B., & Hassan, A. E. (2016). Continuously mining distributed version control systems: an empirical study of how linux uses git. *Empirical Software Engineering*, *21*(1), 260–299. doi:10.1007/s10664-014-9356-2

Liang, S. & Viswanathan, D. (1999). Comprehensive profiling support in the java virtual machine. In *Coots* (S. 229–242).

Martin, R. C. (2003). *Agile software development: principles, patterns, and practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR.

Microsoft Team Foundation Server REST API Documentation. (2017). https://www.visualstudio.com/en-us/docs/integrate/api/overview. Accessed: 2017-05-13.

Ryan, C. & Rossi, P. (2005 September). Software, performance and resource utilisation metrics for context-aware mobile applications. In *11th ieee international software metrics symposium (metrics'05)*. doi:10.1109/METRICS.2005.44

Sokol, F. Z., Aniche, M. F., & Gerosa, M. A. (2013 September). Metricminer: supporting researchers in mining software repositories. In *2013 ieee 13th international working conference on source code analysis and manipulation (scam)* (S. 142–146). doi:10.1109/SCAM.2013.6648195

Wampler, D. (2007). Aspect-oriented design in java/aspectj and ruby. In *Companion to the proceedings of the 29th international conference on software engineering* (S. 184–185). ICSE COMPANION '07. Washington, DC, USA: IEEE Computer Society. doi:10.1109/ICSECOMPANION.2007.22

Wood, T., Cherkasova, L., Ozonat, K., & Shenoy, P. (2008). Profiling and modeling resource usage of virtualized applications. In *Proceedings of the 9th acm/ifip/usenix international conference on middleware* (S. 366–387). Middleware '08. Leuven, Belgium: Springer-Verlag New York, Inc. Zugriff unter http://dl.acm.org/citation.cfm?id=1496950.1496973

Yang, H. Y., Tempero, E., & Melton, H. (2008 März). An empirical study into use of dependency injection in java. In *19th australian conference on software engineering (aswec 2008)* (S. 239–247). doi:10.1109/ASWEC.2008.4483212