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

BENJAMIN MACH
BACHELOR THESIS

EXTENDING A INNER SOURCE PATCH-FLOW CRAWLER

FOR GITLAB AND GITHUB ENTERPRISE

Eingereicht am 31. März 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, 31. März 2017

License

Erlangen, 31. März 2017

Abstract

Due to the advantages, source code management is widely used nearly everywhere in software development,. But often the access to organizational repositories is restricted to individual projects or groups.
In contrast to that Riehle, Capraro, Kips und Horn (2016) describe the application of techniques established in open source development, like organization internal accumulation and publication of knowledge, as an important element of *Inner Source*.

In this context features set of SCM, like following up the author of a repositories commit, is a crucial part for measuring patches between organizational units. The professorship for Open Source Software developed a crawler, with the purpose of gathering and saving Patch-Flow data, by automatic processing of a repositorys metadata.

Extending the Patch-Flow crawler with an interface for GitLab and GitHub Enterprise, allows to use the implemented functionality as standalone or in combination with already existing features. This way, the possible applications and accuracy are enhanced.

Zusammenfassung

Aufgrund der Vorteile beim Einsatz von Source Code Management, sind diese Systeme fast überall in der Software Entwicklung vertreten. Organisationsintern wird der Zugang zu Repositories allerdings oftmals strikt nach Projekt oder Gruppe getrennt.
Demgegenüber beschreiben Riehle et al. (2016) die Anwendung von aus der Open-Source Entwicklung bewährten Techniken, wie die organisationsinterne Veröffentlichung und Sammlung von Wissen, als wichtiges Element von *Inner Source*.

In diesem Kontext ist das Nachverfolgen der Autoren von Beiträgen eines Repository mit Hilfe von SCM, besonders für die Messung von Patches zu anderen Organisationseinheiten interessant.
Um diesen Patch-Flow zu messen, wurde an der Professur für Open-Source-Software ein Crawler entwickelt, welcher bereits unter Verwendung von Git, die Metadaten automatisch verarbeitet und zu speichert.

Mit der Erweiterung des Patch-Flow Crawlers um die Schnittstelle zu GitLab und GitHub Enterprise, eröffnet sich die Möglichkeit die weitere Funktionalität sowohl eigenständig, als auch in Kombination mit bereits bestehender zu nutzen. Somit wird das mögliche Einsatzgebiet erweitert und die Datengenauigkeit erhöht.

Inhaltsverzeichnis

1 Einleitung

Die Verwendung von Source Code Management Systemen bei Software Projekten vereinfacht das Erkennen von Konflikten, sowie das Speichern und Wiederherstellen des Entwicklungsstandes zu einem bestimmenten Zeitpunkt. Das bietet vor allem dann Vorteile, wenn mehrere Software Entwickler gemeinsam an einem Projekt arbeiten. So muss von diesen weniger Arbeitszeit und -aufwand aufgebracht werden, um die Beiträge der einzelnen Beteiligten zusammen zu führen. Deswegen werden heutzutage nur noch selten Projekte ohne ein Source Code Management System umgesetzt.
Das Kernkonzept der meisten SCM Systeme ist ein Repository. Ein Repository enthällt neben den verwalteten Dateien Metadaten, die Informationen über Benutzer, Zeitstempel der Änderungen und die Änderungen selbst, beschreiben. Änderungen der Dateien eines Repository aus der lokalen Arbeitskopie, werden dem Repository als Commit hinzugefügt.

Das für den Linux Kernel entwickelte SCM Git, gehört zu den bekanntesten und ist aufgrund der dedizierten Architektur und sehr bequemen Einrichtung weit verbreitet. Außer der Möglichkeit ein Git-Repository auf einem eigenen Server einzurichten, bieten die vermutlich größten Anbieter GitHub und GitLab den Dienst an, eigene Repositories auf fremden Servern anzulegen. Um das Speichern der Repositories von vielen Benutzern umzusetzen, haben diese Anbieter Funktionen für die Verwaltung von Repositories, Benutzern, Gruppen und Rechten implementiert. Diese von beiden Anbietern unterschiedlich implementierte Verwaltungsfunktionalität, wird genutzt um neben Hosting, die entwickelten Softwaresysteme als Service anzubieten.
Beide Systeme bieten eine REST API an, welche über HTTP Anfragen angesprochen wird und die Möglichkeit bietet, fast alle Interaktionen mit dem Verwaltungssystem mithilfe von HTTP Anfragen abzuwickeln. Massé (2011) beschreibt Representational State Transfer als Designkonzept einer Programmierschnittstelle mit dem zentralen Anspruch, dass Client und Server bei jedem Kommunikationsschritt alle erforderlichen Informationen austauschen, um zustandslos über das HTTP Protokoll zu kommunizieren.

Riehle et al. (2016) definieren *Inner source* als Anwendung von aus der Open Source Entwicklung etablierten Praktiken, innerhalb einer Organisation, mit dem Ziel die Qualität des Quellcodes für die gesamte Organisation zu verbessern. Mit der organisationsinternen Offenlegung von Quellcode, wird das Ziel verfolgt, die vielfältige Implementierung derselben Funktionalität zu minimieren. Somit können Mitarbeiter aus allen Organisationseinheiten auf alle betreffenden Repositories zugreifen und mit dem Beitragen von Patches die Qualität von Quellcode und Testfällen verbessern (Capraro & Riehle, 2016).

Da das Durchsuchen eines Repositories ein automatisierbarer Prozess ist, hat die Forschungsgruppe *Inner Source Software Engineering* der Professur für Open-Source-Software an der Friedrich-Alexander-Universität Erlangen-Nürnberg, einen Patch-Flow Crawler entwickelt. Dieser in Java geschriebene Patch-Flow Crawler sammelt die Metadaten zu jedem Commit, durch das Iterieren von Repositories und speichert die gesammelten Informationen für die weitere Analyse.

Ziel dieser Arbeit ist die Erweiterung des Patch-Flow Crawlers um mit der API von GitHub Enterprise und GitLab zu Interagieren.

Im Folgenden Kapitel 2 gehe ich auf die Anforderungen und Begebenheiten des Artefakts, in Kapitel 3 auf Architektur und Design ein. Die Implementierung wird in Kapitel 4 beschrieben. Abschließend ist die Evaluation des Entwickelten Artefakts in Bezug auf die Anforderungen in Kapitel 5 zu finden.

2 Das Artefakt

Das zu implementierende Artefakt muss sich über die Plugin Schnittstelle in den existierenden Patch-Flow Crawler integrieren, wobei die Entscheidung, ob das Plugin aktiv werden soll, anhand von Konfigurationoptionen getroffen wird.
Ein Durchlauf des Patch-Flow Crawlers besteht aus Sicht eines Plugins aus den Schritten, vor, während und nach der Iteration von Repositories, dem PreStep, dem PatchProcessor und dem PostStep. Diese sind in Abbildung 2.1 dargestellt.

Abbildung 2.1: Der Ablauf aus Sicht des Plugins

Dabei wird je Durchlauf des Patch-Flow Crawlers jeder Pre- und PostStep einmalig gestartet. Jeder PatchProcessor wird dabei für jeden Commit gestartet. Ein Plugin kann für jeden der Schritte, beliebig viele Instanzen von Prozessoren zur Verfügung stellen, welche vom Patch-Flow Crawler an den jeweiligen Punkten des Crawl Durchlaufs aufgerufen werden.

Zur Veranschaulichung in Abbildung 2.2 gibt Plugin 1, zwei PreStep Prozessoren und einen PatchProcessor vor. Plugin 2, gibt einen PreStep, zwei PatchProcessor Instanzen und einen PostStep vor. Zur Vereinfachung wird hier schematisch der Durchlauf mit einem Repository mit nur einem Commit dargestellt, so dass hier die Anzahl der Aufrufe der PatchProcessor Instanzen minimiert ist.

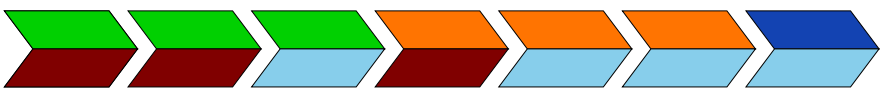

Abbildung 2.2: Beispielablauf mit mehreren Plugins und Prozessoren

Aufgabe des Artefakts ist das Bereitstellen von Funktionalität für die Interaktion mit der API von Github Enterprise und Gitlab. Das Artefakt fordert hierfür die entsprechenden Endpunkte des Anbieters an, konsumiert die Antworten, transformiert diese in dem System bekannte Datenstrukturen, und liefert diese für die weitere Verarbeitung zurück.
Dabei muss das Artefakt unabhängig von anderen Plugins funktionieren. Gleichzeitig liefert es die Funktionalität, alle vom System nutzbaren Informationen von einem Anbieter anzufordern. Letztlich, darf das Adapter keinen Einfluss auf den Ablauf des Patch-Flow Crawlers nehmen, oder den Ablauf anderer Plugins stören.

3 Architektur und Design

Die Architektur des vorgegebenen Patch-Flow Crawlers, gibt für die Schnittstellen des Plugins und der Prozessoren, welche ein Plugin implementieren darf, abstrakte Klassen vor. Dieses Entwurfsmuster stellt sicher, dass die Implementierung die Schnittstelle erfüllt. Genauso muss zum Zeitpunkt der Definition der abstrakten Klasse noch nichts über zukünftig benötigte Funktionalität bekannt sein. Somit kann die gemeinsame Schnittstelle von verschiedenen Plugins, die unterschiedliche Funktionalität bereitstellen, verwendet werden (Gamma, Helm, Johnson & Vlissides, 1995). Mit der Kombination von Interface und abstrakter Klasse kann die abstrakte Klasse als interne Schnittstelle immer zu erfüllende Aufgaben, wie Validierung der Argumente, für alle erbenden Klassen übernehmen. Somit wird eine Schnittstelle in beide Richtungen getrennt.

Das *AbstractScmAdapter* verknüpft die Strukturmuster des Adapters und Iterators.
Gamma et al. (1995) beschreiben das Strukturmuster eines Adapters um eine allgemeine Funktionalität der Schnittstelle anzubieten, welche nicht alle erbenden Klassen implementieren müssen. Die abstrakte Klasse *AbstractScmAdapter* implementiert das Interface *ScmAdapter* und validiert die Parameter der Methode des Interface vor dem Aufruf der eigenen abstrakten Methode.
Ein Iterator bietet die Schnittstelle, eine beliebige Anzahl von Objekten eines Typs zu Durchlaufen. Dies trennt die Auswahl des nächsten zu liefernden Objektes von dem Konsumenten. Somit werden alternative Implementierungen möglich, da der Iterator seine Datenstrukturen und die verwendeten Zugriffsmethoden nicht propagieren muss (Gamma et al., 1995). Ein Beispiel hierfür ist der Iterator des *GitScmAdapter*, welcher mit einer lokalen Arbeitskopie eines Repository arbeitet, im Vergleich zu meiner Implementierung, welche eine Abbildung der Daten anhand einer Liste im Speicher anbietet.

Das Artefakt besteht aus zwei, für einen Anbieter eigenständigen Plugins, welche die Erzeugung des Clients, der Prozessoren und des Transformators für diesen Anbieter übernehmen. Dabei gibt es den nicht in der Schnittstelle spezifizierten Konsens, dass ein Plugin eine statische *create()* Methode anzubieten hat, welche

eventuell benötigte Parameter aus der Konfiguration läd und eine Objektinstanz der Klasse zurück gibt. Die Trennung nach Funktionalität, in Client, Prozessor und Transformator, hat das Ziel der verbesserten Übersicht und Wartbarkeit.

Die spezialisierten Plugins sind in eigene Pakete gegliedert und gemeinsame Funktionalität soweit möglich in einem übergeordneten Paket untergebracht, wie in Abbildung 3.1 dargestellt ist. In dem übergeordneten Paket *generic* sind Klassen mit möglichst hoher Wiederverwendung. In den Paketen *github* und *gitlab* sind die auf den Anbieter spezialisierten Klassen. Die *utils* Pakete enthalten zusätzliche Funktionalität, wie Clients und Transformatoren, unter *model* sind die Objektabbildungen. Hierbei entschied ich mich, das Interface der Servicedefinitionen, aufgrund des Schnittstellencharakters als *util* zu behandeln und von den Objektrepräsentationen zu trennen.

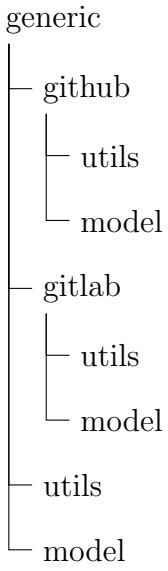

Abbildung 3.1: Paketstruktur des Artefakts

4 Implementierung

In diesem Kapitel gehe ich exemplarisch darauf ein, wie die Verwendung des Retrofit Frameworks für die API Interaktion 4.1 und das Plugin zum konfigurationsbasierten Laden eines Plugins 4.2 implementiert wurde.

4.1 Retrofit Framework

Das unter der Apache Lizenz [1] stehende Retrofit Framework [2] bietet die Funktionalität, die Interaktion mit einer API klassenbasiert und generisch umzusetzen. Die Retrofit Builder Klasse erzeugt mit der Information des URL Pfads und einer ConverterFactory ein Retrofit Objekt. Dieses Retrofit Objekt wird mit einem Interface, welches die API Endpunkte beschreibt, dazu genutzt, eine mit einer Factory verknüpfte Instanz des Interface zu generieren. Der Aufruf einer im Interface definierten Methode dieser Instanz erzeugt ein Call Objekt vom Typ des im Interface definierten Rückgabewertes. Die durch den Aufruf der *execute()* Methode des Call Objekts angefragte Antwort des Servers wird von der mit der Instanz verknüpften Factory mithilfe der ConverterFactory in den im spezifizierten Typ konvertiert. Diese Transformation des Objektes der API zu einem Java Objekt funktioniert jedoch nur, wenn die Factory die Antwort des Servers mit den Java Datentypen kombinieren kann (Square, Inc., 2017). Deswegen enthalten beide anbieterspezifische *model* Pakete die Abbildung aller zur Transformation nötigen Informationen. Um ein Attribut des Objektes von der API abweichend zu benennen, werden Annotations verwendet, welche von der ConverterFactory zum Zeitpunkt der Objekttransformation verwendet werden.

Da das Interface aufgrund von Bedingungen der instanzerzeugenden Factory innerhalb des Retrofit Frameworks keine Unterklasse sein darf, bot es sich an, eine generische Methode zur Erzeugung des Service zu implementieren. Diese Funktionalität ist in der Klasse *GenericRestClient* implementiert, welche *GithubClient*

[1] http://www.apache.org/licenses/LICENSE-2.0
[2] http://square.github.io/retrofit/

und *GitlabClient* erweitern.

```
protected <T> T retrofitSetup(Class<T> _interface) {
  ...
  // Initialisierung des OkHttpClient und der Retrofit Instanz
  return retrofit.create(_interface);
}
```

Listing 4.1: Methodendefinition retrofitSetup

Genauso ist im *GenericRestClient* der Aufruf der *execute()* Methode des generischen Call Objekts, mit Fehlerbehandlung, Validierung und Abarbeitung der Header der Antwort des Servers.

```
private <T> T callApiRequest(Call<T> _call) throws IOException {
  // Aufruf des Endpunktes
  retrofit2.Response<T> response = _call.execute();
  ...
  // Validieren der Antwort und Bearbeiten der Header
  return response.body();
}
```

Listing 4.2: Methodendefinition callApiRequest

Um vereinzelt auftretende Fehler, wie einen Paketverlust, abzufangen, ist die Methode *pullApi* für Klassen, die von *GenericRestClient* ableiten, sichtbar. Diese fängt eine zum Zeitpunkt des Kompilierens definierte Zahl an Ausnahmen ab. Tritt eine Exception auf, wird in der Behandlung eine vordefinierte Zeit lang im Backoff gewartet. Dies geschieht mit dem Ziel, dass der erneute Aufruf nach dem Backoff eine Antwort des Servers zurückliefert.

Die effektiv passendsten Werte hängen von vielen Faktoren, wie der Netzwerkanbindung oder auch der Auslastung des Servers, ab. Dabei müssen diese Werte so gewählt sein, dass ein Paketverlust keine zu große Verzögerung auslößt und gleichzeitig nicht endlos versucht wird eine Antwort des Servers zu erhalten.

Für die Authorisierung erwartet jeder Anbieter eine Zeichenfolge in Form eines Tokens. Dieser Token kann als Url Parameter oder im Header übermittelt werden, wird allerdings aufgrund des Kriteriums der Zustandslosigkeit bei jeder Anfrage erwartet (GitHub, Inc., 2017c; GitLab, Inc., 2017). Um das Hinzufügen des Tokens bei jeder Anfrage zentral zu behandeln, wird eine Implementierung der Klasse *Interceptor* in die Abarbeitungskette eines Objekts der Klasse *OkHttpClient* eingehängt. Die Instanz von *Interceptor* überschreibt die *intercept* Methode und hat somit vollen Zugriff auf die Anfrage. Ebenso wird der *Cache-Control* Header der Antwort modifiziert, um die Vorgabe des Servers zu überschreiben

und das Nutzen des Cache zu erzwingen.

Diese Methode habe ich auch bei der Implementierung des *MockInterceptor* verwendet um selbst konstruierte Antworten an den Client zurück zu geben. Die Implementierung des *MockInterceptor* war zu Beginn der Arbeit notwendig, um eine Umgebung für Unit Tests des Clients zu schaffen, da das im Patch-Flow Crawler verwendete Mockito Framework [3] zu diesem Zeitpunkt die Anfragen nicht erkannt hat, und somit niemals die präparierte Antwort statt der reellen Serverantwort zurückgeliefert wurde.

4.2 Plugin Loader

Die Klasse *GenericPluginLoader* läd zum Zeitpunkt der Instanziierung die Konfigurationsoption, in welcher der voll qualifizierte Klassenname des zu startenden Plugins angegeben ist. Das zentrale Ziel ist, die Entscheidung des genutzten Plugins näher an die Ausführungszeit zu rücken, statt vor dem Kompilieren nötig zu sein. Somit kann der Patch Flow Crawler mit einem Kompiliervorgang bei allen implementierten Einsatzzwecken, je nach Konfiguration genutzt werden.

Die Implementierung sucht mithilfe der Java Reflect API nach dem angegebenen Klassennamen und gibt diese nur zurück, falls die gefundene Klasse von der Klasse *AbstractPlugin* abgeleitet ist.

```
1 Class<? extends AbstractPlugin> classLoader = this.getClass()
2   .getClassLoader()
3   .loadClass(pluginName)
4   .asSubclass(AbstractPlugin.class);
```

Listing 4.3: Verwendung von Reflection

Durch den Aufruf von *asSubClass* ist die Typsicherheit zur Laufzeit gegeben. Solange der Entwickler des Plugins sich an das Schema der statischen *create* Methode gehalten hat, schlägt die Suche nach dieser nicht fehl und sie kann aufgerufen werden, um die zurückzugebende Plugin Instanz zu erhalten.

[3] http://site.mockito.org/

5 Evaluation

Die Evaluation dieser Arbeit gliedert sich in den Abschnitt des Testens 5.1, der Erkennung von Objekten 5.2 und den Ausblick auf Weiterentwicklungen 5.3.

5.1 Testumgebungen

Die Maven Umgebung der Vorgabe enthällt die Möglichkeit für JUnit- und Integrationstests. Der erfolgreiche Ablauf dieser gehört in der vorgegebenen Konfiguration von Maven zu jedem Durchlauf.

Um die Umgebung mit Maven, PostgreSQL und ApacheDirectoryStudio von meinem System abgetrennt testen zu können, habe ich eine virtuelle Maschine mit Xubuntu verwendet. Zu Beginn war je eine virtuelle Maschine mit eigenständiger Umgebung für Maven, GitHub Enterprise und GitLab das Ziel.

Das Testen mit einer eigenen GitHub Enterprise Evaluationsinstanz war jedoch aufgrund der hohen Hardware Mindestanforderungen für eine Virtualbox Instanz nicht möglich (GitHub, Inc., 2017b). Aus diesem Grund wurden alle Testläufe des github Plugins mit der öffentlichen API von github.com durchgeführt. Hier ist anzubringen, dass die öffentliche API von GitHub im Gegensatz zur Enterprise Version, keinen Versionsstring in der Url enthällt (GitHub, Inc., 2017a). Aus demselben Grund habe ich im *InnerSourceProjectPreStep* und *GithubScmAdapter* je eine Konstante eingeführt, welche die maximale Anzahl an Anfragen an den Client übergibt. Die Einführung eines optionalen Parameters von maximalen Anfragen ermöglicht gleichzeitig Integrationstests der Methoden mit einer begrenzten Anzahl von Ergebnissen. Da die öffentliche API allerdings die Anzahl an Anfragen innerhalb einer Zeiteinheit limitiert, musste ich die Funktionalität implementieren, wie der Client damit umzugehen hat.

Zum Testen von GitLab schied eine Ubuntu VM aufgrund von Bug #1574349 [1] und weiteren zwar auch aus, allerdings konnte ich ein funktionierendes Docker Image mit GitLab in der Community Edition, in der Version 8.17.3 finden. Dieses von Niclas Mietz und Sameer Naik zusammengestellte Image [2] , enthällt alle für eine GitLab CE Instanz erforderlichen Bestandteile. Mit dieser Instanz war es mir möglich neben der API von gitlab.com, einen Zugang mit administrativen Rechten zu testen.

Beide Systeme unterscheiden ob ein Token zur Authentifizierung übertragen wurde und welche Berechtigungen der Benutzer für diesen Token hat. Für GitLab konnte ich dies auch verifizieren, für GitHub lediglich die Unterscheidung ob ein Authorisierungstoken geliefert wurde oder nicht.

5.2 Erkennen von Objekten

Das Wiedererkennen von Objekten stellte sich besonders bei der Abbildung von Personen problematisch dar. Git speichert die vom Benutzer in der Konfiguration eingetragenen Werte für *Name* und *Email*. Diese werden auch bei beiden Anbietern von den Daten eines Commits zurück gegeben. Die Antwort von GitHub enthällt, falls *author* oder *committer* dem System als Benutzer bekannt sind, ein zusätzliches Feld, mit *id* und *login*. Da die Benutzerdaten somit nicht verlässlich gefüllt sind, schloß ich das Verwenden von nur einem der Felder *Name*, *Email* oder *Login* für den unique *token* aus. Mithilfe der Levenshtein Distanz und unterschiedlichen Kombinationen näherte ich mich an die jetzige Kombination von *local-part*@domain.tld, *Vorname* und *Nachname* an. Somit können folgende Personen über den generierten Token als eine Person wiedererkannt werden:

Name	**Email**
Andreas Müller	hackz0r@gmail.com
Andreas Müller	hackz0r@haxking.com
Andreas, Müller	hackz0r@andreas-müllers-imac

Tabelle 5.1: Erfolgreiches Matching mit generierten Token

Allerdings wird für folgende Personen nicht der selbe Token generiert:

Name	**Email**
Andreas Müller	hackz0r@gmail.com
Andreas Müller	ab42cdef@cip.cs.fau.de

[1] https://bugs.launchpad.net/ubuntu/+source/gitlab/+bug/1574349
[2] https://hub.docker.com/r/sameersbn/gitlab/

Müller, Andreas	hackz0r@gmail.com

Tabelle 5.2: Erfolgloses Matching mit generierten Token

Der erste und letzte Eintrag von Tabelle 5.2 wird allerdings in meiner Implementierung der Suche nach bekannten Personen über den Vergleich der beiden *Email* Strings gefunden.

5.3 Ausblick

Zum jetzigen Zeitpunkt sind mir keine Alternativen zu GitHub oder GitLab, mit einem vergleichbaren Funktionsumfang bekannt. Sollten sich jedoch in der Zukunft andere Systeme etablieren, könnte die Implementierung weiterer spezialisierter Plugins, oder Generalisierung vorhandener Funktionalität vorstellbar sein.

Shvaiko und Euzenat (2005) zeigen interessante Ansätze, um die momentan fest definierten Klassen einer Objektrepräsentation dynamisch zu erzeugen.
Cohen, Ravikumar und Fienberg (2003) beschreiben ihre Ergebnisse mit verschiedenen Ansätzen des String Matchings. Diese könnten bei einer dynamisch erzeugten Objektrepräsentation angewandt werden.

Literaturverzeichnis

Capraro, M. & Riehle, D. (2016). Inner Source Definition, Benefits, and Challenges. *ACM Computing Surveys*, *9*(4).

Cohen, W. W., Ravikumar, P. & Fienberg, S. E. (2003). A Comparison of String Metrics for Matching Names and Records. In *KDD WORKSHOP ON DATA CLEANING AND OBJECT CONSOLIDATION*.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995). *Design Patterns – Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

GitHub, Inc. (2017a). Enterprise – GitHub Developer Guide. Aufgerufen am: 29.03.2017. Zugriff unter https://developer.github.com/v3/enterprise/

GitHub, Inc. (2017b). FAQ – GitHub Enterprise. Aufgerufen am: 29.03.2017. Zugriff unter https://enterprise.github.com/faq

GitHub, Inc. (2017c). GitHub API v3 – GitHub Developer Guide. Aufgerufen am: 30.03.2017. Zugriff unter https://developer.github.com/v3/#authentication

GitLab, Inc. (2017). doc/api – 8.17-stable – GitLab.org. Aufgerufen am: 30.03.2017. Zugriff unter https://gitlab.com/gitlab-org/gitlab-ce/tree/8-17-stable/doc/api#private-tokens

Massé, M. (2011). *REST API Design Rulebook*. Sebastopol, CA: O'Reilly Media.

Riehle, D., Capraro, M., Kips, D. & Horn, L. (2016). Inner Source in Platform-Based Product Engineering. *IEEE Transactions on Software Engineering*, *42*(12), 1162–1177.

Shvaiko, P. & Euzenat, J. (2005). A Survey of Schema-Based Matching Approaches. In *Journal on Data Semantics IV. Lecture Notes in Computer Science* (Bd. 3730). Berlin, Heidelberg: Springer.

Square, Inc. (2017). Retrofit. Aufgerufen am: 28.03.2017. Zugriff unter http://square.github.io/retrofit/