FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Fachhochschule Aachen, Campus Jülich

Fachbereich 09 - Medizintechnik und Technomathematik
Studiengang Angewandte Mathematik und Informatik

Seminararbeit

Evaluation von Legacy-Code und geeigneter Analyse-Methoden zur Strukturierung

Autorin:
Carla Sophie Gengnagel, 3277052

Betreuer:
Prof. Dr. rer. nat. Gerhard Dikta
Philipp Brämer, M. Sc.

Aachen, 20. Dezember 2022

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Seminararbeit mit dem Thema

Evaluation von Legacy-Code und geeigneter Analyse-Methoden zur Strukturierung

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war. Ich verpflichte mich, ein Exemplar der Seminararbeit fünf Jahre aufzubewahren und auf Verlangen dem Prüfungsamt des Fachbereiches Medizintechnik und Technomathematik auszuhändigen.

Name: Carla Sophie Gengnagel
Aachen, den 20. Dezember 2022

Unterschrift der Studentin

Zusammenfassung

In vielen Unternehmen sind schwer verständliche Legacy-Systeme vorzufinden. Bestehende Abhängigkeiten sind kaum nachvollziehbar und die Umsetzung von Änderungen erweist sich als schwierig. Die Software-Systeme sind infolgedessen unflexibel gegenüber Marktänderungen. In dieser Arbeit wird ein konkretes Legacy-System, bestehend aus Cobol-Programmen, betrachtet. Es werden drei Verfahren vorgestellt, die potenziell geeignet sind, die Struktur des Systems anhand der zur Verfügung stehenden Daten zu analysieren. Dabei handelt es sich um einen genetischen Algorithmus, eine Netzwerkanalyse durch Clustering sowie ein grobes Clustering eines *Artifact Dependency Graph*. Die Strukturanalyse soll zum Verständnis des Systems beitragen, zu einer leichteren Nachvollziehbarkeit der Abhängigkeiten führen und eine Grundlage für zukünftige Refaktorisierungs-Arbeiten bilden, welche die Flexibilität des Systems gegenüber Marktänderungen erhöhen. Die Eignung der drei Verfahren wird anschließend diskutiert.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Motivation und Zielsetzung

Um 1950 wurden die ersten Programme in Assemblersprachen geschrieben, in den darauffolgenden Jahren auch in höheren Programmiersprachen, wie z. B. Fortran ab 1957 und Cobol ab 1960 (Wolf, 2020, S. 235-240). So existieren in Unternehmen noch Programme, die bereits vor mehreren Jahrzehnten erstellt und teilweise seitdem nicht mehr verändert wurden. Beim Umgang mit Systemen, die aus solchen älteren Programmen bestehen, treten oft Verständnisschwierigkeiten auf. Dies ist unter anderem auf den Aufbau und die Formulierungen sowie die fehlende oder veraltete Dokumentation des Codes zurückzuführen. Hinzu kommt, dass die für die Systeme Zuständigen meist nicht die ursprünglichen Entwickler sind, die z. B. aufgrund von Arbeitsbereichswechseln oder Renteneintritten nicht mehr zur Verfügung stehen. (Kirchmayr, Moser, Nocke, Pichler & Tober, 2016)

Unternehmen und somit auch ihre Software-Systeme müssen schnell auf Marktänderungen reagieren (Fuhr, Horn & Riediger, 2011). Neben Wartungsarbeiten am Code ist es daher erforderlich, neue Funktionalitäten hinzuzufügen oder Änderungen umzusetzen, wie z. B. die Verwendung einer anderen Programmiersprache. Zur sicheren Durchführung der Änderungen sind Änderungs- und Testpunkte zu identifizieren, um entsprechende Tests erstellen zu können (Feathers, 2020, S. 42). Um diese Punkte bestimmen bzw. die Auswirkungen von Änderungen einschätzen zu können, ist es essentiell, die Abhängigkeiten zu kennen, die zwischen den zu ändernden System-Komponenten und anderen Bestandteilen des Systems bestehen.

In dieser Arbeit sollen existierende Methoden zur Analyse der Struktur eines Software-Systems beschrieben und diskutiert werden. Die Strukturanalyse soll dazu beitragen, das System zu verstehen und Abhängigkeiten zu erkennen. Zudem soll sie eine Grundlage für zukünftige Refaktorisierungs-Arbeiten bilden, welche eine möglichst schnelle Umsetzung von Änderungen ermöglichen und somit die Flexibilität des Systems gegenüber Marktänderungen erhöhen. Die Analysemethoden sollen dabei im Kontext eines Software-Systems, bestehend aus Cobol-Programmen, betrachtet werden.

Diese Arbeit ist wie folgt gegliedert: In Kapitel 2 werden die theoretischen Grundlagen zu *Legacy Code* erläutert. Kapitel 3 befasst sich mit den Daten, die für das zu betrachtende System zur Verfügung stehen. In Kapitel 4 werden drei Verfahren zur Strukturanalyse von

Software-Systemen beschrieben. Anschließend werden sie und ihre Eignung für das konkrete System in Kapitel 5 diskutiert. Die Arbeit schließt mit einer Zusammenfassung der Ergebnisse und einem Ausblick in Kapitel 6.

2 Legacy Code

Dieses Kapitel fokussiert sich auf die theoretischen Grundlagen, die es zum einen erlauben, den Zustand nachzuvollziehen, in dem sich das Software-System befindet, und zum anderen die Identifikation der System-Komponenten und der bestehenden Abhängigkeiten ermöglichen. Der Begriff *Legacy Code* wird in Abschnitt 2.1 als Bezeichnung für Code eingeführt, wie er in dem zu betrachtende System vorliegt. Da das System in Cobol verfasst ist, geht Abschnitt 2.2 auf die Struktur von Cobol-Programmen und ihre Schnittstellen ein. In Abschnitt 2.3 wird die im System vorliegende monolithische Architektur mit der alternativen Microservice-Architektur verglichen.

2.1 Definition

Das englische Wort *legacy* wird im Deutschen mit *Erbe* oder *Hinterlassenschaft* übersetzt. Daraus ergibt sich folgende Definition: „Legacy Code ist Code, den wir von jemand anderem übernommen haben“ (Feathers, 2020, S. 16). Die Bezeichnung *Legacy Code* wird aber unter anderem auch für schwer verständlichen und schwer änderbaren Code verwendet (Feathers, 2020, S. 16), also solchen, der schlecht zu unterhalten ist (Sahraoui et al., 2002) und fehlendes Potenzial zur Weiterentwicklung besitzt (Barbier & Recoussine, 2015, S. 14). Auch das Alter des Systems spielt eine Rolle (NASCIO, 2008). Dem Begriff werden allerdings auch positive Eigenschaften zugeordnet, wie Zuverlässigkeit oder, dass Legacy Code geschäftskritische Logik abbildet. Obwohl die Programmiersprache eines Systems nicht bestimmt, ob es sich um ein Legacy-System handelt, baut ein Großteil der Legacy-Systeme auf Cobol auf. (Barbier & Recoussine, 2015, S. 14) Zudem liegen oft monolithische Strukturen in Legacy Code vor (Sahraoui et al., 2002), auf die in Abschnitt 2.3 näher eingegangen wird.

2.2 Cobol-Struktur und Schnittstellen

Ein Cobol-Programm besteht aus vier Teilen bzw. DIVISIONS. Die DATA DIVISION ist eine davon. In Abschnitten, zu denen die WORKING-STORAGE SECTION und die LINKAGE SECTION gehören, erfolgt in der DATA DIVISION die Beschreibung der im Programm

verwendeten Daten. Ein weiterer Teil des Programms ist die PROCEDURE DIVISION. Sie enthält die Anweisungen, welche beim Aufruf des Programms ausgeführt werden. Bei Bedarf besteht die Möglichkeit einer Unterteilung dieser Anweisungen in SECTIONS. Die Cobol-Anweisungen der PROCEDURE DIVISION werden anhand der von Kontrollflusselementen bestimmten Reihenfolge oder sequentiell abgearbeitet. Ein Kontrollflusselement ist beispielsweise der PERFORM-Befehl. Er erlaubt es, eine SECTION von einer beliebigen Stelle der PROCEDURE DIVISION aus aufzurufen und vor den sequentiell folgenden Befehlen auszuführen. (Hambeck, 2019, S. 21, 24-25, 44, 83-84; Schwickert, 2018, S. 135, 166)

Der Aufruf anderer Programme erfolgt mit der CALL-Anweisung. Analog zur PERFORM-Anweisung findet auch hier nach Ausführung des Moduls ein Rücksprung zum nächsten Befehl nach der Aufruf-Anweisung statt. Um beim Aufruf Daten übergeben zu können, müssen die entsprechenden Datenfelder im aufrufenden Programm sowohl in der WORKING-STORAGE SECTION deklariert sein als auch in der USING-Klausel angegeben werden, welche die CALL-Anweisung erweitert. (Schwickert, 2018, S. 165-166, 172-173) Dies ist in Abbildung 2.1 dargestellt.

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 DATENFELD                    PIC X(5).

PROCEDURE DIVISION.
CALL :Name des aufzurufenden Programms: USING DATENFELD
```

Abbildung 2.1: Beispielhafte Darstellung eines aufrufenden Programmes

In dem aufzurufenden Programm werden die Datenfelder, wie in Abbildung 2.2 zu sehen, in der LINKAGE SECTION deklariert und über eine USING-Klausel an die Überschrift PROCEDURE DIVISION angebunden (Schwickert, 2018, S. 172-173).

```
DATA DIVISION.
LINKAGE SECTION.
01 DATENFELD                    PIC X(5).

PROCEDURE DIVISION USING DATENFELD.
```

Abbildung 2.2: Beispielhafte Darstellung eines aufgerufenen Programmes

Des Weiteren kann die CALL-USING-Anweisung um die Angabe BY REFERENCE ergänzt werden, wodurch sowohl das aufrufende als auch das aufgerufene Programm auf denselben Speicherbereich für übergebene Datenfelder zugreifen. Bei der Verwendung einer der alternativen Angaben, BY CONTENT oder BY VALUE, wird nicht auf denselben Speicherbereich zugegriffen. Änderungen an den Datenfeldern im aufgerufenen Programm beeinflussen somit

nicht das aufrufende Programm. Eine fehlende Spezifikation ist mit der Angabe BY REFERENCE gleichzusetzen. (IBM, 2021) Zudem ist es möglich, über das COPY-Statement vorformulierten Code, genannt COPY-Strecken, zur Kompilierzeit in ein Programm einzubinden. Die eingebundenen COPY-Strecken können wiederum COPY-Statements enthalten. Eine solche Verschachtelung kann jedoch nicht rekursiv aufgebaut sein. (IBM, 2022)

2.3 Monolithische Architektur versus Microservices

Die Software-Architektur ist die grundlegende Struktur einer Software. Zu den am weitesten verbreiteten Software-Architekturen gehören die monolithische Architektur und die Microservice-Architektur. (Gos & Zabierowski, 2020)

Als Monolith bezeichnet Newman (2020) ein System, das als Ganzes bereitgestellt bzw. deployt wird. Neben der Deployment-Abhängigkeit liegt häufig auch auf der Implementierungsebene ein hohes Maß an Abhängigkeiten zwischen den Teilen des Systems vor. (Newman, 2020, S. 12, 15)

Die Microservice-Architektur ist eine verteilte Architektur (Richards & Ford, 2020, S. 129-130). Eine Anwendung, die entsprechend dieser Architektur aufgebaut ist, besteht aus kleineren, unabhängigen Komponenten, die über definierte Schnittstellen miteinander kommunizieren. Jede dieser als Microservices bezeichneten Komponenten kann wiederum als Anwendung betrachtet werden, für die z. B. die einzusetzende Plattform sowie die Programmiersprache und die Entwicklungsmethodik bestimmt werden können. Auch die Skalierung und Bereitstellung können individuell gehandhabt werden. (Xiao, Wijegunaratne & Qiang, 2016)

Bei der Festlegung der Microservice-Grenzen werden Kopplung und Kohäsion berücksichtigt, denn sie dienen als Metriken zur Messung von Modularität. Die Kohäsion gibt den Grad der Zusammengehörigkeit der Teile an, die einer Komponente zugeordnet wurden, wobei die Zusammengehörigkeit auf verschiedene Weisen definiert werden kann, beispielsweise anhand der Funktionalität. Die Kopplung beschreibt den Grad der Abhängigkeit zwischen Komponenten. Angestrebt wird eine möglichst lose Kopplung und hohe Kohäsion, um nach Constantines Gesetz ein stabiles System zu erhalten. (Newman, 2020, S. 17-18; Richards & Ford, 2020, S. 40-41, 44)

Im Vergleich zu einem monolithisch aufgebauten System ist ein aus Microservices bestehendes System besser verständlich. Zudem ist bei Änderungen nur lokal im betroffenen Microservice oder an der betroffenen Schnittstelle mit Auswirkungen zu rechnen, wenn eine

Microservice-Architektur vorliegt. Dementsprechend können Änderungen schneller erfolgen. Auch Änderungen, die das ganze System betreffen, wie die Überführung in eine andere Programmiersprache, können schneller und risikoärmer durchgeführt werden, da Teams parallel die verschiedenen Services umstellen können und Probleme nicht automatisch zu einem Ausfall des vollständigen Systems führen. (Newman, 2015, S. 25-28; Newman, 2020, S. 2, 6)

3 Daten

Dieses Kapitel geht auf die Daten ein, die für das zu betrachtende Legacy-System zur Verfügung stehen. Auch die Extraktion dieser Daten wird betrachtet. Informationen über die sich im System befindlichen Objekte sowie deren Eigenschaften und Relationen untereinander liegen im Metadaten-Repository MMR (Method Manager) vor, welches für diese Arbeit zur Verfügung steht (Holzem, 2022c). Jedem Objekt wird eine automatisch generierte, eindeutige und mit einer Typkennung beginnenden Zeichenkette zugeordnet. Diese verwendet der MMR als Name, unter dem er das Objekt verwaltet. (Holzem, 2022d, 2022b)

Im Rahmen dieser Arbeit wird eine Teilmenge der verfügbaren Objekte betrachtet. Diese sind die Programmsourcen und die Copy-Strecken. Vernachlässigt werden unter anderem die zu den Programmsourcen gehörenden Lademodule und Objekte, die der Verwaltung und Dokumentation anderer Objekte dienen. Lediglich die Objekte, die den Quellcode des Systems repräsentieren, und die Relationen zwischen ihnen werden betrachtet. Bei den Relationen handelt es sich um das Einbinden von Copy-Strecken und das Aufrufen von Programmen bzw. Programmsourcen. (Holzem, 2022e)

Bei der Extraktion der Daten ist die Betrachtung der Bauteilgruppen wesentlich. Diese ermöglichen es, MMR-Objekte Systemen bzw. System-Komponenten zuzuordnen. Statt für jedes Objekt einzeln können Zuständigkeiten und Rechte auf der Ebene der Bauteilgruppen verwaltet werden. Bei der Einteilung gilt, dass ein Bauteil genau einer Bauteilgruppe zugeordnet ist. Neben Programmsourcen und Copy-Strecken kann auch eine Bauteilgruppe wiederum Bauteil einer Bauteilgruppe sein, sodass eine hierarchische Gruppierung entsteht. Als Wurzel fungiert dabei die Bauteilgruppe *A0000000*. (Holzem, 2022a)

Die in dieser Arbeit betrachteten Objekte und ihre Abhängigkeiten sind in Abbildung 3.1 als ER-Diagramm in Chen-Notation dargestellt. Die Attribute der Objekte bzw. der Entitäten sind aus Gründen der Übersichtlichkeit nicht im ER-Diagramm abgebildet.

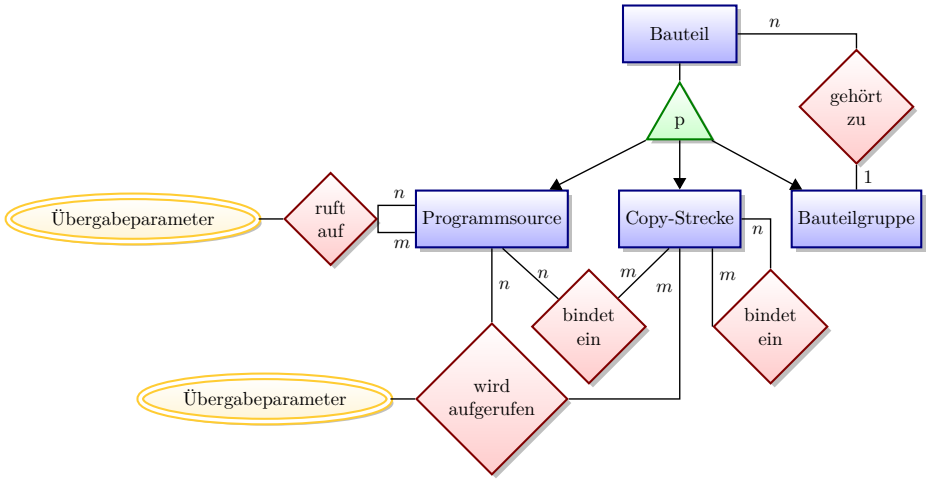

Abbildung 3.1: ER-Diagramm zur Darstellung der MMR-Objekte und ihrer Relationen

Aufgrund der großen Datenmenge, die in *A0000000* vorzufinden ist, wird zunächst die Bauteilgruppe *A0000023* zur Durchführung der Datengewinnung ausgewählt. Sie ist eine der 76 Bauteilgruppen, die direkt in Bauteilgruppe *A0000000* liegen.

In einem ersten Schritt wird ein MMR-Befehl über die PC 3270-Umgebung ausgeführt, um eine Liste aller Programmsourcen zu erhalten, die in *A0000023* enthalten sind. Der Befehl beinhaltet in einem ersten Schritt die Bestimmung aller Bauteilgruppen, welche *A0000023* direkt oder indirekt zugeordnet sind. Für jede dieser Bauteilgruppen werden dann alle Programmsourcen bestimmt, die dieser Bauteilgruppe direkt zugeordnet sind. Zur Weiterverarbeitung wird die Liste der Programmsourcen mithilfe des FTP-Clients außerhalb des Host-Umfeldes in einer Datei abgespeichert. Ein Ausschnitt der Liste ist in Abbildung 3.2 zu sehen.

```
PM-QV1497   UXX053   UXX053  AOM (Cobol recursive)                    BG-A000003M
PM-QV1499   TXX090   TXX090 Checkpointschreibung IMS                   BG-A00000GM
PM-QV3279   UXX130   UXX130 Checkpoint/Restart mit Fehlerbehandlung    BG-A00000GM
PM-QV4173   UVVAOM   UVVAOM Hülle zum Aufruf AOM (UXX053)             BG-A000003M
PM-QV5547   UVVPRZ   UVVPRZ   Prüfziffernberechnung Modulo 9          BG-A00003M4
```

Abbildung 3.2: Auszug der Programmsourcen-Liste

Pro Zeile der Liste werden die Informationen zu einer Programmsource dargestellt. Dazu gehören der MMR-Name, der eigentliche Name der Programmsource, auch als *Alias kurz* bezeichnet, eine Kurzbeschreibung und die Bauteilgruppe, zu der die Programmsource gehört. Der MMR-Name der Programmsource beginnt dabei mit *PM-*, der MMR-Typkennung für Programmsourcen. *BG-* ist die Kennzeichnung für Bauteilgruppen. Eine Liste der in *A0000023* enthaltenen Copy-Strecken kann ebenfalls extrahiert werden. Die Gewinnung erfolgt analog zur Extraktion der Programmsourcen-Liste.

Die zu den Bauteilen von *A0000023* gehörenden Quellcode-Dateien werden mithilfe des FTP-Cients lokal abgelegt. Sie sind jeweils dem Unterordner zugeordnet, der den Namen der Bauteilgruppe trägt, der sie zugeordnet sind.

Für die Ermittlung der Relationen zwischen den Programmsourcen und Copy-Strecken wird ein Java-Projekt erstellt, da über den MMR keine Informationen bezüglich der Parameter zur Verfügung stehen, die bei Programmaufrufen übergeben werden. Die Programmsourcen, die Copy-Strecken und ihre Beziehungen werden über Instanzen der Klassen *Component* und *CallParameter* dargestellt, welche in Abbildung 3.3 in UML-Schreibweise zu sehen sind.

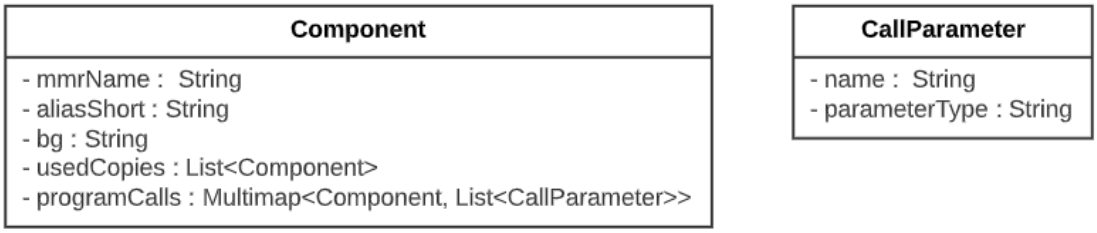

Abbildung 3.3: UML-Darstellung der Klassen *Component* und *CallParameter*

Eine Instanz der Klasse *Component* stellt eine Programmsource oder eine Copy-Strecke dar. Sie enthält den MMR-Namen, den eigentlichen Namen (*Alias kurz*) und die Bauteilgruppe, der das Objekt zugeordnet ist. Weitere enthaltene Informationen sind Programmaufrufe und verwendete Copy-Strecken. Die Typkennung des MMR-Namens ermöglicht die Identifikation der Instanz als Programmsource oder als Copy-Strecke. Die gewählte Darstellungsform für die Programmaufrufe ist eine Multimap, da pro Programmaufruf das aufgerufene Programm den übergebenen Parametern zugeordnet werden kann. Zudem ist es möglich, den wiederholten Aufruf eines Programmes, potenziell mit unterschiedlichen Übergabeparametern, abzubilden. Eine Instanz der Klasse *CallParameter* stellt einen solchen Parameter dar und enthält den Namen des Parameters sowie die Art der Übergabe, z. B. als Referenz.

Erstellt werden die Component-Objekte, indem die Zeilen der Programmsourcen-Liste und der Copy-Strecken-Liste durchgegangen und die enthaltenen Informationen extrahiert werden. Über die Angabe des Ablageortes der Quellcode-Dateien und den Namen der Bauteilgruppe wird pro Component-Instanz bestimmt, wo der zugehörige Quellcode abgelegt ist. Die darin enthaltenen Programmaufrufe und COPY-Statements werden der Instanz zugeordnet. Anschließend werden die gewonnen Objekte abgespeichert. Dieser Ablauf ist in Abbildung 3.4 als Sequenzdiagramm dargestellt.

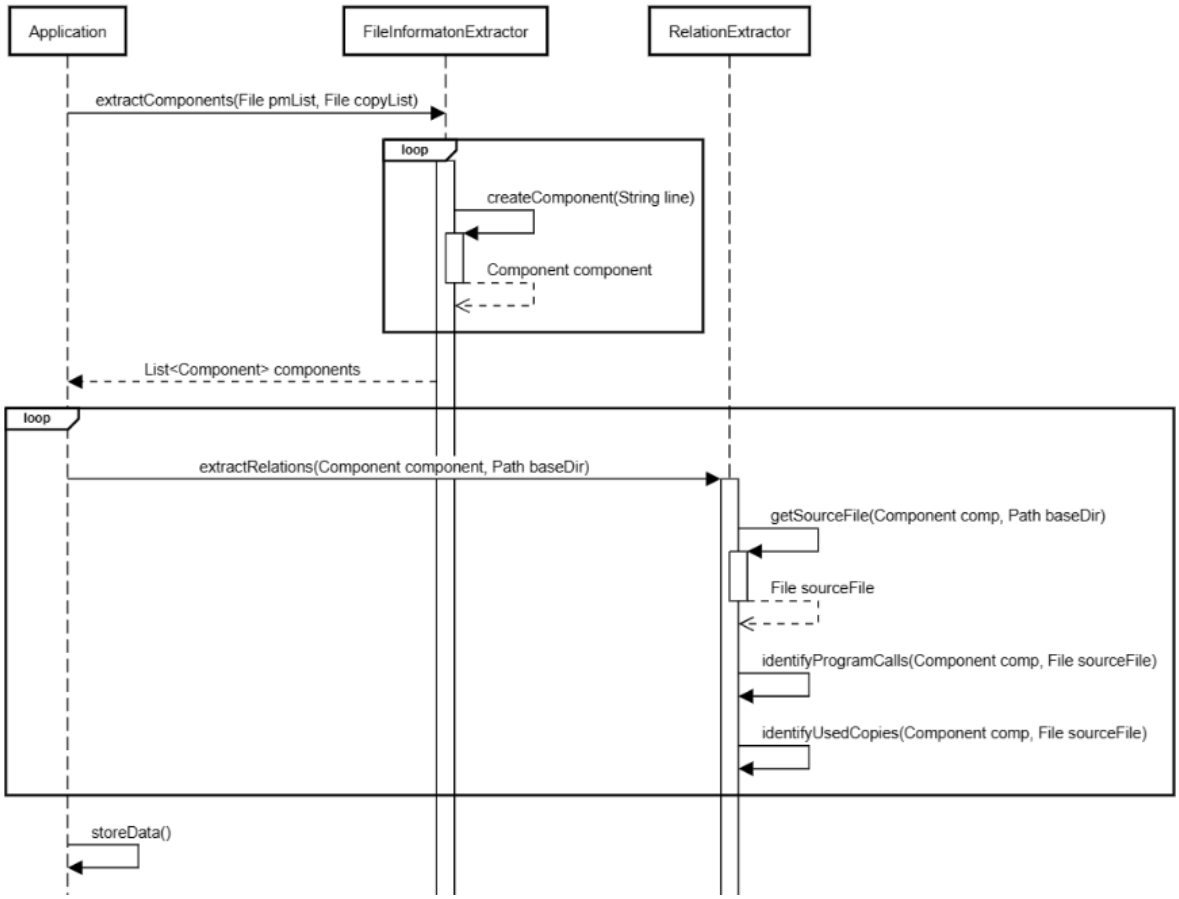

Abbildung 3.4: Sequenzdiagramm zur Erstellung der Component-Instanzen

4 Analysemöglichkeiten

Um die Struktur eines Software-Systems zu analysieren, gibt es verschiedene Verfahren. In diesem Kapitel werden drei Verfahren beschrieben, die potenziell für die Analyse des zu betrachtenden Software-Systems, anhand der in Kapitel 3 beschriebenen Daten, geeignet sind. Abschnitt 4.1 beginnt mit einer allgemeinen Beschreibung genetischer Algorithmen. Anschließend wird der konkrete genetische Algorithmus nach Sahraoui et al. (2002) beschrieben. In Abschnitt 4.2 werden die Begriffe *Netzwerk* und *Clustering* eingeführt und die Unterverfahren dargestellt, in die sich Clustering-Verfahren einteilen lassen. Anschließend wird die Netzwerkanalyse durch Clustering nach Girvan und Newman (2002) beschrieben. In Abschnitt 4.3 wird grobes Clustering als Annäherung für herkömmliches Clustering eingeführt und das grobe Clustering eines *Artifact Dependency Graph* nach Jahnke (2004) beschrieben. Die drei Verfahren werden dann in Kapitel 5 diskutiert.

4.1 Genetische Algorithmen

Genetische Algorithmen stellen eine Möglichkeit zur Gruppierung von Code dar. Sie sind von den evolutionären Mechanismen der Biologie abgeleitet. Die Durchführung eines genetischen Algorithmus beginnt mit einer Menge von Lösungen, der initialen Population. Ziel ist es, neue und bessere Lösungen zu erhalten. Die Eignung einer Lösung wird über die Fitness-Funktion bestimmt. Die Lösungen werden auch als Chromosomen bezeichnet. Pro Iteration eines genetischen Algorithmus werden Chromosomenpaare ausgewählt. Die Chromosomen werden dazu entsprechend ihrer Eignung priorisiert. Auf die Paare werden mit einer gewissen Wahrscheinlichkeit die Operatoren *Crossover* und *Mutation* angewendet. Die entstehenden Chromosomen bilden die neue Population. Die Größe dieser Population ist identisch mit der Größe der initialen Population. Die pro Iteration entstehenden Populationen werden auch als Generationen bezeichnet. Der Algorithmus endet mit Erreichen des Abbruchkriteriums. Ein Abbruchkriterium kann zum Beispiel die Anzahl der durchzuführenden Iterationen sein. (Sahraoui et al., 2002)

Zur folgenden Beschreibung des konkreten genetischen Algorithmus wird sich an Sahraoui et al. (2002) orientiert. Eine Lösung bzw. ein Chromosom ist als Menge von Gruppen festgelegt. Jede Gruppe wird als Gen bezeichnet und beinhaltet Code-Bestandteile. Als gültig wird eine Lösung

betrachtet, wenn die Vereinigung der in ihr enthaltenen Gruppen alle Code-Bestandteile genau einmal beinhaltet. Eine solche Lösung ist in Abbildung 4.1 zu sehen. Die Code-Bestandteile sind als Buchstaben, die Gruppen über die Abtrennungen dargestellt.

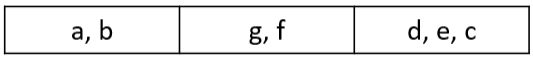

Abbildung 4.1: Beispiel einer gültigen Lösung bzw. eines gültigen Chromosoms nach Sahraoui et al. (2002)

Der Crossover-Operator ist so gewählt, dass jede Generation ausschließlich aus gültigen Lösungen besteht. Wie in Abbildung 4.2 zu sehen, werden die zwei zu kreuzenden Chromosomen P_1 und P_2 jeweils in zwei Teile geteilt. In einem ersten Schritt entsteht $C_1{}^*$, indem der rechte Teil von P_1 zwischen den beiden Teilen von P_2 eingesetzt wird. $C_2{}^*$ wird analog aus P_1 und dem rechten Teil von P_2 zusammengesetzt.

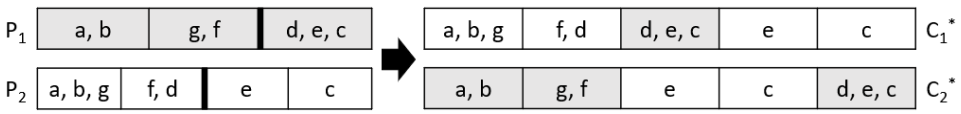

Abbildung 4.2: Erster Schritt der beispielhaften Anwendung des Crossover-Operators nach Sahraoui et al. (2002)

Bei $C_1{}^*$ und $C_2{}^*$ handelt es sich nicht um gültige Lösungen. Daher werden alle Code-Bestandteile aus den ursprünglichen Gruppen gelöscht, die in den eingefügten Gruppen enthalten sind. Dies ist in Abbildung 4.3 dargestellt. Die Chromosomen C_1 und C_2 sind sind das Resultat des Crossover-Operators.

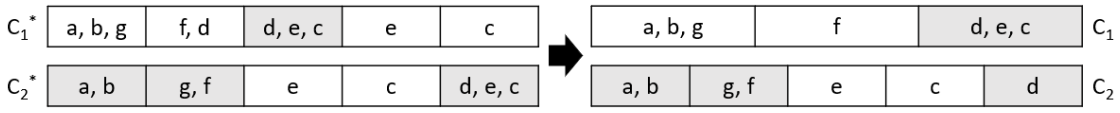

Abbildung 4.3: Zweiter Schritt der beispielhaften Anwendung des Crossover-Operators nach Sahraoui et al. (2002)

Auch der Mutation-Operator ist so festgelegt, dass eine Lösung, auf die er angewendet wird, weiterhin gültig ist. Es wird zufällig entschieden, ob zwei Gene des betroffenen Chromosoms zu einem zusammengeführt werden oder ein Gen in zwei aufgeteilt wird. So kann ein Chromosom P beispielsweise, wie in Abbildung 4.4 zu sehen, zu Pm_1 oder Pm_2 mutieren.

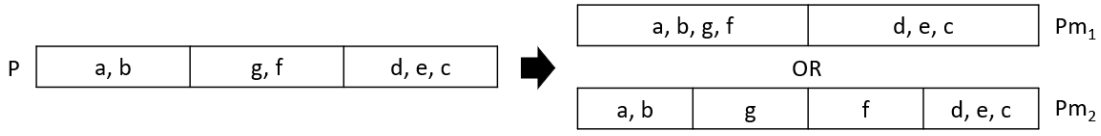

Abbildung 4.4: Beispielhafte Anwendung des Mutation-Operators nach Sahraoui et al. (2002)

Die Fitness eines Chromosoms bzw. einer Lösung S wird als der Durchschnitt der Fitness f der in S enthaltenen Gene bzw. Gruppen definiert:

$$F(S) = \frac{\sum_{i=1}^{N} f(G_i)}{N};\ N > 0$$

N ist die Anzahl der in S enthaltenen Gruppen G_i.

Die Fitness einer Gruppe G_i wird über ihre Kohäsion, $COH(G_i)$, und ihre Kopplung zu den anderen Gruppen, $COUP(G_i)$, bestimmt. Hierfür wird ein Schwellenwert MIN_COH für den minimal akzeptablen Kohäsionswert festgelegt. Die Funktion f ist so definiert, dass für zwei Objekte G_1 und G_2 gilt:

- $(COH(G_1) < MIN\_COH \wedge COH(G_2) < MIN\_COH) \wedge (COUP(G_1) > COUP(G_2)) \implies f(G_1) > f(G_2)$
- $(COH(G_1) \geq MIN\_COH \wedge COH(G_2) < MIN\_COH) \implies f(G_1) > f(G_2)$
- $(COH(G_1) \geq MIN\_COH \wedge COH(G_2) \geq MIN\_COH) \wedge (COUP(G_1) > COUP(G_2)) \implies f(G_1) > f(G_2)$

Die Funktion f, welche die Fitness einer Gruppe bestimmt, sieht wie folgt aus:

$$f(G_i) = \begin{cases} f_1(G_i) = \frac{1}{COUP(G_i)+2} + \frac{1}{2} \text{ falls } COH(G_i) \geq MIN\_COH \\ f_2(G_i) = \frac{1}{4(COUP(G_i)+1)} \text{ falls } COH(G_i) < MIN\_COH \end{cases}$$

Diese Funktion erfüllt die oben aufgeführten Anforderungen, denn die Lösung mit höherer Kohäsion wird systematisch bevorzugt. So gilt: $f_1 \in [\frac{1}{2}, 1]$ und $f_2 \in [0, \frac{1}{4}]$. Zudem ist die Fitness einer Gruppe umso niedriger, je höher ihre Kopplung ist.

Sahraoui et al. (2002) verfolgen das Ziel, ein Programm zu gruppieren, welches aus Variablen und Routinen, die diese Variablen verwenden, besteht. Die in den Gruppen eines Chromosoms enthaltenen Code-Bestandteile sind deshalb die Variablen des Programms. $COH(G_i)$ und $COUP(G_i)$ sind über die Beziehungen zwischen den Variablen und Routinen eines Programms und den Routinen untereinander definiert. Durch die Verwendung anderer Code-Bestandteile als Variablen und die Anpassung der Funktionen $COH(G_i)$ und $COUP(G_i)$ auf diese Bestandteile sollte es möglich sein, den Algorithmus auf andere Anwendungsfälle zu übertragen.

4.2 Netzwerkanalyse durch Clustering

Ein Datensatz, bestehend aus Knoten, über die relationale Daten zur Verfügung stehen, ist ein Netzwerk. Ein Netzwerk wird in der Regel als Graph $G(N, L)$ dargestellt, der aus einer Menge von Knoten N und einer Menge von Kanten L besteht. Die Kanten verbinden die Knoten. Ein Bereich der statistischen Netzwerkanalyse ist es, wichtige Akteure, dargestellt als Knoten, zu bestimmen. Dafür werden Zentralitätsmaße verwendet. Hierzu gehört die Betweenness-Zentralität, bei welcher der Einfluss eines Knotens, über seine Rolle bei der

Verknüpfung anderer Knoten, bestimmt wird. Dies wird mithilfe der kürzesten Pfade zwischen Knoten-Paaren, die durch den Knoten laufen, berechnet. Ein weiterer Aufgabenbereich der Nezwerkanalyse ist das Clustering. (Salter-Townshend, White, Gollini & Murphy, 2012)

Als Clustering wird der Prozess bezeichnet, Objekte zu gruppieren bzw. in Cluster aufzuteilen. Die Objekte innerhalb eines Clusters weisen eine hohe Ähnlichkeit zueinander und eine niedrige Ähnlichkeit zu Objekten anderer Cluster auf. Es gibt verschiedene Clustering-Methoden. Zu den wichtigsten gehören die hierarchischen Methoden. Sie lassen sich wiederum in agglomerative und divisive hierarchische Clustering-Methoden unterteilen. Agglomerative Methoden verfolgen eine Bottom-Up-Strategie. Jedes Objekt wird zu Beginn einem eigenen Cluster zugeordnet. Die Cluster werden dann zu immer größeren Clustern vereinigt, bis ein Cluster alle Objekte indirekt enthält oder eine andere festgelegte Abbruchbedingung erreicht wird. Divisive Methoden verfolgen eine Top-Down-Strategie. Alle Objekte befinden sich zu Beginn in einem Cluster. Dieses wird immer weiter unterteilt, bis jedes Objekt einem eigenen Cluster zugeordnet ist oder eine andere festgelegte Abbruchbedingung erreicht wird.

Neben hierarchischen Methoden gibt es auch partitionierende, gitter-, dichte- und modellbasierte Methoden. Partitionierende Clustering-Methoden teilen die Objekte in eine vorgegebene Anzahl an Gruppen auf. Anschließend werden die Objekte zwischen den Gruppen verschoben, um die Aufteilung zu verbessern. Bei gitterbasierten Methoden erfolgt eine Unterteilung des Objektraums in endlich viele Zellen. Die Zellen bilden eine Gitterstruktur, auf der das Clustering durchgeführt wird. Dichtebasierte Clustering-Methoden bilden die Cluster nach der Objektdichte im Objektraum. Mit modellbasierten Methoden wird ein Modell für jedes der Cluster angenommen, dem die Objekte dann bestmöglich passend zugeordnet werden. Zudem gibt es hybride Clustering-Methoden, die verschiedene Methoden kombinieren. Clustering kann automatisch oder semi-automatisch ausgeführt werden. Die Durchführung eines semi-automatischen Clustering-Prozesses erfolgt mit Unterstützung der Nutzer. (Asif, Shahzad, Saher & Nazar, 2009)

Clustering im Bereich der Netzwerkanalyse strebt eine Partitionierung des Netzes in stark verbundene Teilgraphen an (Salter-Townshend et al., 2012). Viele Netzwerke weisen eine Gemeinschaftsstruktur (community structure) auf, anhand der das Clustering durchgeführt werden kann. In einem Netzwerk mit einer Gemeinschaftsstruktur lassen sich die Knoten in Teilmengen einteilen, die als Gemeinschaften bezeichnet werden. Innerhalb dieser Teilmengen ist die Dichte an Verbindungen zwischen den Knoten höher als zwischen den Teilmengen. Ein Netzwerk mit Gemeinschaftsstruktur ist in Abbildung 4.5 dargestellt. Die Kreise repräsentieren die Knoten, die dunklen Linien stellen die Verbindungen zwischen den Knoten innerhalb einer Gemeinschaft dar und die hellen Linien sind die Verbindungen zwischen Knoten verschiedener Gemeinschaften. Eine Gemeinschaft kann wiederum eine Gemeinschaftsstruktur aufweisen,

sodass sich eine hierarchische Struktur ergibt. (Girvan & Newman, 2002)

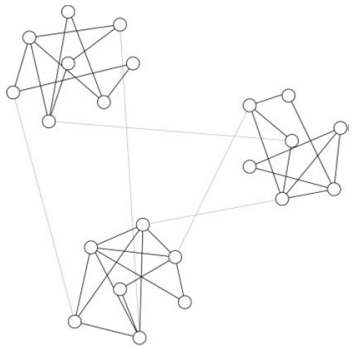

Abbildung 4.5: Darstellung eines Netzwerkes mit einer Gemeinschaftsstruktur (Girvan & Newman, 2002)

Zur Bestimmung der Gemeinschaften definieren Girvan und Newman (2002) eine Kanten-Betweenness (edge betweenness) analog zur Betweenness-Zentralität eines Knotens. Die Kanten-Betweenness ist definiert als die Anzahl der kürzesten Wege zwischen Knotenpaaren, die entlang der Kante führen. Existieren mehrere kürzeste Wege zwischen einem Knotenpaar, werden die Wege gleich gewichtet, sodass die Summe der Gewichte 1 ergibt. Enthält ein Netzwerk Gemeinschaften, die nur durch wenige Kanten verbunden sind, verlaufen alle kürzesten Pfade zwischen den Gemeinschaften entlang dieser wenigen Kanten. Somit hat mindestens eine Kante zwischen zwei Gemeinschaften eine hohe Kanten-Betweenness. Durch die Entfernung dieser Kanten werden die Gemeinschaften voneinander getrennt und somit identifiziert.

Der durchzuführende Algorithmus beginnt mit der Berechnung der Kanten-Betweenness jeder Kante des Netzwerkes. Anschließend wird die Kante mit der höchsten Kanten-Betweenness entfernt. Die Schritte werden dann so lange wiederholt, bis keine Kanten übrig sind. Nach der Entfernung der ersten Kante können die Betweenness-Werte der Kanten, die nicht davon betroffen sind, wiederverwendet werden. Das Ergebnis lässt sich, wie in Abbildung 4.6 zu sehen, als hierarchischer Baum darstellen. Die Kreise stellen die Knoten dar, die Teilbäume repräsentieren die Gemeinschaften und die Baumstruktur zeigt die hierarchische Anordnung der Gemeinschaften. (Girvan & Newman, 2002)

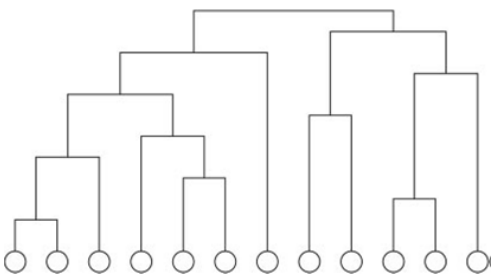

Abbildung 4.6: Beispiel eines hierarchischen Baumes (Girvan & Newman, 2002)

4.3 Grobes Clustering eines Artifact Dependeny Graph

In diesem Absatz wird sich an Jahnke (2004) orientiert und sein Ansatz vorgestellt. Ein *Artifact Dependency Graph* (ADG) wird verwendet, um die zu gruppierenden Software-Artefakte und die zwischen ihnen bestehenden Abhängigkeiten darzustellen. Der Graph besteht aus einer endlichen Menge an Software-Artefakten A und einer endlichen Multimenge R, die die Abhängigkeiten zwischen den Artefakten darstellt. Ein Element aus R hat die Form $\{a, b\}$, mit $a, b \in A$.

Das angestrebte Ergebnis eines Clustering-Prozesses ist die Einteilung von A in nichtleere Teilmengen P_i, mit $i = 1, 2, .., n$. Dabei gilt, dass jedes Element von A in der Mengenfamilie $P = \{P_1, P_2, ..., P_3\}$ genau einmal vorliegt. Es existieren jedoch Software-Elemente, die zu mehr als einer Teilmenge P_i passen. Als Approximation für P wird daher $\underline{P} = \{\underline{P}_1, \underline{P}_2, ..., \underline{P}_n\}$ verwendet, eine Familie von groben bzw. ungenauen Mengen. Eine grobe Menge $\underline{S}$ ist als $\underline{S} := (S_{\approx},\ S^{\approx})$ definiert und nähert die Menge S an. Die Menge $S_{\approx}$ wird *untere Approximation* genannt und enthält nur die Elemente, für die feststeht, dass sie zu S gehören. Die Menge $S^{\approx}$ wird als *obere Approximation* bezeichnet und enthält alle Elemente, die potenziell zu S gehören. Es gilt $S_{\approx} \subseteq S^{\approx}$.

Für die Einteilung der Elemente von A in die groben, in $\underline{P}$ enthaltenden Teilmengen gilt:

$$\forall a \in A\ \exists \underline{P}_i : ((a \in P_{i\approx} \wedge \nexists \underline{P}_k : (\underline{P}_k \neq \underline{P}_i \wedge a \in \underline{P}_k)) \vee (a \in P_i^{\approx} \wedge \nexists \underline{P}_k : a \in P_{k\approx}))$$

Jedes Element von A ist entweder in genau einer unteren Approximation eines Elements aus $\underline{P}$ enthalten oder in der oberen Approximation von mindestens einem Element aus $\underline{P}$. Zudem enthält jede grobe Teilmenge mindestens ein Samen-Artefakt, also ein Artefakt, dass nach Ansicht eines menschlichen Experten eindeutig ihrer unteren Approximation zugeordnet werden kann. In Abbildung 4.7 ist eine Einteilung der Software-Artefakte eines ADGs in drei grobe Teilmengen bzw. Cluster zu sehen. Die Kreise stellen Artefakte dar und sind fettgedruckt, falls es sich um Samen-Artefakte handelt. Abhängigkeiten zwischen Artefakten werden als Linien dargestellt. Die Ovale bilden die unteren Approximationen der Cluster ab. Jedes Oval ist in einer Form eingeschlossen, welche die zugehörige obere Approximation repräsentiert.

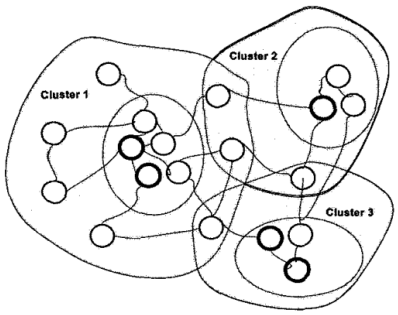

Abbildung 4.7: Beispielhafter, in grobe Cluster unterteilter ADG (Jahnke, 2004)

Zur Bewertung der Qualität der Einteilung in grobe Cluster werden drei Metriken verwendet. Die erste Metrik gibt an, wie konkret die Einteilung ist. Die Konkretheit der Einteilung ist dabei umso größer, je weniger Artefakte sich in Grenzregionen von Clustern befinden. Für ein grobes Cluster $\underline{S} := (S_{\approx},\ S^{\approx})$ ist die Grenzregion als $\square(\underline{S}) = S^{\approx} \setminus S_{\approx}$ definiert. Die Konkretheit $\Diamond^*$ der Menge der gewählten Cluster $\underline{P} = \{\underline{P}_1, \underline{P}_2, ..., \underline{P}_n\}$ wird über die Konkretheit $\Diamond$ der einzelnen Cluster $\underline{P}_i$ definiert:

$$\Diamond^*(\underline{P}) := \frac{\sum\limits_{\underline{P}_i \in \underline{P}} \Diamond(\underline{P}_i)}{|\underline{P}|} \text{ mit } \Diamond(\underline{P}_i) := \frac{|\underline{P}_{i\approx}|}{|\underline{P}_i^{\approx}|}$$

Als zweite Metrik wird die Mehrdeutigkeit Θ verwendet. Sie wird darüber definiert, wie viele Artefakte $a \in A$ in den Grenzregionen mehrerer Cluster vorliegen, also nicht eindeutig einem Cluster zugeordnet sind:

$$\Theta(\underline{P}) := \frac{|\{\underline{P}_i, \underline{P}_k, a \mid (\underline{P}_i, \underline{P}_k \in \underline{P}) \wedge (\underline{P}_i \neq \underline{P}_k) \wedge (a \in \underline{P}_i) \wedge (a \in \underline{P}_k)\}|}{2|A| \cdot |\underline{P}|^2}$$

Als dritte Metrik wird die Unvollständigkeit Ξ verwendet. Sie betrachtet die Anzahl der Artefakte, die einer einzigen Grenzregion zugeordnet sind. Die Definition der Unvollständigkeit lautet:

$$\Xi(\underline{P}) := \frac{|\{a \mid \neg(\exists \underline{P}_i, \underline{P}_k \in \underline{P})(\underline{P}_i \neq \underline{P}_k \wedge a \in \square(\underline{P}_i) \wedge a \in \square(\underline{P}_k))\}|}{|A|}$$

Bei der Überführung der Menge der Software-Artefakte A in eine Familie grober Teilmengen bzw. Cluster $\underline{P}$ soll das Wissen verfügbarer menschlicher Experten genutzt werden. In dem Clustering-Algorithmus ist daher Nutzer-Interaktion eingeschlossen. Zu Beginn des Algorithmus definiert der Nutzer ein *business-concept model* (BMC), welches die Domäne beschreibt, die von dem zu gruppierenden Software-System implementiert wird. Zur weiteren Eingrenzung werden den einzelnen Konzepten des BCM Stichwörter zugeordnet. Die Konzepte werden als Kandidaten für grobe Cluster verwendet und die Stichwörter mit den im Software-System verwendeten Idiomen verglichen, um den Cluster-Kandidaten Software-Artefakte zuzuordnen. Die Cluster-Kandidaten, denen keine Artefakte exklusiv zugeteilt sind, werden nicht als grobe Cluster verwendet. Die exklusiv zugeordneten Artefakte sind die Samen-Artefakte des Clusters.

Der folgende Schritt wird als impedanzbasiertes Clustering bezeichnet. Dabei werden die noch nicht exklusiv zugeordneten Artefakte den groben Clustern zugeordnet. Die Cluster-Impendanz $\Omega(\underline{P}_i, a)$ zwischen einem Artefakt a und einem Cluster $\underline{P}_i$ ist dabei so definiert, dass sie kleiner ist, je höher der Grad der Zugehörigkeit ist bzw. je enger die Kopplung zwischen den Samen-Artefakten $X(\underline{P}_i)$ und dem Artefakt a ist:

$$\Omega(\underline{P}_i, a) = \begin{cases} 0 & \text{falls } a \in X(\underline{P}_i) \\ \dfrac{1}{\displaystyle\sum_{s \in X(\underline{P}_i)} \dfrac{1}{\omega^*(s,a)}} & \text{sonst} \end{cases}$$

ω^* ist die Impedanz zwischen zwei Artefakten. Sie ist kleiner, je enger die Kopplung zwischen den beiden Artefakten ist, und wird bestimmt, indem die im AGM vorzufinden Abhängigkeiten nach ihrem Typ gewichtet werden. Software-Artefakte, die eine gemeinsame Programmvariable verwenden, werden beispielsweise als enger gekoppelt betrachtet, als solche, die über einen Funktionsaufruf in Beziehung stehen. Sie bekommen daher ein niedrigeres Gewicht zugeordnet. ω^* berücksichtigt, wenn zwei Artefakte über mehrere Relationen direkt verbunden sind oder sie keine direkte Relation vorweisen können, aber transitive Relationen über andere Artefakte vorliegen.

Beim impedanzbasierten Clustering kann der Nutzer zwei Schwellenwerte LT und HT festlegen. Diese Schwellenwerte werden für jedes Artefakt a und jedes Cluster $\underline{P}_i$ mit der Cluster-Impedanz verglichen. Ist $\Omega(\underline{P}_i, a)$ kleiner als LT und gilt dies für a bei keinem anderen Cluster, so wird a der unteren Approximation von $\underline{P}_i$ zugeordnet. Ist $\Omega(\underline{P}_i, a)$ größer als HT, wird a dem Cluster $\underline{P}_i$ nicht zugeteilt. Ansonsten wird a der Grenzregion von $\underline{P}_i$ zugewiesen:

- $a \in \underline{P}_{i\approx}$ falls $(\Omega(\underline{P}_i, a) < LT) \wedge (\neg\exists \underline{P}_k \in P : (\Omega(\underline{P}_k, a) < LT \wedge \underline{P}_i \neq \underline{P}_k))$
- $a \notin \underline{P}_i$ falls $\Omega(\underline{P}_i, a) > HT$
- $a \in \underline{P}_i^{\approx}$ sonst

In einem nächsten Schritt werden die drei Qualitäts-Metriken Konkretheit $\Diamond^*$, Mehrdeutigkeit Θ und Unvollständigkeit Ξ berechnet und dem Nutzer angezeigt. Der Nutzer kann entscheiden, ob die Zuordnung der ADG-Elemente zum BCM zufriedenstellend ist. Ist dies nicht der Fall, können die Schritte wiederholt werden, wobei das BCM und die Zuordnung der Samen-Artefakte verändert werden können.

Ist der Nutzer mit der Zuordnung zufrieden, kann die Cluster-Einteilung weiter verfeinert werden. Hierfür kann der Nutzer Artefakte von den Grenzregionen der Cluster zu ihren unteren Approximationen verschieben, die Artefakte also eindeutig zuordnen. Zudem können neue Cluster erstellt und ihnen Samen-Artefakte zugeordnet werden. Die Schwellenwerte LT und HT können angepasst werden, die den verschiedenen Abhängigkeitstypen zugeordneten Gewichte ebenfalls. Auch diese Schritte können wiederholt werden, bis der Nutzer mit der sich ergebenen Partition zufrieden ist. Dies kann beispielsweise dann der Fall sein, wenn alle Artefakte eindeutig einem Cluster zugeordnet sind.

5 Diskussion der Analysemöglichkeiten

In Kapitel 4 wurden drei Verfahren zur Strukturanalyse eines Software-Systems beschrieben. Diese Verfahren und ihre Eignung für das zu untersuchende Legacy-System werden in diesem Kapitel diskutiert. Die Analyse des Legacy-Systems ist anhand der zwischen Programmsourcen und Copy-Strecken bestehenden Relationen durchzuführen, welche in Kapitel 3 beschrieben sind. Die Strukturanalyse soll zum Verständnis des Systems beitragen und zu einer leichteren Nachvollziehbarkeit der Abhängigkeiten führen. Zudem soll sie eine Grundlage für zukünftige Refaktorisierungs-Arbeiten bilden, beispielsweise um das System in die in Abschnitt 2.3 beschriebenen Microservices zu überführen.

An dem genetischen Algorithmus nach Sahraoui et al. (2002), beschrieben in Abschnitt 4.1, und der Netzwerkanalyse durch Clustering nach Girvan und Newman (2002), beschrieben in Abschnitt 4.2, wurden unterschiedliche Anzahlen von Tests durchgeführt. Auch die für die Tests verwendeten Daten unterscheiden sich. Der genetische Algorithmus wurde an drei realen Software-Systemen durchgeführt, um die Variablen und Routinen der Programme zu gruppieren. Um die Netzwerkanalyse durch Clustering zu testen, wurden computergenerierte Graphen mit bekannter Gemeinschaftsstruktur, zwei reale Netze mit gut dokumentierter Struktur und zwei reale Netzwerke mit Strukturen, die nicht gut dokumentiert waren, verwendet. Die ausgewählten realen Netzwerke umfassten zum Beispiel ein Freundschafts-Netzwerk aus der bekannten Karate-Club-Studie von Zachary (1977). Für das in Abschnitt 4.3 beschriebene grobe Clustering eines *Artifact Dependency Graph* (ADG) nach Jahnke (2004) stand eine Evaluierung zum Zeitpunkt der Publikation noch aus. Ein Vergleich der drei Verfahren erweist sich daher als schwierig.

Der genetische Algorithmus erzielte für zwei der Systeme, die zum Testen verwendet wurden, ein gutes Ergebnis. Die beiden Systeme waren gut implementiert. Das Resultat für das dritte System erwies sich als weniger gut. Dies könnte auf die geringe Modularität des Systems zurückzuführen sein. Zudem kann keine Aussage über die Skalierbarkeit für größere Systeme getroffen werden, da die Auswahl der getesteten Systeme dies nicht erlaubt. (Sahraoui et al., 2002) Für das hier zu untersuchende System kann nicht von einer hohen Modularität ausgegangen werden. Aus diesem Grund und angesichts der Größe des Systems muss damit gerechnet werden, dass das Resultat der Methode für dieses System nicht zufriedenstellend ist.

Die Netzwerkanalyse durch Clustering erzielte für die computergenerierten Graphen mit bekannter Gemeinschaftsstruktur und für die realen Netze mit gut dokumentierter Struktur sehr gute Resultate. Auch bei der Anwendung auf die Netzwerke, deren Strukturen nicht gut dokumentiert waren, wurden die Ergebnisse als plausibel und informativ wahrgenommen. Zur Berechnung der Kanten-Betweenness wurde ein Algorithmus mit der Laufzeit $\mathcal{O}(mn)$ verwendet, wobei m die Anzahl der Kanten des Graphen ist und n die Anzahl der Knoten. Für die gesamte Methode ergibt sich im ungünstigsten Fall eine Laufzeit von $\mathcal{O}(m^2n)$, sodass es bei sehr großen Graphen zu Laufzeitproblemen kommen kann. (Girvan & Newman, 2002) Aufgrund der Größe des zu untersuchenden Systems ist bei der Verwendung dieser Methode mit Laufzeitproblemen zu rechnen.

Das grobe Clustering eines ADGs erlaubt es möglicherweise, eine realitätsnähere Partitionierung zu generieren als andere Verfahren, da auch die Darstellung nicht eindeutiger Zuordnungen über die groben Cluster möglich ist. Die Code-Artefakte, die nicht eindeutig zugeordnet sind, bieten Ansatzpunkte für Refaktorisierungsmaßnahmen, welche zu schärferen Cluster-Grenzen führen. Der Algorithmus ist semi-automatisch und bietet den Vorteil, Expertenwissen einbinden zu können und so potenziell ein, aus menschlicher Sicht, aussagekräftiges Ergebnis zu generieren. Dies wird durch den iterativen und inkrementellen Ansatz unterstützt, da der Nutzer das Ergebnis durch eine Anpassung der Parameter immer weiter verbessern kann. Für die Experten bedeutet die Nutzerinteraktion zusätzlichen Arbeitsaufwand und für das Unternehmen Mehrkosten. Durch den menschlichen Einfluss ist das Verfahren zudem langsamer als vergleichbare automatische Algorithmen. Aufgrund der Größe des zu untersuchenden Systems ist mit einem hohen Kosten- und Zeitaufwand zu rechnen. Außerdem stehen nicht für alle Teilsysteme Personen mit ausreichender Expertise zur Verfügung und es ist fraglich, ob Nutzerinteraktion das beste Resultat hervorbringt. Das Wissen über bestehende Strukturen könnte eine objektiv bestgeeignete Partitionierung verhindern. Auch die Zuordnung der Software-Artefakte zu den Cluster-Kandidaten, anhand der im Software-System verwendeten Idiome, könnte problematisch sein, da Bestandteile von Legacy Code oft nicht angemessen benannt sind (Fuhr et al., 2011).

Für alle drei Methoden sind Adaptionen erforderlich, damit sie für das zu untersuchende System verwendet werden können. Beispielsweise müssen die im genetischen Algorithmus verwendeten Funktionen zur Berechnung der Kohäsion und der Kopplung angepasst werden. Da Programmaufrufe und das Einbinden von Copy-Strecken eine Richtung aufweisen, muss die Clustering-Methode zur Analyse eines Netzwerkes adaptiert werden, sodass sie gerichtete Graphen unterstützt. Das grobe Clustering eines ADGs erfordert die Festlegung der benötigten Schwellenwerte und der Gewichte, die den Relationen zugeordneten werden.

Aufgrund der genannten Nachteile, die aus einer Nutzerinteraktion resultieren, erscheint das grobe Clustering eines ADGs am wenigsten geeignet zu sein. Auch wenn die anderen Methoden

nicht die Möglichkeit bieten, Software-Artefakte explizit mehreren Clustern zuordnen zu können, sollte es dennoch möglich sein, diese Artefakte als schwer zuordenbar zu identifizieren, beispielsweise über einen Vergleich der Relationen mit Artefakten im eigenen Cluster zu den Relationen mit Artefakten anderer Cluster. Nach den Testergebnissen verspricht die Netzwerkanalyse durch Clustering das beste Resultat. Wie bei den anderen Methoden auch, ist damit zu rechnen, dass weitere Anpassungen für ein System dieser Größe notwendig sind.

6 Zusammenfassung und Ausblick

In dieser Arbeit wurden die für eine Analyse zur Verfügung stehenden Daten eines Legacy-Systems betrachtet. Anschließend wurden der genetische Algorithmus nach Sahraoui et al. (2002), die Netzwerkanalyse durch Clustering nach Girvan und Newman (2002) und das grobe Clustering eines *Artifact Dependeny Graph* nach Jahnke (2004) vorgestellt und diskutiert. Von ihnen verspricht die Netzwerkanalyse durch Clustering bei einer Anwendung auf das betrachtete Software-System das beste Resultat. Um diese Behauptung bestätigen oder widerlegen zu können, sind weitere Untersuchungen notwendig, da die Methoden unter verschiedenen Bedingungen und weder an dem betrachteten Software-System noch an vergleichbaren Datensätzen getestet wurden.

In einem nächsten Schritt kann die Methode von Girvan und Newman (2002) an das zu untersuchende System angepasst und anschließend implementiert werden. Nach der Vervollständigung der in Kapitel 3 beschriebenen Extraktion der Daten für ein Teilsystem kann die Methode auf dieses Teilsystem angewendet werden. Die Anwendung der Verfahren zur Datenextraktion und zur Analyse auf das Gesamtsystem kann erfolgen, nachdem notwendige Änderungen identifiziert und durchgeführt wurden. Die Ergebnisse der Analyse können dann zur Refaktorisierung des Legacy-Systems verwendet werden, beispielsweise um eine Microservice-Architektur zu erhalten.

Literaturverzeichnis

Asif, N., Shahzad, F., Saher, N. & Nazar, W. (2009, 12). Clustering the source code. *WSEAS Transactions on Computers*, *8*, 1835-1844.

Barbier, F. & Recoussine, J.-L. (2015). *COBOL Software Modernization: From Principles to Implementation with the BLU AGE Method.* John Wiley & Sons.

Feathers, M. C. (2020). *Effektives Arbeiten mit Legacy Code: Refactoring und Testen bestehender Software.* BoD–Books on Demand.

Fuhr, A., Horn, T. & Riediger, V. (2011). Using dynamic analysis and clustering for implementing services by reusing legacy code. In *2011 18th Working Conference on Reverse Engineering* (S. 275–279).

Girvan, M. & Newman, M. E. (2002). Community structure in social and biological networks. *Proceedings of the national academy of sciences*, *99* (12), 7821–7826.

Gos, K. & Zabierowski, W. (2020). The comparison of microservice and monolithic architecture. In *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)* (S. 150–153).

Hambeck, K. (2019). Einführung in das Programmieren in COBOL. In *Einführung in das Programmieren in COBOL.* de Gruyter.

Holzem, T. (2022a). *CCM Allgemeines zu Bauteilgruppen.* (Interne Dokumentation, Zugriff am 07.12.2022)

Holzem, T. (2022b). *CCM Metadaten (MMR) Vokabular.* (Interne Dokumentation, Zugriff am 07.12.2022)

Holzem, T. (2022c). *CCM MMR (Method Manager).* (Interne Dokumentation, Zugriff am 07.12.2022)

Holzem, T. (2022d). *CCM Namenskonventionen für Software-Elemente und Metadaten.* (Interne Dokumentation, Zugriff am 07.12.2022)

Holzem, T. (2022e). *CCM Typen von Metadaten-Membern und ihre Attribute.* (Interne Dokumentation, Zugriff am 07.12.2022)

IBM. (2021). *CALL statement.* Zugriff am 27.10.2022 auf `https://www.ibm.com/docs/sl/developer-for-zos/9.5.1?topic=statements-call-statement`

IBM. (2022). *COPY statement.* Zugriff am 27.10.2022 auf `https://www.ibm.com/docs/en/cobol-zos/6.2?topic=statements-copy-statement`

Jahnke, J. H. (2004). Reverse engineering software architecture using rough clusters. In *IEEE Annual Meeting of the Fuzzy Information, 2004. Processing NAFIPS'04.* (Bd. 1, S. 4–9).

Kirchmayr, W., Moser, M., Nocke, L., Pichler, J. & Tober, R. (2016). Integration of Static and Dynamic Code Analysis for Understanding Legacy Source Code. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (S. 543-552). doi: 10.1109/ICSME.2016.70

NASCIO. (2008). *Digital States at Risk! Modernizing Legacy Systems, survey.*

Newman, S. (2015). *Microservices: Konzeption und Design.* MITP-Verlags GmbH & Co. KG.

Newman, S. (2020). *Vom Monolithen zu Microservices: Patterns, um bestehende Systeme Schritt für Schritt umzugestalten.* O'Reilly.

Richards, M. & Ford, N. (2020). *Handbuch moderner Softwarearchitektur: Architekturstile, Patterns und Best Practices.* O'Reilly.

Sahraoui, H., Valtchev, P., Konkobo, I. & Shen, S. (2002). Object identification in legacy code as a grouping problem. In *Proceedings 26th Annual International Computer Software and Applications* (S. 689–696).

Salter-Townshend, M., White, A., Gollini, I. & Murphy, T. B. (2012). Review of statistical network analysis: models, algorithms, and software. *Statistical Analysis and Data Mining*, *5* (4), 243–264.

Schwickert, A. C. (2018). Strukturierte Programmierung in COBOL. In *Strukturierte Programmierung in COBOL.* Oldenbourg Wissenschaftsverlag.

Wolf, J. (2020). *Computergeschichte(n): Nicht nur für Nerds.* Rheinwerk Verlag.

Xiao, Z., Wijegunaratne, I. & Qiang, X. (2016). Reflections on SOA and Microservices. In *2016 4th International Conference on Enterprise Systems (ES)* (S. 60–67).

Zachary, W. W. (1977). An information flow model for conflict and fission in small groups. *Journal of anthropological research*, *33* (4), 452–473.