Erkennung von semantisch zusammenhängenden Quelltextabschnitten anhand von Komponententests

Dokumentenart:	Exposé für eine Bachelorarbeit
Autor:	Martin Wittlinger
Matrikel-Nr.:	2066346
Studiengang:	Informatik Bachelor
Betreuer:	Tobias Hey
Datum:	28. August 2019

1 Motivation

Aus dem heutigen Leben ist Software nicht mehr wegzudenken und findet immer mehr Anwendung in kritischen Bereichen wie Infrastruktur oder Medizin. Deswegen wird die Qualität von Software immer wichtiger. Um dies zu gewährleisten gibt es verschiedene Verfahren wie Testen, formale Verifikation und Quelltextanalysen. Die Gewährleistung, dass alle gewünschten Anforderungen umgesetzt sind, ist schwer möglich. Einerseits wegen der natürlichen Sprache, in der die Anforderungen formuliert sind und andererseits weil Software sich wandelt und weiterentwickelt wird. Die Analyse von semantisch zusammenhängenden Quelltextabschnitten bietet eine Möglichkeit zur Verbesserung der automatisierten Zuordnung dieser. Durch eine automatisierte Zuordnung von Quelltextabschnitten zu Anforderungen wäre eine Verifikation der Implementierung der gewünschten Anforderungen möglich. Aktuell gibt es aber noch kein Verfahren, dass vollautomatisch eine zufriedenstellende Präzision und Ausbeute besitzt.
Weil zusammenhängende Quelltextabschnitte zusammen getestet werden sollten, sollten diese eine Informationsquelle für zusammengehörige Quelltextabschnitte sein. Komponententests [MLBK02] bieten sich deswegen an, weil sich Quelltextänderungen an ihnen widerspiegeln. Jede Quelltextänderung, die das Verhalten des Quelltexts ändert, bedingt eine Änderung der Komponententests. Somit können Komponententests eine aktuelle Dokumentation des Quelltexts sein. Deswegen bieten sich Tests als Informationsquelle für das Finden von zusammenhängenden Quelltextabschnitten an.

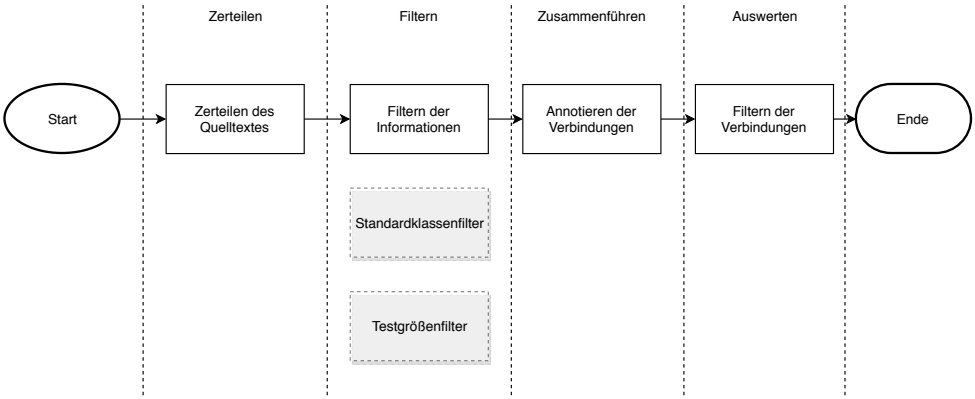

Abbildung 1: Beispielhafter Aufbau des Verfahrens

1.1 INDIRECT

Diese Arbeit findet innerhalb des Projekts `INDIRECT` statt. `INDIRECT` [Hey19] hat das Ziel eine Verbindung von zusammengehörigen Quelltext und Anforderungsbeschreibung in natürlicher Sprache zu erstellen. Um dies zu erreichen, werden zwei Absichtsmodelle (Intent-Model) erstellt und dazwischen Verbindungen erzeugt.
Diese Arbeit befindet sich auf der Seite des Absichtsmodells für den Quelltext und erzeugt die Informationen aus den Tests. Das initiale Absichtsmodelle des Quelltextes, welches eine andere Arbeit in dem Projekt erstellt, soll dadurch die Information bekommen, welche Quelltextteile semantisch zusammengehörig sind.

2 Zielsetzung

Das Ziel der Arbeit ist das Entwerfen und Evaluieren eines Verfahrens zum Finden von semantisch zusammenhängenden Quelltexten. Das Verfahren soll hierfür Informationen aus Testfällen gewinnen und auswerten.
Das Verfahren soll sich hierbei adaptiv an die zur Verfügung stehenden Informationen anpassen können. Diese Informationen können z.B. Kommentare sein.

2.1 Vorgehen

Um die Ziele zu erfüllen ist ein neues Verfahren erforderlich. Bisherige, in der Literatur vorhandene Verfahren, wie z.B. Letzter Aufruf vor Überprüfung (LCBA) [QODL10] , besitzen verschiedene Probleme. LCBA betrachtet nur den Methodenaufruf in der Überprüfung und ordnet diesen den genutzten Klassen zu. Dies ist aber nicht immer erfolgsversprechend wie in dem

```
@Test
public void testCantFindFixture() throws Exception {
  String pageString = "<table><tr><td>NoSuchFixture</td></tr></table>";
  Parse page = new Parse(pageString);
  Fixture fixture = new Fixture();
  fixture.doTables(page);
  String fixtureName = page.at(0, 0, 0).body;
  assertTrue(fixtureName.contains("Could not find fixture: NoSuchFixture."));
}
```

Abbildung 2: Beispiel für einen Komponententest. Dieser testet einen Fehlerfall beim Zerteilen des Strings bei der Seitenerstellung mit einer Tabelle.

Quelltextausschnitt 2 ersichtlich ist.
In der Überprüfung wird dort mit `String::contains` eine Methode der Standardbibliothek genutzt. Dadurch würde nur die Information entstehen, welche Klassen die Methode der Standardbibliothek nutzen und nicht welche Quelltextabschnitte zusammengehörig sind. Um solche Probleme bisheriger Verfahren zu umgehen, soll das Verfahren im Rahmen dieser Arbeit mehr als nur die Überprüfung betrachten und auswerten. In der Abbildung 1 ist ein mögliches Vorgehen dargestellt. Das Vorgehen besteht aus vier aufeinanderfolgenden Phasen: Zerteilen, Filtern, Zusammenführen und Auswerten. Beim Zerteilen des Quelltextes wird aus jeder Testklasse der Quelltext jeder Methode und Metainformationen, wie z.B. Größe der Testklasse, extrahiert. In der nächsten Phase wird jede Methode einzeln gefiltert. In der Filterphase werden verschiedene Filter verwendet, um nur die zielführenden Informationen weiter zu betrachten. Ein solcher Filter könnte z.B. der Standardklassenfilter sein. Dieser würde Methoden aus der Standardbibliothek filtern, weil diese nicht Teil des Quelltexts des Programms sind. Die Aussage, ob ein Quelltextabschnitt mit einer Standardklasse semantisch zusammengehörig ist, ist nicht das Ziel von `INDIRECT`. Ein weiterer möglicher Filter wäre z.B. der Testgrößenfilter. Er würde zu große oder zu kleine Tests aus der Betrachtung entfernen, weil diese nicht verwertbare Informationen liefern könnten. Bei zu großen Tests wäre ein weitere Aufteilung des Tests in Teilszenarien notwendig. Bei zu kleinen Tests, mit nur einer genutzten Methode, wäre keine Aussage möglich, weil nur ein Quelltextabschnitt vorhanden ist. Teil der Arbeit wäre es, diese Filter zu testen und zu evaluieren, welche weiteren Filter das Ergebnis verbessern. In der Phase des Zusammenführens wird der aufgeteilte Datensatz aus der zweiten Phase zusammengeführt und die Informationen gewichtet. Die Auswertung der Information soll im Gesamtkontext der Klassen stattfinden. Falls aus mehreren Tests die Information gewonnen wurde, dass zwei Methoden zusammengehörig erscheinen könnte dies stärker

auf eine Zusammengehörigkeit hindeuten. Die Informationen und Gewichtung, die beim Zusammenführen verwendet werden um gute Ergebnisse zu liefern, sollen im Rahmen dieser Arbeit bestimmt werden. Beispielsweise wird es sein, dass identische Verbindungen aus verschiedenen Testklassen stärker auf eine Zusammengehörigkeit hindeuten. Im vierten Schritt findet die Auswertung statt. In der Auswertung wird, anhand der Informationen aus den vorherigen Phasen, entschieden, welche Aussage über die vermutete Verbindung getroffen werden kann. Jede Vermutung wird nach der Auswertung ein Konfidenzniveau besitzen, dass aussagt wie sehr eine Verbindung für vorhanden erachtet wird. Dieses Vorgehen erlaubt durch den pipelineartigen Aufbau leicht verschiedene Filter einzubauen, zu testen und durch Parameter, wie z.B. ihre Gewichtung anzupassen.

2.2 Beispiel

In dem Quelltextausschnitt, aus `Fitnesse` [1] in Abbildung 2, einer Art Wiki für gute Tests, ist ein Komponententest abgebildet. Dieser Test überprüft das Erkennen eines Fehlers beim Zerteilen während der Seitenerstellung. Das vorgestellte Vorgehen würde nach dem Zerteilen des Quelltextes in einzelne Methoden diese einzeln betrachten. Die hierbei verwendeten Klassen sind `String`, `Parse` und `Fixture` sowie die Methoden `Parse::Parse(String)`, `Fixture::Fixture`, `Fixture::doTables`, `Parse::at`, `Parse::body` und `String::contains`. In der Filterungphase würde der Standardklassenfilter die Klasse `String` und die Methode `String::contains` filtern. Der Schritt des Zusammenführens wird hier der Übersichtlichkeit wegen unterlassen. Übrig wäre eine erste Vermutung, dass die Klassen `Fixture` und `Parse` semantisch zusammenarbeiten und die Methode `Fixture::doTable` semantisch mit der Klasse `Parse` zusammengehörig ist. In der letzten Phase würde die Auswertung der Informationen ergeben, dass die Vermutung nach dem Filtern stimmt.

3 Evaluation

Das Verfahren während der Arbeit soll mit drei verschiedene Mengen an Programmen durchgeführt werden. Einer `Startmenge`, an der sich ein Verfahren überlegt und entwickelt wird. Ein `Zwischentest`, der einem zeigt wie gut das überlegte Verfahren ist. Am Ende soll eine `Endevaluation` mit möglichst verschiedener Software von der Qualität als auch Herkunft und Größe stattfinden.
Dies zeigt wie gut das Verfahren generalisiert und ob es sich auf verschiedene Quelltextqualitäten als auch verschiedene vorhandene Informationen anpassen kann.

[1] https://github.com/unclebob/fitnesse

Als Messgrößen würden sich Ausbeute und Präzision im Vergleich zu einer Musterlösung anbieten, an denen auch die Parameter des Filters optimiert werden können. Präzision dient als Messgröße wie genau das Verfahren ist, Ausbeute als Messgröße wie vollständig das Ergebnis ist. Das F-Maß aus Präzision und Ausbeute, wird als allgemeine Messgröße für die Qualität des Verfahrens verwendet. Ein Problem könnte die Datengrundlage sein. Es wird eine Übungsmenge, die klar klassifiziert hat welche Quelltextabschnitte zusammengehören, benötigt. Hier würde sich ein Projekt mit Musterlösung anbieten. Bei unzureichender Datengrundlage wird für die Evaluation eine Datenmenge erstellt werden. Dafür bieten sich dann Projekte mit vorhandenen Test zu Quelltext Zuordnungen über Bugmeldungen etc. an. Denkbare Software für die Übungsmengen sind ArgoUML [2], OpenJDK [3], Jenkins [4]. Hier ist zu beachten, dass möglichst verschiedene Software betrachtet wird, weil sich die Qualität, Standards und Ansprüche innerhalb von Gemeinschaften stark unterscheiden.

Literatur

[Hey19] Hey, Tobias: INDIRECT: intent-driven requirements-to-code traceability. In: Mussbacher, Gunter (Hrsg.) ; Atlee, Joanne M. (Hrsg.) ; Bultan, Tevfik (Hrsg.): *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019.*, IEEE / ACM, 2019, 190–191

[MLBK02] Malaiya, Y. K. ; Li, M. N. ; Bieman, J. M. ; Karcich, R.: Software reliability growth with test coverage. In: *IEEE Transactions on Reliability* 51 (2002), Dec, Nr. 4, S. 420–426. http://dx.doi.org/10.1109/TR.2002.804489. – DOI 10.1109/TR.2002.804489. – ISSN 0018–9529

[QODL10] Qusef, Abdallah ; Oliveto, Rocco ; De Lucia, Andrea: Recovering Traceability Links Between Unit Tests and Classes Under Test: An Improved Method. In: *Proceedings of the 2010 IEEE International Conference on Software Maintenance*. Washington, DC, USA : IEEE Computer Society, 2010 (ICSM '10). – ISBN 978–1–4244–8630–4, 1–10

[2] http://argouml.tigris.org/

[3] https://github.com/AdoptOpenJDK/openjdk-jdk11u

[4] https://github.com/jenkinsci/jenkins