Universität Leipzig

Wirtschaftswissenschaftliche Fakultät

Institut für Wirtschaftsinformatik

Prof. Dr. Ulrich Eisenecker

Thema

Arten der Redundanz im Zusammenhang mit Code-Clones

Bachelorarbeit zur Erlangung des akademischen Grades

Bachelor of Science – Wirtschaftsinformatik

vorgelegt von: Willert, Nico

Email-Adresse: NicoWillert@gmx.net

Telefonnummer: 01771418190

Anschrift: Neuer Weg 12

04539 Groitzsch

Leipzig den 11.08.2017

Abstract

Durch Redundanz im Quellcode kommt es zur Einschränkung wichtiger Faktoren wie der Lesbarkeit oder Wartbarkeit des Codes. Damit einhergehend kann Fehlverhalten im Programmablauf entstehen, wenn Code-Fragmente gezielt dupliziert werden, anstatt sie wiederzuverwenden. Für die frühzeitige Erkennung solcher Probleme ist es daher nötig, die Redundanz in ihren verschiedenen Formen aufzuschlüsseln.

Das Ziel dieser Arbeit war es zu untersuchen, wodurch sich diese Formen beziehungsweise Arten der Redundanz unterscheiden, wie diese zusammenhängen und auf welche Weise man Redundanz mit dem Begriff Code-Clone zusammenführen kann. Zu diesem Zweck wurde eine Literaturstudie durchgeführt, um den aktuellen Forschungsstand zu erfassen. Dabei wurden neben der Redundanz auch die Themen Code-Clones und Ähnlichkeit betrachtet. Die Ergebnisse der Literaturstudie wurden anhand der Arten der Redundanz gegliedert und durch Code-Clone-Beispiele verdeutlicht.

Die Literaturstudie ergab, dass Redundanz vorwiegend durch Duplikation von Code-Fragmenten entsteht, wodurch sich mithilfe von Code-Clones ein Großteil der Redundanz abbilden lässt. Des Weiteren sind die Arten der Redundanz nicht disjunkt, wodurch sich eine hundertprozentige Untergliederung nicht durchführen lässt.

Schlüsselwörter

Redundanz, Code Clones, Ähnlichkeit

Gliederung

Gliederung……..…….. I
Abbildungsverzeichnis……..……... II
Quellcode-Listing…… II
1. Einleitung…… 1
1.1 Motivation…….…... 1
1.2 Zielstellung……... 1
1.3 Aufbau der Arbeit…… 2
2. Definitionen……2
3.Vorgehen……... 5
3.1Methodisches Vorgehen…….…... 5
3.2 Planung ……... 6
3.3 Selektion…….. 7
3.4 Extraktion……8
3.5 Ausführung…….. 9
4. Ergebnisse……9
4.1 Negative Software Redundanz……9
4.2 Textuelle Redundanzen…… 11
4.3 Funktionelle Redundanz…….. 14
4.4 Boilerplate-Code……..16
4.5 Entstehungsgrund-basierte Redundanzen…… 18
4.5.1 Gezwungene Redundanz…… 18
4.5.2 Zufällige Redundanz……...20
4.6 Abgrenzung der Redundanzarten voneinander…… 21
5. Fazit…… 23
6. Ausblick……... 25
Quellen……III
Ehrenwörtliche Erklärung……VI

Abbildungsverzeichnis

Abbildung 1: Phasenmodell der Literaturstudie aus Okoli …….5
Abbildung 2: Ansätze zur Untersuchung der Redundanz ……...6

Quellcode-Listing

Listing 1 Multiplikation zweier Variablen in C++……..11
Listing 2: Implementierung der Swap-Funktion in C++ …… 13
Listing 3: Implementierung der Swap-Funktion mithilfe Funktionstemplates in C++ …. 13
Listing 4: Implementierung eines einfachen Stapels in Lua ……...16
Listing 5: Implementation einer Key-Value-Tabelle und eines Arrays in Lua ……... 17
Listing 6: Simpler Bankaccount in C++ ……. 19
Listing 7: Implementierung eines simplen Bankaccounts durch das Idiom einer Klasse in Lua ..……...…….. 21

1. Einleitung

1.1 Motivation

Durch den steigenden Konsum digitaler Produkte ist Software ein wichtiger Bestandteil des Alltags geworden. Eine Vielzahl an alltäglichen Prozessen wird durch Software erleichtert und gesteuert. Vor allem in den Bereichen des Militärs oder der Medizin darf kein Fehler auftreten, da hier Entscheidungen über Leben und Tod richten können. Ebenso ist der Einsatz von Computern und Programmen aus der Wirtschaftswelt nicht mehr wegzudenken. Viele dieser Vorgänge laufen automatisiert ab. Vor allem Banken und der Onlinehandel sind mittlerweile sehr stark von der Verfügbarkeit und Leistung der Softwaresysteme abhängig. Ein Ausfall kann sehr hohe Verluste zufolge haben.
Um sicher zu stellen, dass das System funktioniert, muss eine gewisse Qualität der Software vorherrschen. Diese definiert sich häufig über ein gewisses Maß an Struktur, Übersichtlichkeit und Robustheit der Quellcodes. Viele Faktoren werden direkt durch den Programmierstil des Programmierers beeinflusst. Ein Beispiel wäre die Benennung von Variablen und Funktionen oder die Schachtelung einzelner Code-Fragmente. Andere Faktoren unterliegen den Anforderungen, der Programmiersprache oder Designentscheidungen. Durch die Entwicklung neuer Programmierkonzepte und immer komplexeren Anforderungen kommt es zu einem beständigen Alterungsprozess der Software. Bei der Anpassung an diese neuen Umstände werden Code-Fragmente vollständig dupliziert oder ihre Struktur wird als Template verwendet. Dadurch kann die Lesbarkeit des Codes sinken und der Anteil an redundanten Informationen sowie Code-Zeilen steigt an, was in der Regel nicht wünschenswert ist. Aus diesem Grund steigt auch die Bedeutung der Redundanz in der Softwareentwicklung. Es stellt sich die Frage, ob die Entwicklung neuer Programmierkonzepte unter anderem durch ihr Potential zur Vermeidung von Redundanz getrieben wird.
Um dies genau zu analysieren, ist es wichtig zu verstehen, wodurch sich Redundanz auszeichnet.

1.2 Zielstellung

Im Rahmen dieser Bachelorarbeit wird untersucht, in welche Arten sich Redundanz einteilen lässt, wodurch diese entstehen und anhand welcher Merkmale sie erkennbar sind.

Im Kontext der Arbeit beschränkt sich der Begriff Redundanz auf jene Arten, die sich negativ auf den Code auswirken.
Das Ziel ist es, anhand einer Literaturstudie den bestehenden Stand an Definitionen für verschiedene Arten der Redundanz zu erforschen. Des weiteren ist zu erforschen, wie diese Arten zusammenhängen und wie sie mit Code-Clones in Verbindung zu bringen sind. Wenn noch keine Definitionen für eine bestimmte Art Redundanz bestehen, wird eine Definition anhand von Ähnlichkeit und Code-Clones hergeleitet. Diese Zusammenhänge werden mithilfe von Code-Clone Beispielen dargestellt.

1.3 Aufbau der Arbeit

Das erste Kapitel erläutert die Motivation, die dem Thema der Arbeit zugrunde liegt. Es beschreibt auch die allgemeine Zielstellung sowie den Aufbau der Arbeit. Im zweiten Kapitel sind grundlegende Begriffe definiert, die zum klaren Verständnis der Arbeit erforderlich sind. Das methodische Vorgehen, auf dem die Arbeit beruht, wird im dritten Kapitel verdeutlicht. Es wird darin beschrieben, auf welche Weise die Literaturstudie und deren Auswertung durchgeführt wurden. Im vierten Kapitel werden die Ergebnisse der Studie dargelegt. Diese umfassen die herausgearbeiteten Definitionen der Redundanz und ihren Vergleich mit Code-Clones. Das Kapitel *Fazit* fasst die gewonnenen Erkenntnisse der Arbeit zusammen und bewertet diese. Im letzten Kapitel wird die Arbeit mit einem Ausblick abgeschlossen. Dieser beschreibt, welche weiteren Forschungsfragen durch die Ergebnisse abgeleitet werden können.

2. Definitionen

Redundanz: Der Begriff der Redundanz beschreibt im Allgemeinen den überflüssigen Teil einer Nachricht, der den Informationsgehalt nicht steigert. Er kann auch für die Überladung einer Aussage mit gleichen Merkmalen verwendet werden (vgl. [Wortschatz 2017] und [DudenRedundanz 2017]).
In der Informationstheorie gilt die Redundanz als „Maß für den Grad des Überflüssigen in einer Nachricht aus einer Informationsquelle“ [Topsøe 1974, 64].
In den Gebieten der Fehlertoleranz und „selbstheilender Software“ beschreibt die Redundanz die Fähigkeit, bestimmte Funktionalitäten über verschiedene Wege oder Instanzen auszuführen [Carzaniga 2009]. Ebenso kann die Redundanz durch die Existenz

verschiedener Komponenten beschrieben werden, die die selbe Funktionalität erfüllen [Carzaniga 2015]. Somit kann das System Fehler vermeiden oder sich von Fehlern erholen. Die Redundanz kann durch Hardware gegeben sein, das heißt durch mehrere Recheneinheiten, oder durch Software, beispielsweise durch unterschiedliche Module, die mit gleichen Funktionalitäten ausgestattet sind. Es ist zwischen aktiver und passiver Redundanz zu unterscheiden. Aktive Redundanz beschreibt Komponenten, die parallel an einer Aufgabe arbeiten. Komponenten der passiven Redundanz werden nur beim Fehlerfall aktiv [IEEE Standard Vocabulary 2010].
Diese Formen der Redundanz sind beabsichtigt, um einen positiven Effekt für das System zu erzeugen. Aufgrund von verschiedenen Faktoren kann Redundanz der Lesbarkeit und Wartbarkeit der Software schaden (siehe negative Software-Redundanz). Diese negative Redundanz entsteht durch unterschiedliche Gründe, die zum Teil ungewollt aber auch erzwungen oder gewollt sein können. Beispielsweise entsteht diese negative Redundanz durch eine fehlende Abstraktion oder das Kopieren von Programmfragmenten.

Code-Clone : Der Begriff Code-Clone variiert aufgrund verschiedener Definitionen von Ähnlichkeit (Englisch *similarity*). Im Allgemeinen werden darunter Quelltext-Fragmente und Programm-Strukturen verstanden, die identisch oder ähnlich zueinander sind. Die Länge des betrachteten Codes und die untersuchte Ähnlichkeit unterliegen der menschlichen Einschätzung (vgl. [Basit 2006]). In den meisten Fällen nutzen Programmierer *copy and paste,* um den bereits erstellten Code an weiteren Positionen einzufügen und dann an neue Anforderungen anzupassen. Es lassen sich verschiedene Fälle oder Typen von Code-Clones unterscheiden. Ein exakter Code-Clone (Typ Eins) beschreibt Programmfragmente, die bis auf Kommentare und Leerzeichen übereinstimmen. Der parametrisierte Code-Clone (Typ Zwei) ist strukturell beziehungsweise syntaktisch ähnlich, mit Änderungen von Bezeichnern, Literalen, Typen, Layout und Kommentaren. Ein als *near-miss* (Typ Drei) bezeichneter Code-Clone ist eine Kopie mit erweiterten Modifikationen wie eingefügten oder gelöschten Anweisungen [Bellon 2007]. Der semantische Code-Clone (Typ Vier) beschreibt Quelltext-Fragmente, die eine ähnliche Funktionalität aufweisen, ohne auf textueller Ebene gleich zu sein. Jeder Typ kann die vorangegangenen Typen enthalten [Rattan 2013]. Weiterhin wird von strukturellen Code-Clones gesprochen, wenn Entwurfsmuster eingesetzt wurden. Außerdem werden zwei Funktionen, die eine ähnliche Funktionalität abdecken, jedoch unterschiedlich implementiert sind, als funktionell ähnliche Code-Clones bezeichnet (vgl. [Wagner 2016]).

Nach [Kapser 2008] werden Code Clones als schädlich erachtet. Durch ihre Existenz werden die Wartung sowie die Komplexität des gesamten Systems erhöht. Weiterhin besteht das Problem, dass nötige Änderungen konsistent auf alle vorhandenen Code-Clones angewendet werden müssen, was nicht immer gewährleistet ist und dann die Softwarequalität verschlechtert. Duplizieren von Code kann jedoch auch für die Wiederverwendung genutzt werden, um den Quellcode zu abstrahieren. Dies kann zur Übertragung von Fehlern führen. Dies bedeutet, dass Fehler in Programmfragmenten durch das Kopieren und Einfügen vervielfältigt werden (vgl. [Anil Kumar 2012]).

Ähnlichkeit: Der Begriff der Ähnlichkeit beschreibt, wie ähnlich, austauschbar oder gleich mehrere betrachtete Systeme sind. Ebenso kann er angeben, wie Systeme durch bestimmte Eigenschaften und Verhalten voneinander abweichen (vgl. [Walenstein 2007]). Walenstein beschreibt die Ähnlichkeit als den Grad, in dem zwei unabhängige Programme sich darin ähneln, bestimmte Aufgaben auszuführen. Die Ähnlichkeit ist zunächst in zwei Typen zu unterscheiden. Die gegenständliche beziehungsweise repräsentative Ähnlichkeit und die semantische beziehungsweise verhaltensorientierte Ähnlichkeit. Die repräsentative Ähnlichkeit beschreibt die Gemeinsamkeiten auf textueller Ebene. Diese umfassen jede textuelle, syntaktische und strukturelle Form auf verschiedenen Abstraktionsebenen (vgl. [Walenstein 2007] & [Juergens 2010]). Semantische Ähnlichkeit unterteilt sich noch einmal in funktionelle und Ausführungsähnlichkeit. Zwei Programme sind funktionell ähnlich, wenn sie ähnliche Funktionen implementieren, und sie sind in der Ausführung ähnlich, wenn sie ein gemeinsames Verhalten für Programmanweisungen, Byte Code oder Statusänderungen aufweisen. Ähnlichkeit auf textueller Ebene ist über Text- und Tokenvergleiche analysierbar und auf semantischer Ebene über Input-Output Verhalten analysierbar (vgl. [Juergens 2010]).

Idiome: Ein Idiom beschreibt im Allgemeinen eine eigentümliche Sprachausprägung in syntaktischer, grammatischer oder struktureller Form [DudenIdiom 2017]. Speziell im Softwarebereich stellen Idiome ein Konstrukt dar, welches von Programmierern genutzt wird, um Sprachmerkmale nachzubilden, die nicht in der Programmiersprache enthalten sind [Snyder 81]. Das bedeutet Idiome sind Muster auf einer niedrigen Abstraktionsebene, die beschreiben, wie bestimmte Aspekte von Komponenten oder Beziehungen mit den Mitteln einer Programmiersprache umgesetzt werden [Buschmann 98]. Diese Idiome

umfassen vor allem Sprachkonventionen, typische Lösungsansätze, Code Strukturen und auch Implementierungen von Entwurfsmustern.

3. Vorgehen

Dieses Kapitel erläutert die Vorgehensweise, die zur Bearbeitung der Forschungsfrage verfolgt wurde.

3.1 Methodisches Vorgehen

Die Basis der Bachelorarbeit bildet eine Literaturstudie zu den Themen Redundanz, Code-Clones, Ähnlichkeit und Boilerplate-Code sowie ihrem Zusammenwirkungen.
Das methodische Vorgehen orientiert sich an dem Modell von Chitu Okoli [Okoli 2010]. Okoli beschreibt eine Literaturstudie mit vier Hauptphasen, die jeweils aus zwei Unterphasen bestehen.

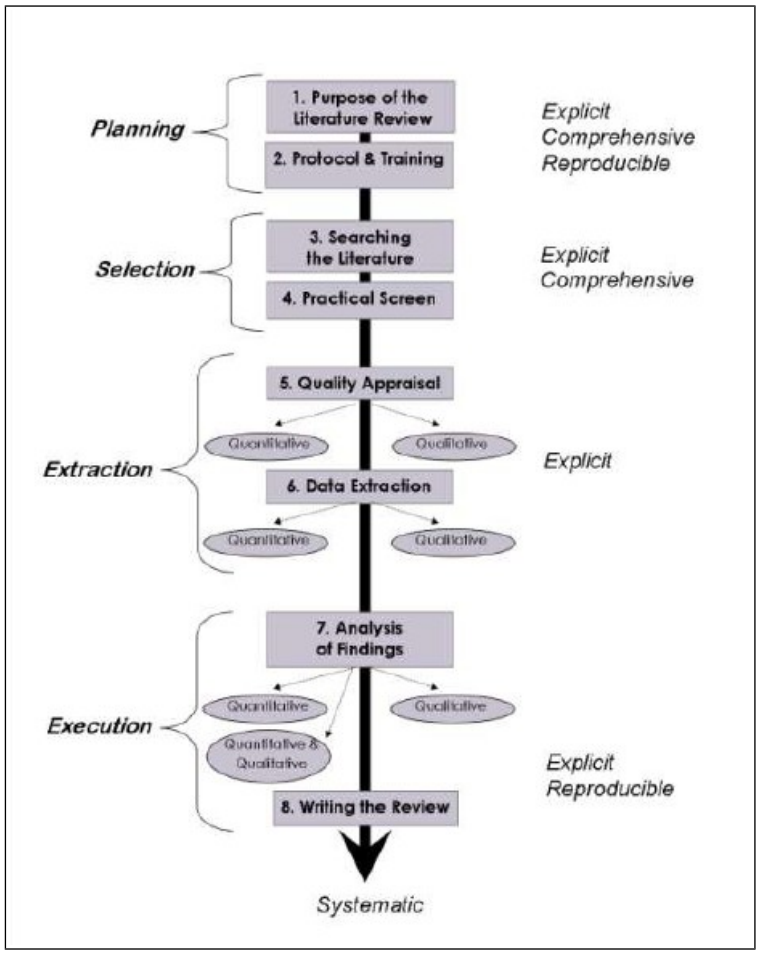

Abbildung 1: Phasenmodell der Literaturstudie aus Okoli [Okoli 2010]

Die erste Phase legt die genauen Untersuchungskriterien und Begrifflichkeiten fest. Es erfolgt die Eingrenzung anhand der untersuchten These. Während der zweiten Phase wird die Literaturrecherche angefertigt. Quellen, die sich nicht mit der Forschungsfrage beschäftigen, aber ähnliche Schlagwörter verwenden, werden ausgesondert. In der dritten Phase wird die Literatur anhand ihres Informationsgehaltes geordnet und die Informationen werden extrahiert. In der letzten Phase werden nun die nötigen Verbindungen zwischen den extrahierten Daten gezogen und es wird die abschließende Auswertung verfasst (vgl. [Okoli et al. 2010]).

3.2 Planung

Die Literaturstudie dient der Erarbeitung von aktuellen Erkenntnissen und Definitionen verschiedener Redundanzarten. Der Fokus liegt auf Redundanz im Bereich der Software Entwicklung. Redundanz aus den Bereichen Datenbanken oder Biologie sind nicht Forschungsgegenstand der Recherche.
Das Ziel ist somit eine Überprüfung der bestehenden Forschungen, welche sich mit einer „negativen“ Redundanz beschäftigen. Hiermit ist auch gezielt eingebrachte Redundanz ausgeschlossen, wie zum Zweck der Fehlertoleranz oder nicht ausgeführter beziehungsweise toter Code.
Die Literaturrecherche ist auf deutsch- und englischsprachige Literatur beschränkt. Unter dem Begriff „Literatur“ sind Fachartikel aus wissenschaftlichen Journals und Konferenzen sowie Fachbücher und Abhandlungen zu verstehen, welche sich mit der Thematik beschäftigen.

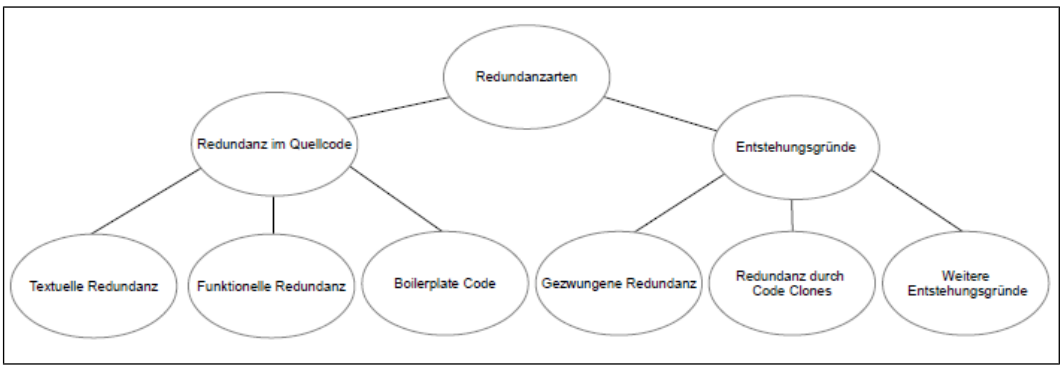

Abbildung 2: Ansätze zur Untersuchung der Redundanz

Aufgrund verschiedener Definitionen sind zwei Ansätze der Literaturrecherche ausgewählt worden.

Im ersten Ansatz wird von der Redundanz im Quellcode ausgegangen. Es wird untersucht, wodurch textuelle und funktionelle Redundanz im Quellcode entsteht, in welchen Formen diese auftritt und wie stark sie mit den Definitionen von Code-Clones übereinstimmen. Es wird auch geprüft, wie sehr Boilerplate-Code sich mit diesen Redundanzen überschneidet und ob Boilerplate-Code selbst eine Art der Redundanz darstellt.
Der andere Ansatz beschäftigt sich mit den Entstehungsgründen der Redundanzen. Es wird geprüft, wodurch sich gezwungene Redundanz auszeichnet und wodurch sie im Quellcode sichtbar wird. Zusätzlich wird untersucht, in welchem Zusammenhang das Klonen von Code-Fragmenten sich auf die Zunahme von Redundanz auswirkt. Im Verlauf der Literaturstudie wird erforscht, ob es weitere Entstehungsgründe gibt, wodurch sich weitere Arten der Redundanz festlegen lassen.

3.3 Selektion

Die Suche nach geeigneter Fachliteratur und Forschungsarbeiten bildet den ersten Schritt der zweiten Phase. Für die Untersuchung wurden aufgrund der wachsenden Bedeutung und Verbreitung der Softwareentwicklung vorwiegend englische Schlagwörter verwendet. Dadurch rücken vorwiegend englischsprachige Quellen in den Fokus.
Der Hauptteil der Quellensuche erfolgt über die Online-Literaturdatenbanken des *Institute of Electrical and Electronics Engineers*, der *Association for Computing Machinery* und *SpringerLink,* sowie *googlescholar.* Diese Plattformen eignen sich aufgrund des hohen Anteils an wissenschaftlichen Abhandlungen zu den Thematiken der Informatik und Softwareentwicklung.
Der erste Schritt zeichnet sich durch die Suche nach Fachliteratur anhand der in Phase Eins erarbeiteten Ansätze aus. Die Suchanfragen werden über einzelne Themen und deren Teilmengen gestellt.
Durch den zweiten Schritt erfolgt die Aufspaltung der Artikel nach ihrem Inhalt. Dies geschieht anhand des Titels, der Zusammenfassung und der Art der Nachforschung. Jene Artikel, die zur Beantwortung der Untersuchung dienen, werden in die Literatursammlung aufgenommen. Bedeutende Segmente der Artikel können durch eine *Rückwärtssuche* näher untersucht werden. Dies bedeutet, dass die Quellen der Artikel weiter betrachtet werden.
Bei Auswahl der Artikel wurde darauf geachtet, dass ein negativer Redundanzbegriff vorliegt oder, dass Ähnlichkeit und Code-Clones sich auf Redundanz beziehen oder anwenden lassen. Ein weiteres Kriterium ist die automatische Erkennung, Messung und

Beseitigung von Redundanz und Code-Clones. Diese können Rückschlüsse auf unterschiedliche Entstehungsweisen und Arten liefern.
Zur Literaturstudie wurden die in Phase Eins konkretisierten Suchbegriffe der zwei Ansatzpunkte herangezogen. Ausgehend vom ersten Ausgangspunkt wurden vorwiegend folgende Begriffe und Begriffsuntermengen genutzt:

- Redundancy in software developement and engineering
- Textual and functional redundancy
- Boilerplate-Code
- Redundancy in code
- Detection and measuring of redundancy
- Functional and textual equivalent code

Für den zweiten Ansatz für die Literaturrecherche wurden folgende Schlagwörter verwendet:

- Code Clones
- Forced redundancy
- Reasons for redundancy
- Code duplication
- Clone analysis and detection

Weiterführend zur Suche wurden die Schlagwörter durch die folgenden erweitert:

- Similarity
- Refactoring
- Reuse of software components

3.4 Extraktion

Nicht alle Artikel, die anhand der Zusammenfassung und des Titels ausgesucht wurden, sind für das Forschungsthema relevant. Daher werden die ausgesuchten Quellen anhand ihrer Qualität und der Menge ihrer Informationen beurteilt und hierarchisiert. Dieser erste Schritt erstellt somit eine Auflistung der Dokumente mit den thematischen Zuordnungen. Im zweiten Schritt werden die Daten extrahiert und systematisch aufgeschlüsselt. Die extrahierten Informationen wurden zunächst nach den Themen Redundanz, Code-Clone

und Ähnlichkeit geordnet. Die Untergliederung der Informationen zur Thematik Redundanz befindet sich im Gliederungspunkt *Ergebnisse*.

3.5 Ausführung

Die letzte Phase der Literaturstudie befasst sich mit der Zusammenführung der Information. Durch die Extraktion stehen alle nötigen Daten zur Verfügung, die nun für den gewählten Kontext aggregiert, integriert, interpretiert oder erklärt werden können. Der letzte Schritt befasst sich mit dem Schreiben der Literaturstudie. Der Fokus liegt auf dem Darlegen der Forschungsergebnisse und deren Zusammenhänge. Die Phase der Informationszusammenführung wird ebenfalls im Gliederungspunkt *Ergebnisse* verdeutlicht.

4. Ergebnisse

Die in der dritten und vierten Phase von Okoli beschriebenen Vorgänge, umfassen die Begutachtung und Auswahl der Forschungsartikel. Nach den Betrachtungen der zweiten Phase konnten 54 Artikel als relevant für die weiteren Untersuchungen eingestuft werden. Nach deren Begutachtung wurden 27 ausgesondert, so dass noch 27 für die Extraktion der Daten betrachtet wurden. Die ausgesonderten Dokumente befassen sich zu einem Großteil mit Redundanz aus der Textverarbeitung oder mit Studien von Algorithmen, wodurch diese sich nicht für eine Datenextraktion eigneten.
Die Ergebnisse der Extraktion sowie deren Anwendung auf den Bereich der Redundanz werden durch die folgenden Abschnitte dargelegt. Der Bezug auf die unterschiedlichen Arten wird mithilfe von Code-Clones verdeutlicht. Die Abschnitte umfassen zunächst die Definitionen und Entstehungsgründe sowie die Erkennungsmerkmale der Art. Weiterhin werden die Arten voneinander abgegrenzt.

4.1 Negative Software Redundanzen

Neben den eingangs erwähnten positiven Seiten wie der Fehlertoleranz eines Systems sorgen Redundanzen für erhebliche Probleme bei der Wartung eines Programms oder der Wiederverwendung von Softwarekomponenten. Dieses Problem kann sich häufig durch

fehlende Programmierkonzepte im Quellcode widerspiegeln. Als redundanter Code gilt hierbei jener Code, der durch eine geeignete Abstraktion, eine Wiederverwendungstechnik oder eine Art des Refactorings reduziert werden kann (vgl. [Menezes Leitao 2003]). Auch wenn Code Clones beziehungsweise das Duplizieren von Code einen Hauptgrund für redundanten Code darstellen, werden im Artikel *„Detection of Redundant Code using R2D2“* [Menezes Leitao 2003] noch weitere Gründe genannt. Diese umfassen neben der Duplikation auch die Verwendung von weit verbreiteten Idiomen, verschiedenen Entwurfsmustern oder schlicht Zufälle. Im Artikel *„Survey of Research on Software Clones“* [Koschke 2007] beschreibt es Rainer Koschke mit den Worten *„Although cloning leads to redundant code, not every redundant code is a clone.“*. Diese Aussage wird in verschiedenen Abhandlungen auf eine ähnliche Weise verwendet. Es ist daher davon auszugehen, dass Code Clones eine Untermenge der Redundanz bilden (vgl. [Bharathi 2014]). Die Verbindung für diese Annahme wird durchaus von dem Konzept der Ähnlichkeit gestützt. Diese scheint als Bindeglied zwischen den Bereichen der Code Clones und der Redundanz zu fungieren. Die Ähnlichkeit wird bereits zur Messung, Analyse und Auffindung von Code Clones verwendet und beschreibt wichtige Faktoren für die Redundanz. *„Two code fragments form a redundancy if their program text is similar. [...]“* [Bharathi 2014]. Es ist daher anzunehmen, dass man durch eine hinreichende Abstrahierung von Ähnlichkeiten in Quellprogrammen auf einen redundanten Inhalt auf einer textuellen Ebene schließen kann. Dies entspricht wiederum den Definitionen eines Clones des Typs Eins, Zwei oder Drei. Anders als bei diesen gegenständlichen Ähnlichkeiten lassen sich verhaltensbezogene Ähnlichkeiten nicht unbedingt durch Abstraktion sichtbar machen. Diese können unterschiedlich implementiert sein. Diese Ähnlichkeit beschreibt den Code-Clone des Typs Vier, da beide auf die Ausführung des Programms fokussiert und nicht auf den repräsentativen Status beschränkt sind.

Eine mögliche Darstellung des Sachverhaltes wird durch Listing 1 verdeutlicht. Beide Seiten erfragen zunächst die gewünschten Variablen. Die rechte Seite multipliziert diese miteinander, wohingegen die linke Seite je nach dem Vorzeichen von `x`, `z` und `y` addiert oder subtrahiert. Zuletzt geben beide Seiten den Wert für `z` aus. Zunächst sind hier die Zeilen von Eins bis Neun exakt identisch. Diese würden damit einen Typ-Eins-Clone darstellen. Der Unterschied liegt, abgesehen von der Ausgabe, in der Berechnung. Diese stellt einen Typ-Vier-Clone dar. Beide Programme weisen zwar nicht das gleiche Verhalten auf, jedoch ergibt sich beim selben Input der gleiche Output. Nach einer Abstraktion zeigen beide Fragmente eine Main-Funktion in Zeile Vier, gefolgt von einer Typdeklaration von

Variablen. Die Zeilen Sechs und Acht, sowie 19 auf der linken und Elf auf der rechten Seite sind Ausgaben und in den Zeilen Sieben und Neun wird etwas in das System eingelesen. Der Unterschied in der Berechnung spiegelt sich durch eine Schleife mit arithmetischen Operationen auf der linken Seite und durch eine arithmetische Operation auf der rechten Seite wieder.

```
1   #include <iostream>
2   using namespace std;
3
4   int main() {
5     int y,x,z;
6     cout << "x?"<< endl;
7     cin >> x;
8     cout << "y?"<< endl;
9     cin >> y;
10    z=0;
11    while (x>0){
12       z+=y;
13       x=x-1;
14    }
15    while (x<0){
16       z-=y;
17       x+=1;
18    }
19    cout << "z = " << z << endl;
20  }
```

```
1   #include <iostream>
2   using namespace std;
3
4   int main() {
5     int y,x,z;
6     cout << "x?"<< endl;
7     cin >> x ;
8     cout << "y?"<< endl;
9     cin >> y ;
10    z= x*y;
11    cout << "z = " << z << endl;
12  }
```

Listing 1: Multiplikation zweier Variablen in C++

Es wird bereits bei diesem eingehenden Beispiel deutlich, dass die verschiedenen Arten der Redundanz nicht zwangsweise unabhängig voneinander auftreten müssen und somit ähnlich wie Code-Clone-Typen ineinander übergehen.

4.2 Textuelle Redundanz

Nach [Johnson 93] ist die textuelle Redundanz jede Charakteristik des Quellprogramms, die von einem Kompressionsalgorithmus genutzt werden kann, um den Code in weniger Bits darzustellen. Als eine solche Charakteristik gilt dabei jede Wiederholung oder Asymmetrie, die ein Algorithmus entdecken und damit auch codieren kann [Johnson 93]. Zu diesem Ansatz lässt sich zusätzlich die repräsentative Ähnlichkeit heranziehen. Sequenzen bestehend aus Zeichen, die eine komplexe Struktur oder einen Text formen, sind repräsentativ ähnlich, wenn sie in ihrer Form, ihren Eigenschaften oder Charakteristiken ähnlich sind [Walenstein 2007]. Da diese Ähnlichkeit sich textuell, lexikalisch, syntaktisch oder strukturell ausprägen kann, beeinflusst sie die Definition für textuelle Redundanz. Für den Kontext der negativen Redundanz kann damit gelten, dass

textuelle Redundanz eine jede Charakteristik im Quellcode darstellt, die sich in textueller, lexikalischer, syntaktischer oder struktureller Ausprägung verdeutlicht. Diese Charakteristiken sind Wiederholungen beziehungsweise Duplikate, die durch geeignete Abstraktion oder Programmierkonzepte verringert werden können. Textuelle Redundanz gleicht in ihrer Form den Code-Clone Typen Eins, Zwei und Drei aufgrund der Nähe zur repräsentativen Ähnlichkeit. Außerdem kann diese Art der Redundanz in Form von funktionellen Code-Clones auftreten. Heruntergebrochen bedeutet dies, dass es sich bei textueller Redundanz um Code-Zeilen, ganze Funktionen oder Module handelt, die sich ähneln oder identisch sind. Anhand von Text- oder Token-basierender Vergleiche kann repräsentative Ähnlichkeit und damit auch textuelle Redundanz entdeckt werden [Juergens 2010].

Die textuelle Redundanz entsteht häufig durch *copy and paste* mit einer Anpassung der Code-Fragmente [Menezes Leitao 2003]. Diese Vorgehensweise ist eine einfache Form der Code-Wiederverwendung und führt die Programmierer zu einem schnellen Ergebnis. [Menezes Leitao 2003] benennt auch als Entstehungsgrund eine Beispiel-orientierte-Programmierweise, die oft in großen Software-Systemen praktiziert wird. Dabei orientieren sich die Programmierer an bestehenden Code-Fragmenten und kopieren dadurch oft die Struktur. Bereits [Johnson 93] benennt die Verwendung von weit verbreiteten Idiomen und Sprachkonventionen als Ursache für Redundanz. Ergänzt wird dieser Punkt durch die Anwendung von bekannten Entwurfsmustern, Algorithmen oder Datenstrukturen [Menezes Leitao 2003]. Durch die Umsetzung dieser Muster oder dem *copy and paste* entsteht ein Großteil der textuellen Redundanz.

Weiterhin kann textuelle Redundanz auch durch Zufall entstehen [Koschke 2007] (siehe 4.4.2 Zufällige Redundanz). Aufgrund einer zufälligen Verwendung von Sprachmitteln, unterschiedlichen Anforderungen oder mehreren Programmierern, die gleichzeitig an einem System arbeiten oder einem Programmierer zu unterschiedlichen Zeitpunkten, können Code-Fragmente wie das Listing 2 entstehen. Dieses stellt einen Code-Clone dar, der die Swap-Funktion implementiert. Diese vertauscht die Werte zweier Integer-Variablen auf der linken Seite und zweier String-Variablen auf der rechten Seite. In der Main-Funktion von Zeile Elf bis 21 werden jeweils zwei Variablen mit den dazugehörigen Werten angelegt. Diese werden ausgegeben, vertauscht und erneut ausgegeben. Bei der zweiten Ausgabe sind die Werte vertauscht. Die beiden Code-Fragmente sind sich sehr ähnlich und damit auch redundant. Sie unterscheiden sich lediglich in Typen und Bezeichnern, wodurch sie einen Code-Clone des zweiten Typs darstellen.

```cpp
#include <iostream>

using namespace std;

void swap(int& a, int& b) {
    int temp = a;
    a = b;
    b = temp;
}

int main()
{
    int a = 1;
    int b = 2;

    cout << "a: " << a << endl;
    cout << "b: " << b << endl;
    swap(a,b);
    cout << "a: " << a << endl;
    cout << "b: " << b << endl;
}
```

```cpp
#include <iostream>

using namespace std;

void swap(string& a, string& b) {
    string temp = a;
    a = b;
    b = temp;
}

int main()
{
    string First = "First";
    string Second = "Second";

    cout << "First: " << First << endl;
    cout << "Second: " << Second << endl;
    swap(First,Second);
    cout << "Not First: " << First << endl;
    cout << "Not Second: " << Second << endl;
}
```

Listing 2: Implementierung der Swap-Funktion in C++

```cpp
#include <iostream>

using namespace std;

template<typename T>
void my_swap(T& a, T& b) {
    const T temp = a;
    a = b;
    b = temp;
}

template<typename T>
void my_out(T& q,T& w){
    cout << "First: " << q << endl;
    cout << "Second: " << w << endl;
    my_swap(q,w);
    cout << "Not First: " << q << endl;
    cout <<"Not Second: " << w << endl;
}

int main()
{
    string First = "First";
    string Second = "Second";
    my_out(First,Second);

    int x = 1;
    int y = 2;
    my_out(x,y);
}
```

Listing 3: Implementierung der Swap-Funktion mithilfe Funktionstemplates in C++

Anhand der Definition der textuellen Redundanz sollte sich das Listing 2 auch abstrahierter darstellen lassen. Zu diesem Zweck sind die Fragmente des Listing 2 im Listing 3 zusammengefasst. Darin wurden die Swap-Funktion sowie die Ausgabe in jeweils einem Funktionstemplate verallgemeinert. Zur Übersicht wurde dabei die Ausgabe vom Tauschen der Variablen getrennt. Zur Vereinfachung wurde die Ausgabe leicht angepasst. Die

Funktionstemplates können nun unabhängig von Datentypen verwendet werden und werden zudem automatisch ausgeprägt.

4.3 Funktionelle Redundanz

Die funktionelle Redundanz impliziert zu einem großen Teil den Ansatz der textuellen Redundanz. Sie erweitert diese nach dem Konzept der semantischen Ähnlichkeit. Nach diesem Ansatz entspricht funktionelle Redundanz einer Ähnlichkeit oder Gleichheit von Code-Fragmenten. Nach [Carzaniga 2015] zeigt sich eine Gleichheit in der Beobachtung von Code-Fragmenten, wenn diese ausgehend von einem Anfangsstatus mit gleicher Programmumgebung und gleichem Input den selben Output erzeugen. Weiterführend wird angegeben, dass zwei Code-Fragmente eine Redundanz formen, wenn diese einer Gleichheit in der Beobachtung unterliegen und wenn die Fragmente in jedem Anfangsstatus, sowie jedem weiteren validen Status gleich sind. Die Fragmente können sich dabei zwischen den validen Status unterscheiden (vgl. [Carzaniga 2015]). In diesen Bereich fallen damit auch funktionell ähnliche Code-Clones [Wagner 2016] und strukturelle Code-Clones [Basit 2006]. Diese strukturellen Code-Clones beschreiben ähnliche Programmstrukturen auf verschiedenen Abstraktionsebenen mit ähnlichen Code-Fragmenten auf textueller Basis [Basit 2006]. Für die funktionelle Redundanz gilt damit, dass Code-Fragmente redundant sind, wenn diese sich bei identischem Input ähnlich oder gleich verhalten.

Diese Art der Redundanz zeigt sich häufig als Code-Clone des Typs Vier. Auch wenn textuelle Redundanz impliziert wird, muss funktionelle Redundanz nicht auf Quellcodeebene identisch oder ähnlich sein. Einige Gemeinsamkeiten lassen sich erst durch Muster auf Entwurfsebene entdecken [Rattan 2013]. Im Allgemeinen lassen sich Ähnlichkeiten im Verhalten von Code-Fragmenten durch Input-Output-Tests bestimmen [Juergens 2010]. Dies kann unter Umständen auch für textuelle Redundanz gelten.

Neben der Entstehung durch *copy and paste*, kommt funktionelle Redundanz verstärkt durch die Verwendung von Datenstrukturen, Algorithmen und Idiomen im Quellcode vor [Bharathi 2014]. Eine weitere Perspektive wird von [Menzes Leitao 2003] abgebildet. Es wird beschrieben, dass es häufig zu Redundanz kommt, da Funktionalitäten nicht genügend abstrahiert werden. Dies kann die Folge fehlender Sprachkonstrukte der Programmiersprache sein.

Die Struktur eines Stapels wird oft in der Programmierung genutzt, um Objekte oder Vorgänge der realen Welt zu beschreiben. Listing 4 zeigt einen einfachen Stapel mit der Push- und Pop-Funktion. Die rechte Seite verwendet zu diesem Zweck eine Tabelle. Beim Aufruf der Push-Funktion wird das Objekt an der hintersten Stelle eingefügt. Beim Aufruf der Pop-Funktion wird das hinterste Element der Tabelle entfernt und ausgegeben. Die rechte Seite erzeugt somit bei der Ausführung den Output `b a.`

Die linke Seite verwendet ebenfalls eine Tabelle zu Modellierung. Aufgrund des fehlenden Konzeptes der Klasse in Lua wird in diesem Code-Fragment eine Klasse nur imitiert. Das bedeutet, die Methoden werden als Element der Tabelle gespeichert. Aus dieser Lage ist es dann möglich, die Methoden aufzurufen. Die Push-Funktion ermöglicht die Übergabe mehrerer Elemente und weist diese nacheinander einem Platz in der Tabelle mit Indexangabe zu. Die Pop-Funktion ermöglicht es, eine beliebige Anzahl an Elementen aus der Tabelle zu entfernen. Um die Durchführungsfunktionen und die Ausgabe zu trennen, wurde eine List-Funktion verwendet, die bei der Programmausführung den Output `1 a 2 b` erzeugt.

Beide Fragmente besitzen nur wenig textuelle Redundanz. Sie verwenden jeweils eine Tabelle zum Speichern der Elemente des Stapels, auf der Einfüge-und Löschoperationen durchgeführt werden. Bei einem einzelnen Einlesen der Variablen auf der linken Seite, also `stack:push("a")` und `stack:push("b")`, befinden sich bei beiden Fragmenten die selben Werte an den gleichen Stellen der Tabelle. Bei der Ausgabe hingegen, werden die Werte auf der linken Seite nur ausgelesen, wohingegen die Werte der rechten Seite ausgegeben und danach entfernt werden. Die beiden Fragmente unterscheiden sich daher in bestimmten Spezifikationen ihrer Funktionen und der Vorgehensweise. Jedoch verfügen sie über eine ähnliche Funktionalität, wodurch sie redundant sind.

```lua
Stack = {}

function Stack:Create()

 local myStack = {}
 myStack._x = {}

 function myStack:push(…)
   if ... then
      local Element = {…}
      for _,v in ipairs(Element) do
        table.insert(self._x, v)
      end
    end
  end

  function myStack:pop(num)
    local num = num or 1
    local Inhalt = {}
    for i = 1, num do
      if #self._x ~= 0 then
        table.insert(Inhalt, self._x[#self._x])
        table.remove(self._x)
      else
        break
      end
    end
    return unpack(Inhalt)
  end

  function myStack:list()
    for i,v in pairs(self._x) do
      print(i, v)
    end
  end
  return myStack
end

stack = Stack:Create()
stack:push("a", "b")
stack:list()
```

```lua
Stack ={}

function Stack.push(Element)
   table.insert(Stack,Element)
end

function Stack.pop()
   return table.remove(Stack)
end

myStack = Stack

myStack.push("a")
myStack.push("b")

while(table.getn(myStack)>0) do
   print(mystack.pop())
end
```

Listing 4: Implementierung eines einfachen Stapels in Lua

4.4 Boilerplate-Code

Unter Boilerplate-Code versteht man nach der populär-wissenschaftlichen Quelle Wikipedia Code Sektionen, die an vielen Stellen implementiert werden, aber nur wenig oder keine Änderung besitzen [Wikipedia 2017]. Dieser Code ist *„(...) tiresome to write, and easy to get wrong“* [Lämmel 2003]. Da Boilerplate-Code an vielen Stellen implementiert ist, ist er äußerst anfällig für Änderungen [Lämmel 2003]. Nach diesen Definitionen handelt es sich um Code, der geschrieben werden muss, um kleine Aufgaben zu erledigen. Die Aufgaben sind dabei nicht essentiell für die Programmlogik, allerdings wichtig für die Ausführung. Ein Beispiel dafür, ist die Verwendung der Getter- und Settermethoden für den Variablenzugriff einer Klasse. Weitere mögliche Darstellungen an Boilerplate-Code sind Include-Guards in C++ oder die Verwendung eines Shebangs in Unix-Systemen zur Identifikation von Skriptsprachen. Nach der Definition von [Lämmel 2003] ist ein Beispiel für Boilerplate-Code die Verwendung von Funktionen, die sich selbst

aufrufen, um auf eine Datenstruktur zuzugreifen. Das Problem an dieser Stelle ist, dass in großen Programmen dieser Aufruf zu erheblichen Performanzproblemen führen kann.
Für die Thematik Boilerplate-Code sollte daher auch betrachtet werden, welche Operationen während der Ausführung einer Aufgabe redundant sind. In diesem Zusammenhang ist eine Iteration über eine Datenmenge redundanter als ein direkter Aufruf. In Listing 5 ist ein solches Schema dargestellt. Die `mytable` ist als Key-Value-Tabelle implementiert. Diese erlaubt den direkten Zugriff auf die Daten mithilfe der vergebenen Schlüsselwerte. Der Aufruf der Tabelle gibt damit den direkten Wert von `b` zurück. Über das `array` hingegen muss iteriert werden, um die Werte des Arrays zu erhalten. Dies ist in Zeile Zwölf in Form einer `for`-Schleife verdeutlicht. Die Zeile Elf gibt dabei den Wert `nil` zurück. Der direkte Zugriff auf einen Wert im `array` ist damit nicht gegeben. Die Zeilen Sechs bis Acht dienen dabei nur der Veranschaulichung, die Werte im `array` können auch auf andere Arten gesetzt werden, auf die kein direkter Zugriff besteht. Zusätzlich zu Boilerplate-Code, der auf der Funktionsausführung beruht, ist in Zeile Eins der Unix spezifische Shebang abgebildet, der für die Ausführung des Programms nötig ist.

```
 1 #!/usr/bin/lua
 2
 3 mytable = {a="X", b= "Value set by a function",c="X" }
 4 print(mytable["b"])
 5
 6 a="X"
 7 c="X"
 8 b= "Value set by a function"
 9
10 array = {a, b, c}
11 print(array["b"])
12 for index,value in ipairs(array) do print(index,value) end
```

Listing 5: Implementation einer Key-Value-Tabelle und eines Arrays in Lua

Weiterhin wird der Begriff des *Boiler-plating* von [Kapser 2008] verwendet. Dabei dient der Code als eine Art Template. Durch Sprachbeschränkungen ist eine geeignete Wiederverwendung des Codes nicht gewährleistet, somit wird bestehender Code nur angepasst. Diese Art der Wiederverwendung tritt oft bei HTML-Dokumenten auf. Dabei werden hmtl-, head- und body-Tags verwendet, um eine gewisse Struktur vorzugeben. Diese zählen zu optionalen Tags und müssen daher nicht verwendet werden. Daher sind sie redundant.

4.5 Entstehungsgrund-basierte-Redundanz

Die bislang vorgestellten Arten der Redundanz, sind aufgrund ihres Bezuges zum Quellcode eingeteilt wurden. Um zu verstehen, wie Redundanz entsteht, muss allerdings auch auf die Bedingungen geschaut werden, unter denen Code geschrieben wird. Anhand der Untersuchungen wurden die Ergebnisse in gezwungene und zufällige Redundanz eingeteilt. Beide Arten beschäftigen sich mit den Umständen, unter denen Redundanz entsteht.

4.5.1 Gezwungene Redundanz

Die Benennung „gezwungene Redundanz“ leitet sich aus den Aussagen von Rainer Koschke ab: *„Sometimes programmers are simply forced to duplicate (...)“*[Koschke 2007]. Er bezieht sich dabei auf den am häufigsten benannten Grund zur Duplikation von Quellcode, nämlich das Fehlen von bestimmten Sprachkonzepten, also die Existenz von Beschränkungen in den verwendeten Programmiersprachen [Menezes Leitao 2003]. Durch ein Fehlen von bestimmten Konstrukten können Anforderungen nicht vollständig umgesetzt werden und Idiome und Entwurfsmuster können nicht ausreichend abstrahiert werden. Dies erschwert zudem die Wiederverwendung von Code-Fragmenten und erhöht den Bestandteil von Duplikaten und Redundanz im Code.
Nach [Koschke 2007] kommt es häufig durch einen verzögerten Wiederverwendungsprozess zu Redundanz. Flexible Problemlösungen werden erst bei einem zureichenden Wissen über Funktionen und Variablen angefertigt. Durch weitere Faktoren, wie zum Beispiel Zeitdruck oder die Komplexität des Systems besteht die Möglichkeit, dass diese Redundanz weiterhin im Code verbleibt [Rattan 2013]. Für den *copy and paste*-Prozess benennt [Koschke 2007] zusätzlich noch die Möglichkeit, dass die Produktivität einiger Programmierer anhand des produzierten Codes gemessen wird. Dadurch wird es bekräftigt, Code zu duplizieren und anzupassen, um gewisse Anforderungen zu erfüllen, anstatt den Code wieder zu verwenden, durch eine Art des Refactorings oder neu zu entwickeln. Ähnlich dazu bestimmt auch die Phobie des frischen Codes, die [Rattan 2013] anspricht, das Verhalten der Programmierer. Aufgrund des bestehenden und funktionierenden Codes neigen Programmierer dazu Fragmente wiederzuverwenden oder zu duplizieren. Zuletzt können verschiedene Konzepte eine Redundanz im Quellcode hervorrufen. [Koschke 2007] benennt besonders die

Modularisierung. Bei Veränderungen der Anforderungen oder anderen Anpassungen müssen Veränderungen in vielen Modulen vorgenommen werden. In der Kombination mit Logging oder Parameter-Überprüfungen, die in den meisten Modulen implementiert sein müssen, entsteht eine zusätzliche Redundanz [Koschke 2007].

Ein solcher Zwang zur Redundanz besteht im Vergleich von Listing 6 und 7. Die Sprache C++ besitzt das Konstrukt einer Klasse, wohingegen Lua dieses nicht implementiert (siehe Erläuterung zum Lua-Idiom im Absatz Zufällige Redundanz). Die Klasse `Account` wurde in den Listings 6 und 7 mit den Methoden für das Abheben und Einzahlen auf den Bankaccount ausgestattet. Für Listing 6 war es möglich, eine Klasse mit öffentlichen sowie privaten Zugriffspunkten zu implementieren. Für Listing 7 hingegen ahmt ein Tabellenkonstrukt diese Funktionalität nach. Ein zusätzlicher redundanter Faktor zeigt sich bei der Einbindung der Variable `balance`. In Listing 6 ist es möglich, `balance` direkt zu verwenden, da die Variable als `private` deklariert wurde. In Listing 7 hingegen muss explizit gesagt werden, woher die Variable kommt. Dies geschieht durch den Tabellenzugriff `Account.balance`. Da die Methoden sich bereits auf die Tabelle `Account` beziehen, ist es möglich, `self.balance` zu verwenden. Dies grenzt zwar den Grad der textuellen Redundanz ein wenig ein, jedoch erfüllt die Tabelle als Klasse nicht vollständig die selbe Funktion wie eine Klasse in C++.

```
 1 #include <iostream>
 2
 3 using namespace std;
 4
 5 class Account {
 6 public:
 7     void deposit(int x){
 8      balance+= x;
 9     }
10     void withdraw (int y){
11      if (y<= balance){
12             balance-=y;
13              }
14      else {
14         cout << "insufficient funds" << endl;
16      }
17     }
18 private:
19     int balance=0;
20
21 };
```

Listing 6: Simpler Bankaccount in C++

4.5.2 Zufällige Redundanz

Im Gegensatz zur gezwungenen Redundanz wird der Code bei einer zufälligen Redundanz nicht von sprachlichen oder organisatorischen Faktoren beeinflusst. Die Code-Fragmente, in denen die Redundanz auftritt, werden damit unabhängig voneinander erstellt. Die Benennung leitet sich aus der Aussage

> *„Finally, coincidence is a kind of redundancy that arises when different programmers (or the same programmer at different times) need to implement identical functionality.“*

ab [Menezes Leitao 2003]. Mit diesen Voraussetzungen ist es wahrscheinlich, dass die Funktionen sich stark ähneln. Dieser Effekt tritt ebenfalls auf, wenn der Aufwand, eine neue Funktionalität zu schreiben, kleiner ist, als die bisherigen Bibliotheken des Systems zu durchsuchen [Menezes Leitao 2003].

In diese Kategorie der Redundanz fallen auch Idiome, Entwurfsmuster, Datenstrukturen und Algorithmen. Diese entstehen in redundanter Form, da sie weit verbreitet sind und ihre Funktion sich über die Jahre bewiesen hat. Gleichermaßen werden diese Strukturen stetig für ähnliche Aufgaben verwendet. [Koschke 2007] spricht im Kontext des *copy and paste* von Defiziten in der Bildung, also dass einige Programmierer nicht das volle Potential einer Sprache kennen. Wenn dies nicht der Fall ist, kann es allerdings auch sein, dass eine geeignete Abstraktion ignoriert wird beziehungsweise der Nutzen eines anderen Programmierkonzeptes übersehen wird [Koschke 2007]. Dieser Fall kann dabei auch bei dem Verbleiben von Redundanz im Quellcode auftreten. Der Programmierer übersieht oder ignoriert den Fakt, dass die Funktionalität auch auf eine andere Weise implementierbar ist. Beziehungsweise nach dem Beispiel von [Menezes Leitao 2003] ist der Nutzen, den der Programmierer von einer anderen Implementierung geringer als die Aufwendungen, die für den redundanzfreien Quellcode nötig sind.

Weiterführend spricht [Al-Ekram 2005] zusätzlich von *„Clones by Accident“*. Diese beschreiben Code-Fragmente, die zwar von einem Erkennungsalgorithmus entdeckt werden, jedoch nicht dupliziert wurden. Identifiziert werden sie aufgrund ihres ähnlichen Aussehens [Al-Ekram 2005].

Ein sehr häufiges Idiom für die Skriptsprache Lua ist das Nachahmen einer Klasse. Dies geschieht wie in Listing 7 über eine Tabelle. Dabei wird eine Meta-Tabelle als Prototyp eines Objektes angelegt (Zeilen Drei bis Acht). Beim Erstellen einer Instanz greift diese nun auf die Funktionalität des Prototyps zurück. Somit kann die erstellte Klasse `Account`,

ähnlich wie eine Klasse in anderen Programmiersprachen, ihre Methoden vererben. Die Verwendung solcher Idiome führt somit unweigerlich zu Redundanz, da das selbe Muster für die Modellierung der Umstände genutzt wird. Des weiteren können Sprachlimitationen dafür sorgen, dass diese zufällige Redundanz nur schwer oder gar nicht entfernbar ist. Zudem sorgen die verwendeten Idiome zwar für eine bestimmte Funktionalität, jedoch sind sie häufig umständlich implementiert beziehungsweise im Vergleich mit den Konstrukten anderer Programmiersprachen umständlicher.

```
 1   Account = {balance = 0}
 2
 3   function Account:new (o)
 4      o = o or {}
 5      setmetatable(o, self)
 6      self.__index = self
 7      return o
 8    end
 9
10    function Account:deposit (v)
11      self.balance = self.balance + v
12    end
13
14    function Account:withdraw (v)
15      if v > self.balance then error"insufficient funds" end
16      self.balance = self.balance - v
17    end
```

Listing 7: Implementierung eines simplen Bankaccounts
durch das Idiom einer Klasse in Lua [Ierusalimschy 2003]

4.6 Abgrenzung der Redundanzarten voneinander

Im Punkt der negativen Software-Redundanz wurde bereits erörtert, dass Code-Clones aufgrund der Ähnlichkeit, welche zumeist den menschlichen Einschätzungen entspricht, einen Teil der Redundanz darstellen. Die erarbeiteten Definitionen zeigen, dass unterschiedliche Arten der Redundanz zusammenhängen. Ausgehend von der Basis der textuellen Redundanz, wird diese auch von der Definition der funktionellen Redundanz impliziert. Dies begründet sich daher, dass man mithilfe von textueller Redundanz auch identische oder ähnliche Funktionen und Funktionalitäten erzeugen kann. Dies bedeutet also, dass eine textuelle Redundanz eine Untermenge der funktionellen Redundanz abbildet. In der umgekehrten Richtung gilt dies nicht. Eine funktionelle Redundanz stellt zwar immer eine ähnliche Funktionalität bereit, diese kann dabei auf unterschiedliche Arten implementiert sein. Dies zeigt sich besonders in Listing 4, wo bestimmte

Grundfunktionalitäten durch die Standard-Tabellen-Funktionen implementiert werden, die wiederum unterschiedlich eingebettet werden.

Textuelle und funktionelle Redundanz wirkt sich auf den Boilerplate-Code aus, dennoch ist dieser nicht vollständig mithilfe dieser Redundanz zu erklären. Die höchste Übereinstimmung entsteht bei der Nutzung von Shebangs, Getter- und Settermethoden, Include-Guards und der Template-Struktur. Die teilweise überflüssige Ausführung von Funktionen durch Rekursion oder Iteration wie in Listing 5 ist dabei zum Teil durch funktionelle Redundanz zu beschreiben. Dies wird durch Input-Output-Verhalten unterstützt, da die Key-Value-Tabelle und das Array die selben Werte besitzen, jedoch das Array durch die Art der Abfrage bereits in sich redundant ist [Juergens 2010].

Der allgemeine Bezug zwischen gezwungener und zufälliger Redundanz erschließt sich aus dem Kontext ihrer Entstehung. Beide basieren auf dem Wissen über die Programmierung und sind damit nicht disjunkt in ihrer Erscheinungsform. Die gezwungene Redundanz hängt dabei oft mit dem direkten Kopieren und Einfügen von Code-Fragmenten zusammen. Bei einer zufälligen Redundanz handelt es sich in der Regel um neu geschriebenen Quellcode, der aufgrund einer speziellen Funktionalität nötig ist. Es ist daher nicht klar zwischen diesen zwei Arten zu unterscheiden. Auf der Quellcodeebene treten sie auf eine ähnliche Art und Weise auf.

Beide dieser Formen stehen dabei der Redundanz im Quellcode gegenüber. Die textuelle und funktionelle Redundanz sind nicht konkret auf die gezwungene und zufällige Redundanz aufteilbar. Zunächst sind sie von *copy and paste* abhängig, wodurch man sie der gezwungenen Redundanz zuordnen kann und wodurch funktionelle Redundanz auch die textuelle impliziert. Allerdings ist es vor allem bei einer funktionellen Redundanz der Fall, dass diese von Funktionalität getrieben wird. Der erhöhte Einsatz von funktionsfähigen Konzepten, die sich bereits bewiesen haben, ist nun die Folge. Dies bedeutet, wenn sich durch Zufälle eine Redundanz bildet, erscheint diese häufig in Form einer funktionellen Redundanz. Gleichermaßen jedoch erzeugt gezwungene Redundanz auch eine funktionelle Redundanz. Diese implizite Schlussfolgerung leitet sich aus dem Kopieren von Code-Fragmenten ab. Das Kopieren erzeugt unweigerlich textuelle Redundanz, welche häufig auch eine funktionelle Ähnlichkeit aufweist. Diese Herleitung steht dabei unter dem Grundgedanken, dass das *copy and paste*-Prinzip vorwiegend bei Sprachbeschränkungen oder anderen Defiziten angewendet wird.

Zuletzt steht der Boilerplate-Code noch der Entstehungsgrund-basierten-Redundanz gegenüber. Auch der Boilerplate-Code kann durch gezwungene oder zufällige Redundanz

hervorgerufen werden. Getter- und Settermethoden sind dabei zum Beispiel der gezwungenen Redundanz zuzuordnen, da sich der Variablenzugriff oftmals nicht auf eine andere Art und Weise gestalten lässt. Der zufällige Anteil lässt sich vor allem durch das *Boiler-plating* von [Kapser 2008] beschreiben. Der Grund dafür liegt in den Konventionen, die sich für eine Sprache eingestellt haben. Diese Konventionen erzeugen direkt eine Struktur oder einen Aufbau, der von vielen Programmierern genutzt wird, wodurch auch ohne einen direkten funktionellen Zusammenhang Redundanz entsteht.

5. Fazit

Redundanz in Quellcode kann als eine vorteilhafte Technik eingesetzt werden, um einen gewissen Grad der Robustheit zu erreichen. Allerdings kann sie auch auch die Qualität des Quellcodes negativ beeinflussen, sodass Probleme in der Lesbarkeit oder bei der Wartung auftreten können. Weiterhin müssen kopierte Code-Fragmente stets einheitlich an Änderungen angepasst werden. Eine Vernachlässigung dieser nötigen Anpassungen könnte zu einem unerwünschten Programmverhalten führen.

Zu der Thematik Redundanz existieren viele Quellen. Eine Vielzahl dieser Quellen beschäftigt sich jedoch mit dem positiven Einsatz von Redundanz. Oftmals werden in diesem Kontext die Bereiche der Fehlertoleranz, der selbst-überprüfenden (Englisch: *self-checking*) und selbst-heilenden (Englisch: *self-healing*) Programme genannt. Dazu wird vor allem auf eine funktionelle Redundanz geachtet, um eine möglichst hohe Zuverlässigkeit zu gewährleisten [Carzaniga 2015].

Nur eine kleine Anzahl der Quellen beschäftigt sich in erster Linie mit negativer Redundanz, die sich schadhaft auf den Quellcode auswirken kann. Diese Quellen stellen zumeist Techniken und Programme vor, die eine Entdeckung der Redundanz im Quellcode möglich machen sollen. Eine Fokussierung auf klare Definitionen oder Unterteilungen in verschiedene Arten der Redundanz findet dabei nicht statt. Überwiegend wird von einer allgemeinen Definition ausgegangen, die anhand der Ähnlichkeit oder dem *copy and paste*-Prinzip konkretisiert werden. Ähnlich wie bei Code-Clones wird von einer angemessenen Betrachtungsgröße ausgegangen. Anhand dieses Merkmals, welches der menschlichen Einschätzung unterliegt, bedeutet dies, dass ein Teil der Redundanz unentdeckt bleibt.

Weiterhin beschränkt sich die Redundanz der Quellen auf eine Quellcodebasis, welche eine redundante Ausführung von Code-Fragmenten nicht betrachtet. Eine solche redundante

Ausführung bedeutet, dass Code mehrfach ausgeführt wird, oder dass Code-Duplikate mehrere Instanzen erstellen oder auf Datenstrukturen zugreifen, obwohl nur ein Zugriff beziehungsweise eine Instanz nötig ist. Dies könnte zur Folge haben, dass Programme möglicherweise unter bestimmten Systemvoraussetzungen nicht konsistent laufen. Diese Betrachtung setzt sehr stark am Aspekt der Performance an und ist je nach persönlicher Auffassung kein Bestandteil der Redundanz, da viele Programmierer unter dem Begriff nur eine Duplikation in textueller oder funktioneller Form verstehen.

Der größte Bereich der Forschung zur Redundanz stellt die Untersuchung von Code-Clones und Ähnlichkeit dar. Zu dieser Thematik existieren bereits Forschungsergebnisse und Konferenzen. Diese beschäftigen sich mit der Deklaration von Code-Clones, ihrem Auffinden in und ihrer Entfernung aus Programmen und dem Schaden, den Code-Clones in Softwaresystemen anrichten können. Die Redundanz als vollständige Betrachtungsgröße wird dabei kaum adressiert. Dabei steht es außer Frage, dass negative Redundanz gleichermaßen wie Code-Clones aus dem Quelltext entfernt werden sollten, jedoch ist die Erfassung von Redundanz wesentlich aufwendiger. Wie im Punkt *Entstehungsgrund-basierte-Redundanz* dargelegt wurde, entsteht ein Teil der Redundanz bereits durch den Entwicklungsprozess. Dazu müsste eine Prävention erfolgen, die die direkte Entstehung von Redundanz verhindert. Damit wäre es möglich, Redundanz für neu entwickelte Systeme vorzubeugen. Besonders auch für Altsysteme ist es von Bedeutung, die Redundanz und Code-Clones ausfindig zu machen. Durch die fortlaufende Entwicklung entsteht entsprechend viel Code, sodass es durch häufiges Kopieren zu Code-Clones kommt. Damit steigt die nötige Wartungsarbeit für das System. Die Vermeidung von Redundanz im Quellcode führt also kurzfristig zu einem erhöhten Aufwand, um den Code geeignet zu abstrahieren, jedoch besitzt sie das Potential, die Wartungsarbeit auf lange Sicht zu erleichtern.

Das Ziel der Arbeit war es herauszufinden, auf welche Weise sich Redundanz untergliedern lässt und wie diese mit Code-Clones in Verbindung stehen. Der Fokus wurde dafür auf Redundanz gelegt, die sich negativ auf den Quellcode auswirkt. Die Literaturstudie diente dieser Zielstellung zunächst nur bedingt, da nur wenige Quellen sich auf negative Redundanz fokussieren. Sie erlaubte jedoch einen grundlegenden Überblick über den bestehenden Forschungsbereich der Code-Clones. Durch die Erweiterung zur Thematik Ähnlichkeit wurde es möglich, die Arten von Redundanz anhand der bestehenden Strukturen herzuleiten. Die erarbeiteten Zusammenhänge aus den Quellen lassen jedoch

erahnen, dass eine rein quellcode-basierte Betrachtung nicht das vollständige Spektrum der Redundanz abdecken kann. Zu diesem Zweck ist eine weitere Untersuchung nötig.
Abschließend ist festzuhalten, dass eine gezielte Vermeidung von Redundanz zwar anzustreben ist, dies kann allerdings Auswirkungen auf andere Prinzipien der Programmierung haben. Zudem besteht die Möglichkeit, dass mehr Redundanz durch die Verwendung bestimmter Prinzipien hervorgerufen werden kann. Dies geschieht, wenn der Nutzen des Prinzips größer ist als der Nutzen davon, Redundanz zu vermeiden.

6. Ausblick

Anhand der gewonnenen Erkenntnisse der Arbeit lassen sich weiterführende Untersuchungsthemen ableiten.
Im erarbeiteten Kontext der negativen Software-Redundanz erscheint die Untersuchung von Redundanz bei der Ausführung von Programmen als sinnvoll. Dies umfasst die nicht betrachteten Bestandteile der Arbeit, wie etwa Code, der niemals ausgeführt wird, und Code, der überflüssige Aktionen ausführt, um minimale Änderungen zu bewirken. Gleichermaßen kann von Redundanz gesprochen werden, wenn die Speichervergabe und die Lebenszeit von Instanzen unzureichend optimiert ist. Also zum Beispiel, wenn eine Instanz länger existiert als diese wirklich genutzt wird, oder ein Array mehr Speicher vorbereitet hat, als wirklich gebraucht wird. Dies ist vor allem bei Programmiersprachen von Nutzen, in denen diese Aufgaben nicht automatisiert sind.
Des Weiteren kann ergründet werden, durch welche anderen Faktoren Redundanzentstehung beeinflusst wird. Dazu müsste man erforschen, welche Prinzipien der Programmierung sich auf die Entstehung der Redundanz auswirken. Gleichermaßen stehen dazu die Auswirkungen auf andere Prinzipien, die die Vermeidung von Redundanz mit sich bringen. Für diese Betrachtung erscheint es ebenso sinnvoll zu analysieren, wie sich der kurz- und langfristige Nutzen der Vermeidung von Redundanz verhält. Diese Nutzenanalyse könnte einen klaren Punkt schaffen, an dem es aus wirtschaftlicher und informatischer Sichtweise heraus lohnen kann, Redundanz aus Quellcode zu entfernen beziehungsweise ihrer Entstehung vorzubeugen.

Quellen

[Al-Ekram 2005] Al-Ekram, R., Kapser, C., Holt, R., Godfrey, M., *Cloning by Accident: An Empirical Study of Source Code Cloning Across Software Systems,* im International Symposium on Empirical Software Engineering, Noosa Heads, 2005

[Anil_kumar 2012] Anil kumar, G., Reddy, C., Govardhan, A., Ratna Raju, A., *Code duplication in Software Systems:A Survey,* im International Journal of Software Engineering Research & Practises Vol.2, Hyderabad, 2012

[Basit 2006] Basit, H., *Analyis and Semi-Automated Detection of Design-Level Similarities in Software,* Singapur, 2006

[Bellon 2007] Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E., *Comparison and Evaluation of Clone Detection Tools,* im IEEE Transactions on Software Engineering Vol.33, 2007

[Bharati 2014] Bharathi, R., Dhivya, M., *Textual and Metric Analysis to Figure out all Types of Clones: In Detection Method,* im International Journal of Scientific Engineering and Technology Vol.3 S. 326-329, 2014

[Boilerplate-Code] Wikipedia, URL: https://en.wikipedia.org/wiki/Boilerplate_code Zuletzt gelesen am 31.07.2017

[Buschmann 1998] Buschmann, F., *Pattern-orientierte Software-Architektur. Ein Pattern-System,* Addison-Wesley (Hrsg.) , 1998

[Carzaniga 2009] Carzaniga, A., Gorla, A., Pezzè, M., *Handling Software Faults with Redundancy,* Lugano, 2009

[Carzaniga 2015] Carzaniga, A., Mattavelli, A., Pezzè, M., *Measuring Software Redundancy,* im International Conference on Software Engineering, Florenz, 2015

[DudenRedundanz] Duden (Hrsg.), Redundanz, URL http://www.duden.de/rechtschreibung/ 2017 Redundanz, Zuletzt gelesen am 01.08.2017

[DudenIdiom 2017] Duden (Hrsg.), Idiom, URL http://www.duden.de/rechtschreibung/Idiom , Zuletzt gelesen am 01.08.2017

[IEEE Standard Vocabulary 2010] IEEE Standars Board, *Systems and software engineering-Vocabulary, 2010*

[Ierusalimschy 2003] Ierusalimschy, R., *Programming in Lua*, Kapitel 16, 2003

[Johnson 93] Johnson, J., *Identifying Redundancy in Source Code using Fingerprints,* Conference of the Centre for Advanced Studies on Collaboration research: software engineering Vol 1, Seite 171-183, 1993

[Juergens 2010] Juergens, E., Deissenboeck, F., Hummel, B., *Code Similarities Beyond Copy & Paste,* 14th European Conference on Software Maintenance and Reengineering, Madrid, 2010

[Kapser 2008] Kapser, C., Godfrey, M., *„Cloning considered harmful" considered harmful:patterns of cloning in software*, Empirical Software Engineering Vol. 13, Seite 645-692, 2008

[Koschke 2007] Koschke, R., *Survey of Research on Software Clones,* Internationales Begegnungs- und Forschungszentrum für Informatik, Schloss Dagstuhl, 2007

[Lämmel 2003] Lämmel, R., Jones, S., *Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming,* ACM SIGPLAN international workshop on Types in languages design and implementation, Seite 26-37, 2003

[Menezes Leitao IEEE 2003] Menezes Leitao, A., *Detection of Redundant Code using R2D,* Third International Workshop on Source Code Analysis and Manipulation, Amsterdam, 2003

[Okoli 2010] Okoli, C., Schabram, K., *A Guide to Conducting a Systematic Literature Review of Information Systems Research, 2010*

[Rattan 2013] Rattan, D., Bhatia, R., Singh, M., *Software clone detection:A systematic review,* Information and Software Technology 55, Seite 1165-1199, 2013

[Snyder 81] Snyder, L., *Recognition and Selection of Idioms for Code Optimization,* Acta Infomatica 17, Seite 327-348 , Springer-Verlag, 1981

[Topsøe 1974] Topsøe, F, *Informationstheorie. Eine Einführung*, Vieweg+Teubner Verlag, Wiesbaden. 1974

[Wagner 2016] Wagner, S., Abdulkhaleq, A., Bogicevic, I., Ostberg, J.-P., Ramadani, J., *How are functionally similar code clones syntactically different? An empirical study and a benchmark,* 2016

[Walenstein 2007] Walenstein, A., El-Ramly, M., Cordy, J., Evans, W., Mahdavi, K., Pizka, M., Ramalingam, G., Wolff von Gudenberg, J., Kamiya, T., *Similarity in Programs,* Internationales Begegnungs- und Forschungszentrum für Informatik, Schloss Dagstuhl, 2007

[Wortschatz 2017]: Universität Leipzig, URL http://corpora.informatik.uni-leipzig.de/de/res?corpusId=deu_newscrawl_2011&word=Redundanz , Zuletzt gelesen am 28.05.2017

Ehrenwörtliche Erklärung

Ich versichere, dass ich die Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Darüber hinaus versichere ich, dass die elektronische Version der Bachelorarbeit mit der gedruckten Version übereinstimmt.

_____

Groitzsch , 11.08.2017