UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

Metrik-basiertes Refaktorieren von Klassen am Beispiel der Kohäsion

Metric-based refactoring of classes at the example of cohesion

Masterarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Tobias Gedler

ausgegeben und betreut von
Prof. Dr. Martin Leucker

mit Unterstützung von
Bernhard Werner

Die Masterarbeit ist im Rahmen einer Tätigkeit als Masterand
bei der Firma CODESYS GmbH entstanden.

Lübeck, den 28.02.2021

Erklärung

Ich erkläre hiermit an Eides statt, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

(Tobias Gedler)
Bodolz, den 28.02.2021

Kurzfassung In dieser Masterarbeit wird untersucht, inwiefern basierend auf der Kohäsion einer Klasse ein Refactoring möglich ist. Dazu wird untersucht, welche Metriken sich zur Berechnung der Kohäsion sich auf struktureller Ebene hierfür eignen. Basieren auf diesen Metriken und der Struktur der Klasse soll die Kohäsion für die gesamte Klasse visualisiert werden. Außerdem wird geprüft, welche Code-Smells identifiziert werden können. Aus diesen Code-Smells werden entsprechende Refactorings abgeleitet um die zugrundeliegenden Code-Smells zu beseitigen. Anschließend wird das Konzept als Plugin für CODESYS umgesetzt.

Abstract This master thesis investigates in what way a refactoring is possible based on the cohesion. For that it is investigated which metrics can be used for the calculation on a structural layer. Based on these metrics and the structure of the class, the cohesion of the whole class should be visualized. Moreover will be checked which code smells can be identified. Refactorings will be drived from these code smells to remove these code smells. Afterwards the concept will be implemnted as a plugin for CODESYS.

Inhaltsverzeichnis

1 Einleitung

McEnery geht in seinem Artikel über die Anzahl jährlich geschriebener Zeilen Code von über 92 Miliarden Zeilen aus - geschrieben von 6 Millionen Entwicklern [McE20]. Auch wenn es in der Realität nur zehn Prozent dieser Anzahl an Code-Zeilen sind, ist die Menge an Code die von einem Entwickler selber geschrieben wurde verschwindend gering im Vergleich zum Rest. Trotzdem muss auch fremder Code verstanden werden können. Hier kommen Metriken ins Spiel, die Helfen schlechten Code zu entdecken und mittels Refactoring verständlich zu machen und verständlich zu halten.

1.1 Verwandte Arbeiten

Wolverton unternahm 1974 mit der Lines of Code Metrik [Wol74] einen der ersten Versuche die Eigenschaften eines Programms zu messen. McCabe entwickelte 1976 die nach ihm benannte McCabe-Metrik oder zyklomatische Komplexität [McC76]. Eine der ersten Metrik-Suiten für objektorientierte Programmierung erschien 1991 [CK91]. Zwei der ersten wissenschaftlichen Arbeiten zum Thema Refactoring sind die Dissertationen von Griswold[Gri91] und Opdyke[Opd92]. Wobei Griswold Refactoring im Bereich prozeduraler Software betrachtete, während Opdyke Refactoring bei objekt-orientierte Frameworks untersuchte.
Bieman et al. [BO94] verwendeten einen auf dem Slice-basierten Ansatz um Glue-Tokens im Programm zu finden und somit die Kohäsion zu messen. Simon et al. [SLS01] verwenden eine Distanz-basierte Metrik um die Kohäsion zu messen und Stellen für ein Refactoring zu finden.

1.2 Zielsetzung dieser Arbeit

- Auswahl geeigneter Metriken für die Kohäsion
- Visualisierung der Kohäsion
- Identifizierung von Code-Smells
- Auswahl geeigneter Refactorings

- prototypische Umsetzung in CODESYS
- Evaluieren des Konzepts und der Ergebnisse

1.3 Aufbau der Arbeit

Neben dieser Einleitung und der Zusammenfassung am Ende gliedert sich diese Arbeit in die folgenden vier Kapitel.

Kapitel 2 beschreibt die für diese Arbeit benötigten Grundlagen. In diesem Kapitel werden die Begriffe des Refactoring und der Metriken eingeführt, da diese für die folgenden Kapitel benötigt werden. Außerdem wird auf CODESYS und den IEC-61131-3 Standard eingegangen.

Kapitel 3 stellt die für dieses Konzept verwendeten Metriken vor, definiert die Bedingungen und beschreibt das Vorgehen für das Refactoring.

Kapitel 4 behandelt die Umsetzung des Konzepts aus dem vorherigen Kapitel als Plugin für CODESYS. Focus liegt dabei darauf, welche Besonderheiten bei der Umsetzung des Konzepts zu beachten sind.

Kapitel 5 beinhaltet eine Evaluation des Konzeptes unter Verwendung des Plugins aus dem vorherigen Kapitel.

2 Grundlagen

Dieses Kapitel beschreibt alle für die Arbeit notwendigen Grundlagen. Zuerst werden die Grundlagen zu Metriken erläutert. Danach wird auf Refactoring und anschließend auf die Metrik Kohäsion eingegangen. Abschließend werden das Tool CODESYS und die IEC-61131-Sprachen eingeführt, da diese für die exemplarische Umsetzung des Konzepts benötigt werden. Dabei wird auf deren Anwendungsbereich und Besonderheiten im Vergleich zu herkömmlichen Programmiersprachen eingegangen.

2.1 Metriken

Im Standard 1061 [IEE92] der IEEE wird Software-Metrik wie folgt definiert:

Definition 2.1 (Metrik)**.** „A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality.“ (IEEE 1992)

Eine Metrik bildet also entsprechend eine bestimmte Eigenschaft einer Software oder eines Teils einer Software auf eine einzelne Kennzahl ab.

2.1.1 Arten

Der IEEE-Standard 1061 [IEE92] teilt Software-Metriken in zwei Kategorien ein.

- *Prozessmetriken*, z.B. eingesetzte Ressourcen oder gefundene Fehler
- *Produktmetriken*, z.B. Anzahl der Funktionen oder Komplexität von Klassen

Fenton [Fen91] unterscheidet zusätzlich die Ressourcenmetriken, die bei ihm nicht Teil der Prozessmetriken sind. Die Kohäsion wird den Produktmetriken zugeschrieben.

2.1.2 Anforderungen an Metriken

Für objektorientierte Metriken gibt es von Brito e Abreu et al. [AC94] sieben Kriterien, die diese erfüllen sollten.

- Metrik muss formal definiert sein
- Metrik sollte unabhängig von der Systemgröße sein
- Wert der Metrik sollte entweder dimensionslos sein oder eine wohldefinierte Einheit haben
- Metriken sollten früh im Entwicklungsprozess verfügbar sein
- Metriken sollten herunterskalierbar sein
- Metriken sollten automatisch berechnet werden können
- Metriken sollten sprachunabhängig sein

Gleich und Munck [GM18] stellen zusätzlich noch die Anforderung an Kennzahlen, dass zugrundeliegende Metrik nachvollziehbar sein muss.

2.1.3 Metriken für objektorientierte Programmierung

Zuse [Zus98] teilt die Metriken für Objektorientierung in drei Level auf, auf denen gemessen werden kann.

- *System-Level*
- *Modul-Level*
- *Klassen-Level*

Grundsätzlich gibt es einige Metriken, die auf mehreren dieser Ebenen gemessen werden können, allerdings mit anderem Kontext. Beispiele dafür sind etwa Anzahl der Klassen, Methoden oder Tiefe von Vererbungshierarchien. Auf Klassen-Ebene können zusätzlich Metriken wie Komplexität, Kopplung oder Kohäsion berechnet werden.

2.1.4 Probleme von Metriken

Das größte Problem von Metriken, lässt sich direkt aus der Definition von Software-Metriken der IEEE ableiten. Dem Wert, den eine Metrik als Ergebnis berechnet, muss eine Bedeutung zugeschrieben werden können. Obwohl es für viele Metriken Richtwerte für die Interpretation gibt, muss auch immer der Kontext betrachtet werden. Anhand der LOC-Metrik lässt sich exemplarisch zeigen, dass dies nicht immer leicht ist. So kann ein erfahrener Entwickler ein Problem vielleicht mit einem halb so hohen LOC-Wert lösen, dafür kann aber die längere Variante verständlicher und dadurch deutlich leichter zu warten sein.
Ein weiteres Problem ist, dass es Fälle gibt, in denen sich durch das Design schlechte Ergebnisse für eine Metrik ergeben, obwohl das Design nicht grundsätzlich falsch bzw. schlecht ist. Ein Beispiel dafür ist das Besucher-Entwurfsmuster [GHJV11], welches etwa in der Compilertechnologie häufig eingesetzt wird. Für die Kopplung ergibt sich durch das Entwurfsmuster ein schlechter Wert für die Metrik.

2.1.5 Gründe für Verwendung von Metriken

Allgemein erlauben Metriken eine abstrakte Bewertung von Eigenschaften einer Software. Diese Werte können beispielsweise bei der Planung und Abschätzung von Projekten als Hilfe dienen.
Ein weiterer Grund ist, dass Metriken heutzutage in der Regel automatisch berechnet werden, beispielsweise mittels Werkzeugen der statischen Codeanalyse oder im Rahmen von kontinuierlicher Integration. Das heißt, dass für die Berechnung kaum Aufwand entsteht. Gleichzeitig können Metriken aber auch dabei helfen, Stellen mit Refactorierungsbedarf im Code zu finden.

2.2 Refactoring

Martin Fowler unterscheidet in seinem Buch [Fow20] zwischen einem konkreten Refactoring, dass er definiert als

Definition 2.2 (Refactoring - Aktion)**.** „Eine Änderung an der internen Struktur von Software, um sie verständlicher zu machen und Modifikationen zu erleichtern, ohne dass sich das sichtbare Verhalten verändert“ (Fowler 2020)

und der Tätigkeit des Refactorings, die er beschreibt als

Definition 2.3 (Refactoring - Tätigkeit)**.** „Software durch die Anwendung einer Reihe von Refactorings umzustrukturieren, ohne dass sich das sichtbare Verhalten verändert“ (Fowler 2020)

. Zusätzlich sind Refactorings kleine in sich abgeschlossene Schritte. Große Umbauten sind eine Aneinanderreihung oder Kombination von einzelnen Refactorings.

2.2.1 Gründe

Nach Fowler gibt es die folgenden vier Gründe für Refactoring [Fow20].

- *Verständlichere Software*: durch den Erhalt oder die Verbesserung der Architektur ist es einfacher einzelne Komponenten zu verstehen, wodurch die gesamte Software einfacher verstanden werden kann
- *Verbessern des Designs der Software*: ohne Refactoring wird durch Änderungen die Struktur verändert, was wiederum das Verständnis des Designs der Software erschwert
- *Finden von Fehler erleichtert*: durch Refactoring werden einzelne Funktionalitäten der Software besser verstanden, somit kann falsches Verhalten leichter entdeckt werden
- *Ermöglicht schnelleres Programmieren*: durch Refactoring wird das Design erhalten, was das Verständnis der Funktionalität der Software erleichtert, somit wird ab einem gewissen Zeitpunkt in der Entwicklung mehr Zeit eingespart, als durch das Refactoring verloren gegangen ist

Zusammengefasst ist der Hauptgrund für Refactoring das bessere Verständnis der gesamten Software. Darauf bauen der zweite und der dritte Grund auf.

2.2.2 Anforderungen

Die wichtigste Anforderung ergibt sich bereits aus den beiden Definitionen. Refactoring darf das nach außen sichtbare Verhalten nicht verändern. Das heißt insbesondere, dass für die vom Refactoring betroffene Funktionalität Tests vorhanden sein müssen. Auch müssen alle Tests, die vor dem Refactoring erfolgreich durchlaufen wurden, auch danach erfolgreich durchgeführt werden. Die einzige Änderung an den Tests darf bei entsprechenden Refactorings an den genutzen Schnittstellen sein. Das bedeutet auch, dass durch ein korrekt ausgeführtes Refactoring keine neuen Fehler eingebaut werden. Es können aber bisher verdeckte Fehler aufgedeckt werden.
Außerdem muss es Bedingungen geben, die bestimmen, wann welches Refactoring

durchgeführt wird. Zum einen, da es zu einigen Refactorings ein inverses Refactoring gibt, z.B. Funktion extrahieren und Funktion inline platzieren. Zum anderen gibt es Refactorings, bei denen das wiederholte Ausführen des Refactorings zurück zum Ursprungszustand führt, z.B. Funktion in eine andere Klasse verschieben. Fowler benutzt für diese Bedingungen den Begriff der *Code-Smells*, die er als eine bestimmte Struktur im Code definiert, welche darauf hindeuten, dass Refactoring geprüft werden sollte [Fow20]. Er führt 24 Code-Smells auf, z.B. kryptische Namen und umfangreiche Klassen. Entscheidend dabei ist, dass es zu jedem Code-Smell mindestens ein Refactoring gibt, mit dem dieser beseitigt werden kann. Code-Smells können beispielsweise im Rahmen der statischen Codeanalyse gefunden werden.
Auch sollte es für jedes Refactoring eine konkrete Reihenfolge an Aktionen geben, um es erfolgreich durchzuführen. Kann das Refactoring formal definiert werden, ist zudem zumindest in Teilen eine automatisierte Unterstützung möglich.

2.2.3 Automatisierung von Refactoring

Inzwischen haben einige Entwicklungsumgebungen integrierte Werkzeuge oder es gibt entsprechende Erweiterungen, um Refaktoring zu automatisieren oder zumindest zu unterstützen, z.B. ReSharper bei Visual Studio oder integriert bei Eclipse und IntelliJ IDEA. Der Umfang der unterstützten Refactorings hängt dabei von der Entwicklungsumgebung ab. So werden etwa für das Umbenennen einer Methode Informationen über die Struktur der Software benötigt, um auch bei Referenzen der umzubenennenden Methode den Namen entsprechend zu ändern. Visual Studio etwa verwendet hierzu die Informationen aus dem Precompile. Dabei werden die wenigstens Refactorings vollständig automatisiert durchgeführt, nach dem eine Code-Smell gefunden wurde. Der Grund besteht vor allem darin, dass ein entsprechendes Tool keine Kenntnis über den Kontext des entdeckten Code-Smells hat. So kann etwa eine Klasse, die sich noch in der Implementierungsphase befindet, noch nicht benutzte Methoden oder Variablen besitzen. Diese Information würde einem Tool fehlen. Außerdem kann es zu einem Code-Smell mehrere Refactorings geben, um diesen zu beseitigen, was ein Werkzeug nur heuristisch entscheiden könnte. Deshalb wird in der Regel, wenn ein Code-Smell entdeckt wird, dem Entwickler eine Auswahl an Refactorings vorgeschlagen. Ein so ausgewähltes Refactoring wiederum kann dann, soweit technisch möglich automatisch durchgeführt werden. Ausnahme sind etwa Formatierungsfehler wie fehlende Einrückungen oder Leerzeichen. Diese werden meist automatisch korrigiert.

2.3 Kohäsion

Kohäsion beschreibt den inneren Zusammenhang einer Programmeinheit. Eine Programmeinheit kann dabei eine einzelne Funktion, eine Klasse oder ein gesamtes Modul sein. Da diese Arbeit ihren Fokus auf der Kohäsion bei Klassen hat, ist hier noch das Single-Responsibility-Prinzipt [Mar09] der objektorientierten Programmierung zu nennen, welches besagt, dass sich eine Programmeinheit um genau eine Funktionalität kümmern soll. Dies entspricht dem Prinzip der Kohäsion, da Programmeinheiten, die sich um dieselbe Funktionalität kümmern, enger zusammenhängen, als Programmeinheiten, die sich um unterschiedliche Funktionalitäten kümmern.
Im Zusammenhang mit der Kohäsion ist auch die Kopplung zu nennen, welche die externe Verknüpfung von Modulen oder Klassen beschreibt.

2.3.1 Kategorien von Kohäsion

Wuest et al. [WDB97] unterscheiden fünf Kategorien von Kohäsion in Klassen.

- *Trennbare Kohäsion*: es existieren mehrere Abstraktionen innerhalb der Klasse, die nicht miteinander in Beziehung stehen
- *Vielfältige Kohäsion*: es existieren mehrere Abstraktionen innerhalb der Klasse, die aber durch mindestens eine Methode alle miteinander in Beziehung stehen
- *Nicht-delegierende Kohäsion*: es existiert eine Abstraktion innerhalb der Klasse, aber es gibt Variablen, die nur mit einzelnen Komponenten davon in Beziehung stehen
- *Verborgene Kohäsion*: es existiert eine nützliche Abstraktion innerhalb der Klasse, aber es gibt einzelne Methoden und Variablen, die ausgelagert werden können
- *Modell-Kohäsion*: die Klasse repräsentiert genau eine Abstraktion und alle Methoden und Variablen sind Teil dieser Abstraktion

Es ist zu sehen, dass die fünf Kategorien sich Schritt für Schritt dem Single-Responsibility-Prinzip annähern. Somit ist auch bei jeder Kategorie eine höhere Kohäsion als bei der vorherigen Kategorie zu erwarten.

2.3.2 Berechnungsmodelle für Kohäsion in Klassen

In der Literatur gibt es eine Vielzahl von Berechnungsmodellen für objekt-orientierte Systeme, welche sich grundlegend bezüglich der folgenden Eigenschaften einteilen lassen[PM05]:

- Struktur, z.B. LCOM [KC94], TCC, LCC [BK95], ...
- Datenfluss [BO94]
- Semantik
- Data Mining
- weitere, wie Wissens-basierte, Entropie-basierte, u.a.

Neben den Eigenschaften auf denen die verschiedenen Berechnungsmodelle basieren, unterscheiden sie sich auch im Hinblick des Wertebereichs, auf den sie abbilden. Während die meisten Modelle normalisiert auf den Bereich $[0, 1]$ abbilden, bildet etwa LCOM1 auf die natürlichen Zahlen ab. Des weiteren gibt es einen Unterschied bei der Bedeutung des Wertes, da bei fast allen Berechnungsmodellen ein höherer Wert eine höhere Kohäsion bedeutet. Ausnahme sind hier wiederum die LCOM-Metriken, bei denen, entsprechend dem Namen, ein niedrigerer Wert einer höheren Kohäsion entspricht.

2.3.3 Probleme der Kohäsion als Metrik

Wie bei den meisten anderen Metriken, ist auch bei der Kohäsion der Vorteil, dass eine Eigenschaft auf eine bloße Zahl abgebildet wird, zeitgleich das größte Problem, solange dieser Zahl keine Bedeutung zugeschrieben werden kann. Im Gegensatz zu den meisten anderen Metriken gibt es in der Literatur für die Kohäsion so gut wie keine Richtwerte. In der Regel wird stattdessen nur von starker oder hoher bzw. von schwacher oder niedriger Kohäsion gesprochen. Die Ausnahme bilden die beiden Graphen, die in Abbildung 2.1 zu sehen sind. Unabhängig vom Wert der Berechnung handelt es sich beim linken Graph um minimale und beim rechten Graph um maximale Kohäsion.
Außerdem gibt es auch bei der Kohäsion Fälle, in denen aufgrund des Designs niedrige Kohäsion zu erwarten ist. Dies ist beispielsweise bei Utility-Klassen der Fall.

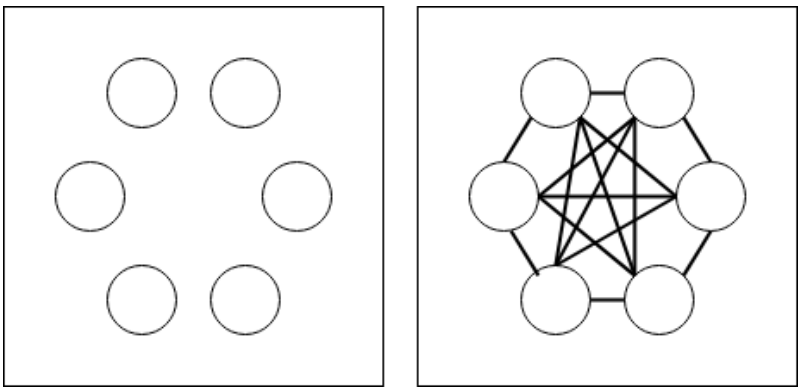

Abbildung 2.1: Die linke Grafik zeigt einen Graphen mit minimaler Kohäsion, die rechte einen Graphen mit maximaler Kohäsion.

2.4 Codesys und IEC-61131-3

CODESYS[1] ist ein herstellerunabhängiges Programmiersystem für speicherprogrammierbare Steuerungen. Dabei werden entsprechend dem Standard IEC 61131-3 [WZ16] die graphischen Programmiersprachen Kontaktplan (KOP), Funktionsbausteinsprache (FUP) und Ablaufsprache (AS) sowie die textuelle Sprache Strukturierter Text (ST) angeboten. Das Klassen-äquivalent in CODESYS ist der Funktionsbaustein (FB). In dieser Arbeit wird der Focus auf ST gelegt.

2.4.1 Besonderheiten

Bei CODESYS gibt es bedingt durch den Sprachstandard und die Entwicklung der SPS-Programmierung einige Besonderheiten im Vergleich zu herkömmlichen Hochsprachen. Da dies Auswirkungen auf, Umsetzung in Kapitel 4 hat, werden diese im Folgenden kurz erörtert.
Applikationen laufen immer zyklisch in Tasks ab, d.h. eine Applikation stoppt nur im Fehlerfall oder wenn der Nutzer sie stoppt.
Es gibt neben Methoden auch Aktionen. Diese haben im Unterschied zu Methoden keine Parameter und keinen Rückgabewert und arbeiten nur auf den Variablen des FBs.
Methoden, Aktionen und Properties sind eigenständige Kindobjekte eines FBs und werden nicht textuell im Rumpf des FBs angelegt.
Variablen können nur im Deklarationsteil angelegt werden und müssen von einem entsprechenden Sprachkonstrukt eingerahmt werden, das angibt, um welche Art von Variable (z.B. Input-Variable) es sich handelt.

[1] `https://de.codesys.com/das-system.html`

Jeder FB hat eine eigene main-Methode. Somit kann eine Instanz eines FBs wie eine normale Methode aufgerufen werden. Entsprechend kann ein FB selber Input- und Output-Variablen besitzen. FBs haben keinen Rückgabewert.

2.4.2 Objektorientierung in der SPS-Programmierung

Obwohl objektorientierte Programmierung seit 2006 auch Teil des IEC-61131-3 Standards ist, sind in der SPS-Programmierung nach wie vor viele Projekte klassisch prozedural programmiert. Das hat zum einen den Grund, dass in der Industrie Bestandsprojekte, die freigegeben und im Einsatz sind, in der Regel nicht ersetzt werden. Außerdem sind die Anwendungsfälle in den jeweiligen Steuerungen jeweils recht ähnlich, so dass auch bei neuen Maschinen Bestandsprojekte neu zu parametrieren schneller geht, als eine Applikation neu zu schreiben. Des Weiteren handelt es sich bei einem Großteil der Applikateure nicht um Software Engineerer, sondern um Personen aus dem Umfeld von Elektrotechnik, Maschinenbau oder ähnlichen Fachrichtungen, die nicht oder kaum mit objektorientierter Programmierung vertraut sind [Gmb16].

3 Konzept

In diesem Kapitel wird das grundlegende Konzept zur Refaktorierung von Klassen auf Basis der Kohäsion erarbeitet. Dazu wird erst kurz auf die grundlegenden Strukturen und Beziehung in Klassen im Allgemeinen eingegangen. Anschließend wird das Datenmodell vorgestellt, in welches die Eigenschaften einer Klasse überführt werden. Danach werden die Kohäsionsmetriken vorgestellt, die in dieser Arbeit verwendet werden. Basierend auf den Metriken wird dann kurz erläutert, wie der Zusammenhang in einer Klasse visualisiert werden kann. Zuletzt wird erläutert, welche Refactorings eingeführt werden.
Essentiell für das hier vorgestellte Konzept sind grundlegende Informationen zur Struktur einer Software. In der Regel sollten dabei die Informationen eines Precompiles, wie ihn etwas leistungsstärkere Entwicklungsumgebungen wie Visual Studio oder IntelliJ haben, ausreichen.

3.1 Beziehungen in Klassen

In der Literatur gibt es hauptsächlich zwei Ansichten, welche Bestandteile einer Klasse bei der Berechnung der Kohäsion berücksichtigt werden sollen. Dabei werden entweder alle Bestandteile in die Berechnung miteinbezogen beziehungsweise es gibt keine Informationen, dass Bestandteile ausgeschlossen werden, oder es werden nur Methoden und Variablen berücksichtigt [AAM19]. Der Ausschluss wird dabei unter anderem damit begründet, dass es sich etwa bei Properties und Konstruktoren um kleine Bestandteile mit wenig Beziehung innerhalb der Klasse handelt und das die Berechnung der Kohäsion verfälscht wird.
In dieser Arbeit werden zusätzlich zu Variablen und Methoden auch Properties beachtet, da Getter und Setter zur Datenkapselung beitragen, welche eines der elementaren Prinzipien der Objektorientierung ist. Konstruktoren werden für die Berechnung der Metriken als normale Methoden behandelt, müssen aber beim Refactoring speziell behandelt werden.
Für die drei genannten Bestandteile einer Klasse lassen sich die folgenden drei elementaren Operationen identifizieren.

- Lesen von Variablen
- Schreiben von Variablen

- Aufrufe von Methoden

Aus diesen Operationen können wiederum die folgenden Beziehungen abgeleitet werden.

- Methode-zu-Methode
- Methode-zu-Property
- Methode-zu-Variable
- Property-zu-Property
- Property-zu-Variable
- Variable-zu-Variable

Wie eingangs erwähnt gilt dabei zu beachten, dass dieses Konzept nur auf elementaren Beziehungen in einer Klasse aufbaut und keine weiteren Informationen, wie sie beispielsweise eine Datenflussanalyse liefern könnte, braucht. Darum handelt es sich bei folgender Beispielklasse nur um zwei Methode-zu-Variable Beziehungen, zwischen der Methode DoSomething und der Variablen a, so wie zwischen der Methode DoSomething und der Variablen b. Es handelt sich aber nicht um eine Variable-zu-Variable Beziehung zwischen a und b.

```
public class Dummy {
  private bool a = true;
  private bool b = false;

  public void DoSomething() {
    a = b;
  }
}
```

Aus diesem Grund handelt es sich bei der Beziehung Variable-zu-Variable um einen Sonderfall, der vor allem im Zusammenhang mit Konstanten vorkommt, beispielsweise bei der Initialisierung von Arrays.

3.2 Datenmodell zur Abbildung der Beziehungen in einer Klasse

Für die Repräsentation der Beziehungen einer Klasse K wird im Folgenden als Datenstruktur ein Graph $G = (V, E)$ verwendet.
Dabei sei die Menge der Knoten $V = A \cup P \cup M$ mit

A = Menge der Variablen von K
P = Menge der Properties von K
M = Menge der Methoden von K
und die Menge der Kanten E = Menge der Referenzen von K. Dabei gilt, dass die Referenz zwischen zwei Knoten ungerichtet ist, das heißt für eine beliebige Kante $(u, v) \in E$ existiert eine Kante $(v, u) \in E$.
Des weiteren sei R_i^X = Menge der Referenzen von Knoten i zu Knoten vom Typ X, mit $i \in Y$, $Y \in \{V, A, P, M\}$ und $X \in \{V, A, M\}$.
So werden beispielsweise etwa mit $R_{m_i}^A$ alle Referenzen von Methode m_i zu Variablen beschrieben.
Außerdem werden den Knoten Gewichte zugeschrieben. Die Gewichte der Knoten werden dabei später benutzt, um die Kohäsion der Klasse zu berechnen. Im weiteren Verlauf dieser werden die Knotengewichte als *Teilkohäsion* bezeichnet. Die Kanten hingegen werden für die Visualisierung der Kohäsion benötigt.
Zusätzlich wird dem Graphen noch eine Matrix beigefügt. Sie spiegelt dabei die Kantengewichte einer vollständigen Version des Graphen wieder, und wird für die paarweisen Knotenabstände aller Knoten im Rahmen der Visualisierung benötigt.

3.2.1 Gewichtsmatrix

	0	$\cdots$	$m-1$	m	$\cdots$	$p-1$	p	$\cdots$	$v-1$
0	$w_{0,0}$	$\cdots$	$w_{0,m-1}$	$w_{0,m}$	$\cdots$	$w_{0,p-1}$	$w_{0,p}$	$\cdots$	$w_{0,v-1}$
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$m-1$	$w_{m-1,0}$	$\cdots$	$w_{m-1,m-1}$	$w_{m-1,m}$	$\cdots$	$w_{m-1,p-1}$	$w_{m-1,p}$	$\cdots$	$w_{m-1,v-1}$
m	$w_{m,0}$	$\cdots$	$w_{m,m-1}$	$w_{m,m}$	$\cdots$	$w_{m,p-1}$	$w_{m,p}$	$\cdots$	$w_{m,v-1}$
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$p-1$	$w_{p-1,0}$	$\cdots$	$w_{p-1,m-1}$	$w_{p-1,m}$	$\cdots$	$w_{p-1,p-1}$	$w_{p-1,p}$	$\cdots$	$w_{p-1,v-1}$
p	$w_{p,0}$	$\cdots$	$w_{p,m-1}$	$w_{p,m}$	$\cdots$	$w_{p,p-1}$	$w_{p,p}$	$\cdots$	$w_{p,v-1}$
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$v-1$	$w_{v-1,0}$	$\cdots$	$w_{v-1,m-1}$	$w_{v-1,m}$	$\cdots$	$w_{v-1,p-1}$	$w_{v-1,p}$	$\cdots$	$w_{v-1,v-1}$

Tabelle 3.1: Gewichtsmatrix der Graphstruktur

Tabelle 3.1 zeigt schematisch die Struktur der Matrix für die zusätzlichen Kantengewichte. Die Matrix hat die Größe $|V| \times |V|$ und ist symmetrisch, da die Richtung für den Abstand in der Visualisierung nicht von Interesse ist. Für die Indizes von Spalten und Reihen gilt dabei

- 0 bis $m-1$: Methoden
- m bis $p-1$: Properties

- p bis $v-1$: Variablen

mit $m = |M|$, $p = |M| + |P|$ und $v = |V|$.

Damit spiegeln die Bereiche der Matrix die folgenden Knotenbeziehungen wieder:

- $w_{0,0}$ bis $w_{m-1,m-1}$: Methode-zu-Methode
- $w_{m,0}$ bis $w_{p-1,m-1}$: Methode-zu-Property
- $w_{p,0}$ bis $w_{v-1,m-1}$: Methode-zu-Variable
- $w_{m,m}$ bis $w_{p-1,p-1}$: Property-zu-Property
- $w_{p,m}$ bis $w_{v-1,p-1}$: Property-zu-Variable
- $w_{p,p}$ bis $w_{v-1,v-1}$: Variable-zu-Variable

Die restlichen drei Bereiche sind entsprechend der Symmetrie Kopien der Bereiche Methode-zu-Property, Methode-zu-Variable und Property-zu-Variable.

3.3 Berechnungsmodelle für Kohäsion

In diesem Abschnitt werden die beiden Kohäsionsmetriken vorgestellt, die in dieser Arbeit verwendet werden. Ziel bei der Auswahl der Metriken war es insbesondere, dass die Informationen, die der Zusammenhangsgraph bietet, für die Berechnung der Kohäsion ausreichen. Außerdem muss die Kohäsion von verschiedenen Klassen, unabhängig von ihrer Größe und Struktur, vergleichbar sein, weswegen nur eine normierte Metrik in Frage kommt.

3.3.1 Gewichtete direkte Referenzen

Die gewichteten direkten Referenzen bilden am genauesten die Struktur des Zusammenhangsgraphen ab. Außerdem lässt sich leicht die Datenbasis ändern, wodurch beispielsweise leicht der Unterschied zwischen Datenreferenzen und allen Referenzen betrachtet werden kann.

Teilkohäsion

Definition 3.1 (Teilkohäsion)**.** Kohäsion eines Knoten im Zusammenhangsgraph, die dazu verwendet wird, die Kohäsion der Klasse zu berechnen.

Die Teilkohäsion eines Knoten durch die gewichteten direkten Referenzen ist gegeben durch

$$Coh_x = \frac{|R_x^Y|}{|Y|}$$
$$\text{mit} \quad x \in X,\ X \in \{V, A, P, M\},\ Y \in \{V, A, M\}$$

.

Kohäsion

Die Kohäsion einer Klasse ist entsprechend durch

$$Coh = \frac{\sum_{v \in V} \frac{|R_v^Y|}{|Y|}}{|V|} = \frac{\sum_{v \in V} |R_v^Y|}{|Y| * |V|}$$
$$\text{mit} \quad Y \in \{V, A, M\}$$

definiert.

3.3.2 Hamming Distanz

Mittels der Hamming Distanz kann die Ähnlichkeit von zwei Knoten bestimmt werden. In dieser Arbeit wird sie genutzt um die Kantengewichte für die Visualisierung zu bestimmen.

$$Coh_{(u,v)} = \frac{|R_v^V \cap R_u^V|}{|R_v^V \cup R_u^V|}$$
$$\text{mit} \quad v, u \in V$$

Die Berechnungen für die Teilkohäsion und die Kohäsion sind der Vollständigkeit halber im Folgenden kurz aufgeführt, werden in dieser Arbeit aber nicht weiter verwendet.

Teilkohäsion

$$Coh_v = \frac{\sum_{u \in V} \frac{|R_v^V \cap R_u^V|}{|R_v^V \cup R_u^V|}}{|R_v^V|} = \sum_{u \in R_v^V} \frac{|R_v^V \cap R_u^V|}{|R_v^V \cup R_u^V| * |V|}$$
$$\text{mit} \quad v \in V$$

Kohäsion

$$Coh = \frac{\sum_{v \in V} \sum_{u \in V} \frac{|R_v^V \cap R_u^V|}{|R_v^V \cup R_u^V| * |V|}}{|V|} = \sum_{v \in V} \sum_{u \in V} \frac{|R_v^V \cap R_u^V|}{|R_v^V \cup R_u^V| * |V| * |V|}$$

3.4 Visualisierung

Unterstützend zum Refactoring auf Basis des Zusammenhangsgraphen, soll der Graph auch visualisiert werden. Dabei soll der Graph nach Möglichkeit auch die Kohäsion widerspiegeln.

3.4.1 Darstellungsproblem der Kohäsion

Abbildung 3.1 stellt einen Graphen mit maximaler Kohäsion dar, das heißt alle Knoten haben die selbe Teilkohäsion. Soll nun die Kantenlänge fest auf Basis der Kohäsion skaliert werden, so sollten alle Kanten gleich lang sein. Wie anhand der Kanten (A,C) und (B,D) im Verhältnis zu den restlichen Kanten zu sehen ist, ist das nicht möglich, da sich das Problem analog zum XOR-Problem [CTC20] verhält. Somit ist in der n-ten Dimension nur ein vollständiger Graph mit n + 1 Knoten so darstellbar.

3.4.2 Kräfte-basierte Graphen

Da eine Darstellung mit fixer Kantenlänge nicht in allen Fällen möglich ist, soll eine möglichst gute Annäherung an die optimale Kantenlänge erreicht werden. Eine Möglichkeit dies zu erreichen, ist mittels Kräfte-basierter Graphen [Tut63]. Basierend auf ihrer Beziehung zueinander, stoßen sich die Knoten ab beziehungsweise ziehen sich an, bis sich der Graph in einem stabilen Zustand befindet.

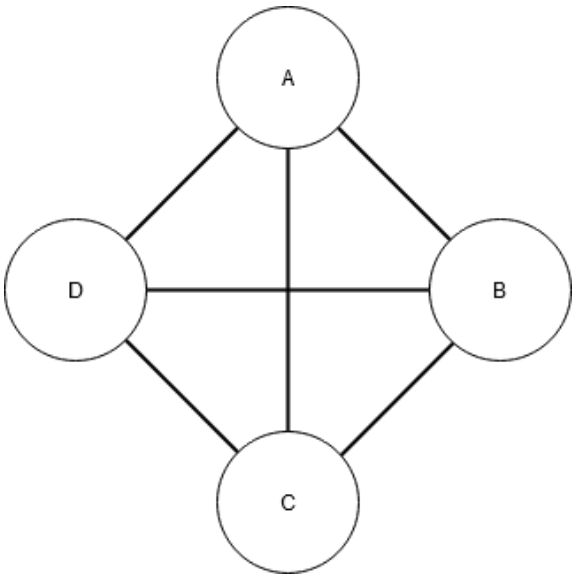

Abbildung 3.1: Vollständiger Graph mit vier beliebigen Knoten. Das heißt, jeder Knoten hat die selbe Teilkohäsion und damit auch sind auch alle Kantengewichte gleich. Zu sehen ist aber, dass die Kanten (A,C) und (B,D) deutlich länger sind.

Fruchterman-Reingold-Algorithmus

Da der Spring-EmbedderAlgorithmus [Ead84] für jeder Knotenbeziehung entweder nur abstoßende oder nur anziehende Kräfte berechnet, besteht das Problem, dass sich bei Graphen mit wenig Kanten die Knoten kreisförmig auf maximalen Abstand entfernen. Bei einer hohen Anzahl an Kanten, werden die Knoten hingegen sehr eng zusammengedrückt, so dass in beiden Fällen die Struktur nur bedingt zu erkennen ist. Aus diesem Grund wurde für die Visualisierung der Fruchterman-Reingold-Algorithmus (FRA) [FR91] gewählt um die Knoten zu platzieren. Dieser weißt Knoten die in einer Beziehung stehen abstoßende Kräfte zu, so dass diese nicht zu eng zueinander gezogen werden.

3.4.3 Anpassung des Fruchterman-Reingold-Algorithmus

Da der Fruchterman-Reingold-Algorithmus hier trotzdem noch das Problem hat, dass unzusammenhängende Knoten schnell weit auseinander driften und der Algorithmus auch mit fixen Kantelängen arbeitet wurden einige Anpassungen vorgenommen.

Abstand der Knoten

Um die Kohäsion miteinzubeziehen, wird jeder Kante (u, v) mit $u, v \in V$ ein Gewicht $w = 1 - Coh_{(u,v)}$ zugewiesen. Diese Gewicht wird mit der Standardkantenlänge s multipliziert, so dass Knoten mit höherer Kohäsion näher beieinander liegen. Des weiteren sei $\vec{r}$ ein Zufallsvektor der Länge 1 und $d(u, v)$ die Distanz von u und v. Dann ergibt sich die anziehende Kraft F_{attr} durch

$$F_{attr} = \begin{cases} \vec{0} & \text{if} w = 0, d(u,v) = 0 \\ \vec{r} & \text{if} w > 0, d(u,v) = 0 \\ \frac{||(u-v)|| * (s*w)^2}{d(u,v)} & \text{else} \end{cases}$$

und die abstoßende Kraft durch

$$x = \begin{cases} \vec{0} & \text{if} w = 0, d(u,v) = 0 \\ \vec{r} & \text{if} w > 0, d(u,v) = 0 \\ ||(u-v)|| & \text{if} d(u,v) > 0, w0 \\ \frac{||(u-v)|| * d(u,v) * d(u,v)}{(s*w)^2} & \text{else} \end{cases}$$

Initialisierung

Da endgültige Knotenposition stark von der initialen Positionierung abhängt, wurde ähnlich zum GRIP-Algorithmus [GK00] eine Initialisierungsphase eingefügt, in der nur die Variablen-Knoten mittels des Algorithmus vorplatziert werden. Erst danach werden die restlichen Knoten im Schwerpunkt ihrer referenzierten Knoten platziert.

Zusätzliche Schwerpunkte

Um das zu schnelle abdriften der Knoten zu verhindern, wurden zwei weitere Modifizierungen eingebaut. Zum einen bekommt jeder Knoten den Schwerpunkt seiner referenzierten Knoten als referenzierenden Knoten. Außerdem werden in regelmäßigem Abstand nicht die Knoten entsprechend der anderen Knoten. Stattdessen werden sie etwas in Richtung des Mittelpunktes bewegt. Dabei werden die Bewegungen Richtung Schwerpunkt und Mittelpunkt mit einem Gewicht von 0,25 skaliert, was sich als guter Wert erwiesen hat, um das austarieren der Knoten nicht zu stark zu beeinflussen aber trotzdem ein Abdriften verhindert.

3.4.4 Ablauf des Algorithmus

Damit läuft der modifizierte Frucherman-Reingold-Algorithmus folgendermaßen ab:

```
// Variablen-Knoten platzieren
plaziere_variablenknoten()
for i ← 1 to 10 do
    führe_algorithmus_aus()

// restliche Knoten platzieren
plaziere_restliche_knoten()
for i ← 1 to 50 do
    // jede zehnte Runde alle Knoten
    // Richtung Mittelpunkt bewegen
    if i % 10 == 9 do
        bewege_knoten_mittelpunkt()
    else
        führe_algorithmus_aus()
```

3.5 Refaktorierung

In diesem Abschnitt werden die Refaktorierungsoperationen vorgestellt und darauf eingegangen, welche formalen Bedingungen erfüllt sein müssen um die jeweilige Operation anwenden zu können. Im Gegensatz zu der Arbeit von Simon et al.[SLS01], welche einen Distanz-basierten Ansatz zur Identifizierung von Refaktorierungsmöglichkeiten betrachten, werden hier die Eigenschaften des Zusammenhangsgraphen der jeweiligen Klasse verwendet. Außerdem wird auf einige Spezialfälle eingegangen.

3.5.1 Code-Smells

In diesem Abschnitt werden zwei Code-Smells vorgestellt, die aus der Kohäsion und der Struktur des Zusammenhangsgraphen abgeleitet werden können.

Mehrere unabhängige Teilgraphen

Besteht der Zusammenhangsgraph einer Klasse aus mehreren unabhängigen Teilgraphen, so deutet dies auf mehrere Funktionalitäten innerhalb einer Klasse hin, was

das Single-Responsibility-Prinzip der objektorientierten Programmierung verletzt. Formal lässt sich das wie folgt ausdrücken.

$$\exists G'(V', E') \in G : \exists G''(V'', E'') \in G : G' \cap G'' = \varnothing \wedge G' \cup G'' = G$$
$$\wedge |V'| > 1 \wedge |V''| > 1 \wedge \forall v \in V' : \forall u \in V'' : R_v^V \cap R_u^V = \varnothing$$
$$\wedge \forall p \in V' : \forall q \in V' : path(p, q) \wedge \forall v \in V' : Coh_v > 0$$
$$\text{mit } path(u, v) = \begin{cases} 1 & \text{if es existiert ein Pfad von Knoten } u \text{ zu Knoten } v \\ 0 & \text{else} \end{cases}$$

Diese Cluster lassen sich durch eine Breiten- oder Tiefensuche identifizieren. Ein Cluster zeichnet sich insbesondere dadurch aus, dass jeder Knoten eine Teilkohäsion größer 0 besitzt.

Isolierter Knoten

Ein isolierter Knoten v zeichnet sich dadurch aus, dass er keine Kante zu irgendeinem anderen Knoten besitzt.

$$\exists v \in V : R_v^X = \varnothing \wedge Coh_v = 0$$

Da ein solcher Knoten außerdem eine Teilkohäsion von 0 besitzt, kann er leicht identifiziert werden.

3.5.2 Refactoring

Im Folgenden werden die einzelnen Refactorings beschrieben, die auf Basis der entdeckten Code-Smells ausgeführt werden können. Dabei verwendet das Refactoring *Aufteilen von Klassen* den Code-Smell *mehrere unabhängige Teilgraphen*. Die restlichen Refactorings basieren auf dem Code-Smell *isolierte Knoten*.

Aufteilen von Klassen

Besteht der Zusammenhangsgraph aus mehreren unabhängige Teilgraphen, so deutet dies darauf hin, dass die entsprechende Klasse für mehr als eine Abstraktion zuständig ist. In diesem Fall sollte die Klasse aufgeteilt werden. Um die Funktionalität der Software zu bewahren und gleichzeitig so wenig Änderungen wie möglich am

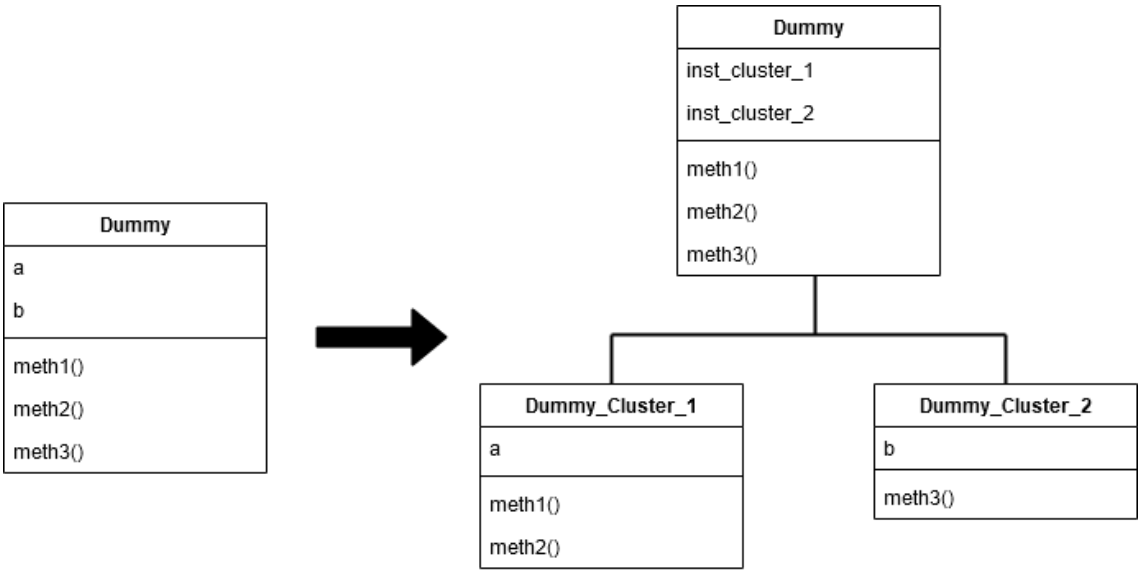

Abbildung 3.2: Die Grafik zeigt schematisch das Entwurfsmuster Fassade. Die Variablen a und b, sowie die Methoden meth1, meth2 und meth3 werden in die beiden neuen Klassen Dummy_Cluster_1 und Dummy_Cluster_2 verschoben. Die Klasse Dummy bleibt bestehen und erhält eine Instanz der beiden neuen Klassen. Aufrufe der drei Methoden werden dann von Dummy an die entsprechende Instanz umgeleitet.

Code vornehmen zu müssen, wurde für diese Arbeit entschieden, das Entwurfsmuster *Fassade* [GHJV11] zu verwenden.

Abbildung 3.2 zeigt schematisch das Refactoring. Dabei besitzt die Klasse DUMMY die beiden folgenden Cluster:

- Cluster_1: {a, meth1, meth2}
- Cluster_2: {b, meth 3}

Das Refactoring läuft dabei wie folgt ab:

1. für jedes Cluster eine neue Klasse anlegen
2. alle Variablen in die neue Klasse ihres Clusters verschieben
3. alle Methoden in die neue Klasse ihres Clusters kopieren, dabei Methode in der alten Klasse nicht löschen
4. von jeder neuen Klasse eine Instanz in der alten Klasse anlegen
5. die Funktionalität der Methoden der alten Klasse durch einen Aufruf der Methode der entsprechenden neuen Klasse ändern
6. die Schritte 3 bis 5 für Properties wiederholen

Die Methode meth1 der alten Klasse schaut dann nach dem Refactoring so aus:

```
public void meth1() {
  this.inst_cluster_1.meth1();
}
```

Auf diese Art muss keine sonstige Änderung im Code erfolgen, da alle Verwendungsstellen der ursprünglichen Klasse diese nach wie vor verwenden können. Für zukünftige Implementierungen hingegen können dann entsprechend der benötigten Funktionalität die einzelnen neuen Klassen verwendet werden.

Verschieben von Methoden

Beim Verschieben von Methoden wird zwischen zwei Fällen unterschieden. Hat eine Methode Referenzen zu einer anderen Klasse, muss geprüft werden, ob die Kohäsion der anderen Klasse nicht verschlechtert wird, wenn die Methode in diese Klasse verschoben wird. Ist die Berechnung der Teilkohäsion für alle Knotentypen identisch, kann die Veränderung der Kohäsion einer anderen Klasse $G'(V', E')$ durch hinzufügen der Methode m wie folgt berechnet werden:

$$Coh'_{G'} = \frac{|R_m^{X'}| + \sum_{v \in V'} |R_v^{X'}|}{(x * (|V'| + 1)}$$

$$\text{mit } R_m^{X'} = \text{Anzahl der Referenzen von m in G'}$$
$$\text{und } x = \begin{cases} |X'| + 1 & \text{if } m \in X' \text{ nach dem Verschieben} \\ |X'| & \text{else} \end{cases}$$

Daraus ergibt sich für die Referenzen von m in G' die folgende Anforderung, wobei $x = |X'|$ verwendet wird, da dies die striktere Bedingung für einen Erhalt der Kohäsion ist.

$$Coh'_{G'} \geq Coh_{G'}$$
$$\frac{n + \sum_{v \in V'} |R_v^{X'}|}{(|X'| + 1) * (|V'| + 1)} \geq \frac{\sum_{v \in V'} |R_v^{X'}|}{(|X'| * |V'|)}$$
$$(n + \sum_{v \in V'} |R_v^{X'}|) * (|X'| * |V'|) \geq \sum_{v \in V'} |R_v^{X'}| * (|X'| + 1) * (|V'| + 1)$$
$$n * |X'| * |V'| \geq \sum_{v \in V'} |R_v^{X'}| * (|X'| + |V'| + 1)$$
$$n * |X'| * |V'| \geq Coh_{G'} * |V'| * |X'| * (|X'| + |V'| + 1)$$
$$n \geq Coh_{G'} * (|X'| + |V'| + 1)$$
$$\text{mit } n = |R_m^{X'}| = \text{Anzahl der Referenzen die } m \text{ in } G' \text{ hätte}$$
$$\text{und } Coh_{G'} = \text{der Kohäsion von } G' \text{ vor dem Hinzufügen von } m$$
$$\text{und } Coh'_{G'} = \text{der Kohäsion von } G' \text{ nach dem Hinzufügen von } m$$

Da es sich bei der Anzahl der Referenzen um eine ganze Zahl handelt, muss

$$|R_m^{X'}| \geq \lceil Coh_{G'} * (|X'| + |V'| + 1) \rceil$$

gelten.

Damit muss folgende Bedingung erfüllt sein, damit die Methode m in eine andere Klasse G' verschoben werden darf.

$$|R_m^{X'}| \geq \lceil Coh_{G'} * (|X'| + |V'| + 1) \rceil \wedge |R_m^{X'}| > 0$$

Ist diese Bedingung erfüllt, läuft das Refactoring wie folgt ab:

1. alle Aufrufe der Methode merken
2. Methode in die neue Klasse verschieben
3. in der neuen Klasse die Aufrufe der Methode zu lokalen Aufrufen ändern, d.h. Namespaces und Instanznamen im Aufruf entfernen
4. für alle anderen Aufrufe prüfen, ob die entsprechende Klasse bereits eine Instanz der neuen Klasse der Methode besitzt, sonst Instanz hinzufügen

5. alle Aufrufe auf Instanz der neuen Klasse der Methode umbiegen

Der zweite Fall tritt ein, falls das Hinzufügen von m zu G' zu einer Verringerung der Kohäsion von G' führen würde.

$$|R_m^{X'}| < \lceil Coh_{G'} * (|X'| + |V'| + 1) \rceil$$

In diesem Fall wird die Methode in eine Utility-Klasse verschoben.

1. alle Aufrufe der Methode merken
2. Methode in die Utility-Klasse verschieben
3. alle Aufrufe in Aufrufe der Methode in der Utility-Klasse umbiegen

Für dieses Refactoring müssen keine Instanzen angelegt werden, da Utility-Klassen in der Regel statisch sind oder das Entwurfsmuster Singleton [GHJV11] implementieren.
Für beide Refactorings gilt, dass sich nicht um die alten Instanzen gekümmert werden muss. Sollten diese nach dem Refactoring nicht mehr gebraucht werden, werden sie durch das Refactoring *Variable entfernen* entfernt.

Verschieben von Variablen

Für das Verschieben von Variablen gilt analog zum Verschieben von Methoden, dass durch das Verschieben der Variable a in die Klasse G' die Kohäsion nicht gesenkt werden darf.

$$|R_a^{X'}| \geq \lceil Coh_{G'} * (|X'| + |V'| + 1) \rceil \wedge |R_a^{X'}| > 0$$

Die folgenden Schritte müssen durchgeführt werden, wenn die Bedingung erfüllt wird:

1. alle Referenzen der Variable merken
2. Variable in die neue Klasse verschieben
3. in der neuen Klasse die Referenzen der Variable in lokale Referenzen ändern, d.h. Namespaces und Instanznamen in der Referenz entfernen
4. für alle anderen Referenzen prüfen, ob die entsprechende Klasse bereits eine Instanz der neuen Klasse der Variable besitzt, sonst Instanz hinzufügen
5. alle Referenzen auf Instanz der neuen Klasse der Variablen umbiegen

Sollte keine andere Klasse gefunden werden, die diese Bedingung erfüllt, kann die Variable nicht verschoben werden und verbleibt in ihrer ursprünglichen Klasse.

Entfernen von Methoden

Eine Methode m, die weder interne Referenzen hat, noch von einer anderen Klasse G' genutzt wird, kann entfernt werden.

$$|R_m^{X'}| = 0$$

Da die Methode nicht verwendet wird, umfasst das Vorgehen hierbei auch nur das Löschen der Methode.

Entfernen von Properties

Analog zum Refactoring *Entfernen von Methoden* kann auch ein Property p, welches nicht verwendet wird, entfernt werden.

$$|R_p^{X'}| = 0$$

Entfernen von Variablen

Auch eine Variable a, die nicht verwendet wird, kann analog zu den beiden vorherigen Refactorings entfernt werden.

$$|R_a^{X'}| = 0$$

Nicht umgesetzte Operationen

Im Rahmen dieser Arbeit wurde sich dazu entschieden das Prinzip *Komposition an Stelle von Vererbung* anzuwenden. Darum wurden keine Refactorings eingeführt, die Vererbungshierarchien aufbauen oder manipulieren. Ein weiterer Grund ist, dass es je nach Sprachen deutliche Unterschiede im Bezug auf Tiefe von Vererbungshierarchien und Mehrfachvererbung gibt.

3.5.3 Spezialfälle

Es gibt einige Spezialfälle von Klassen, die aufgrund ihrer Aufgabe und ihres Designs nur sehr niedrige Kohäsion haben und anfällig für Code-Smells sind. Das sind beispielsweise

- Utility-Klassen, die viele Methoden haben, die meist nur wenig miteinander verknüpft sind, aber in der Regel keine Variablen

- Daten-Klassen, die normalerweise viele Variablen und eventuell Properties, aber normalerweise keine Methoden haben

3.6 Zusätzliche Einschränkungen

Zusätzlich zu den Bedingungen, die durch die Struktur einer Klasse erfüllt werden müssen, gibt es noch weitere Bedingungen, die Auswirkungen auf die Durchführbarkeit von Refactorings haben. Dies können beispielsweise

- Strukturen auf Modul- oder System-Ebene
- Durchführung zusätzlicher Analysen, etwa Datenflussanalyse
- sprachspezifische Eigenheiten

sein und müssen mit den hier aufgestellten Bedingungen verknüpft werden.
Davon betroffen sind beispielsweise Konstruktoren. Diese dürfen nicht verschoben werden. Auch ein Entfernen von Konstruktoren ist abhängig von der Sprache nur möglich, wenn es sich um einen leeren Default-Konstruktor handelt. Bezüglich des Aufteilen von Klassen muss geprüft werden, ob der Konstruktor Teil eines Clusters ist. In diesem Fall kann mittels Slice-basierter Kohäsion geprüft werden, ob es sich um ein sogenanntes Glue-Token [ZXZY02] handelt und infolgedessen die Funktionalität auf Konstruktoren der mit ihm verbundenen Cluster aufgeteilt werden.
Eine weitere Einschränkung stellen Klassen dar, die Interfaces implementieren. Von diesen dürfen keine Methoden verschoben oder gelöscht werden.
Bei abgeleiteten Klassen verhält es sich ähnlich wie bei Interfaces. Nur müssen zusätzlich noch bei Methoden, welche die Funktionalität erweitern, bestimmte Schlüsselwörter, wie SUPER oder BASE, beachtet werden. Solche Methoden dürfen auch nicht mit Clustern in neue Klassen verschoben werden, da dann der Kontext für das Schlüsselwort falsch ist.
Auch Variablen, die Pointer oder Referenzen auf den Typ der eigenen Klasse, ein Interface oder eine Oberklasse können das Aufteilen der Klasse verhindern. Hier muss geprüft werden, inwiefern der Typ der Variable geändert werden kann, damit die Variable in eine der neuen Klassen verschoben werden kann, ohne dass Funktionalität verlorengeht.

3.7 Auswirkung auf andere Metriken

In diesem Abschnitt wird noch ein kurzer Einblick gegeben, welche Auswirkungen auf andere Klassenmetriken zu erwarten sind. Dabei handelt es sich um Abschät-

zungen basierend auf den strukturellen Informationen der Klasse und den von den Refactorings betroffenen Bausteinen.

3.7.1 Kopplung

Für die Kopplung [Dav77], die hier die Abhängigkeit einer Klasse mit anderen Klassen [HCN98] beschreibt, lassen sich für das Refactoring *Klasse aufteilen* folgende Aussagen treffen:

- die Kopplung der neuen Cluster-Klassen entspricht zusammen dem der ursprünglichen Klasse, da nur die Funktionalität auf die neuen Klassen aufgeteilt wurde, aber nichts hinzugefügt wurde, was nicht in der Ursprungsklasse war
- für die Ursprungsklasse gilt, dass die Signaturen der Methoden nicht verändert wurden, nur die Variablen wurden durch Instanzen der neuen Klassen ersetzt

Damit lässt sich die Veränderung der Kopplung durch

$$\text{Anzahl der Cluster-Klassen} - \text{Kopplung vor dem Refactoring}$$

abschätzen.
Für die Refactorings, die Methoden betreffen, gilt, dass nur die betroffene Methode selber wegfällt. Das heißt, dass sich die Kopplung durch das Refactoring nicht verschlechtern kann und im besten Fall um die Anzahl der Parameter der Methode besser wird.
Bei den Variablen betreffenden Refactorings kommt es darauf an, um was für eine Variable es sich handelt. Bei einen elementaren Datentyp ändert sich die Kopplung nicht, sonst wird sie besser.
Das Entfernen eines Properties sollte im Allgemeinen keinen Einfluss auf die Kopplung haben.

3.7.2 Klassenkomplexität

Für die Klassenkomplexität kann die Metrik *Weighted Methods Per Class* (WMC) [MC05] verwendet werden. Diese stützt sich auf die zyklomatische Komplexität von Methoden [McC76].
Für das Aufteilen von Klassen ergibt sich dabei für die Cluster-Klassen, dass die WMC gemittelt über die Cluster-Klassen identisch zur WMC der ursprünglichen Klasse vor dem Refactoring sein muss. Für die einzelnen Cluster-Klassen kann keine Aussage getroffen werden, da es hier auf die Methoden ankommt, die zum jeweiligen Cluster gehört haben. Bei der ursprünglichen Klasse hat jede Methode nach dem Refactoring einen Komplexitätswert von 1, da sie nur noch den einen Aufruf der

verschobenen Methode enthält[1]. Da die zyklomatische Komplexität nicht niedriger als 1 sein kann und sich die Anzahl der Methoden nicht geändert hat, heißt das, dass die WMC vor dem Refactoring nicht niedriger beziehungsweise besser gewesen sein kann.
Für die Methoden betreffenden anderen Refactorings wird die WMC niedriger, da es nach dem Refactoring eine Methode weniger gibt, die eine zyklomatische Komplexität von mindestens 1 hat. Die übrigen Methoden der Klasse wurden nicht verändert, also muss die WMC um mindestens 1 niedriger sein.
Die restlichen Refactorings haben keinen Einfluss auf die Komplexität.

[1]Die zyklomatisch Komplexität nach McCabe ist definiert als M = Anzahl der binären Verzweigungen des Kontrollflussgraphen + 1. Der Kontrollflussgraph einer Methode mit nur einer Anweisung enthält keine binäre Verzweigung

4 Implementierung

In diesem Kapitel wird auf die Umsetzung des Konzepts als Plugin für CODESYS eingegangen. Das Plugin ist mit Visual Studio 2015 in C# geschrieben und gegen das SDK von CODESYS SP16 gebaut. Zusätzlich wird Newtonsoft.Json für das Erfassen und Speichern der Messdaten verwendet. Installiert wird das Plugin mittels des IPM und kann danach als normales Menükommando verwendet werden.
Die Funktionalität des Plugin basiert dabei auf dem Kontext des Precompiles eines geladenen Projekts.

4.1 Umsetzung der Refactorings

Grundsätzlich lassen sich alle im vorherigen Kapitel vorgestellten Refactorings auch für CODESYS implementieren. Allerdings muss aufgrund von Eigenheiten des Sparchstandards an ein paar Stellen von der Schritten zur Durchführung des jeweiligen Refactorings leicht abgwichen werden oder es sind Ergänzungen nötig.
Die größte Änderung betrifft dabei das Refactoring *Aufteilen von Klassen*. Aufgrund der in Unterabschnitt 2.4.1 erläuterten Tatsache, dass Funktionsbausteine auch selber aufgerufen werden können, müssen deshalb im Implementierungsteil des ursprünglichen FBs alle neu angelegten Instanzen der Cluster-FBs einmal aufgerufen werden. Exemplarisch ist dies im folgenden Code-Fragment zu sehen.

```
// Beispiel für Implementierungsteil des FBs
// alle Instanzen der erzeugten Cluster_Fbs müssen aufgerufen werden
  instance_of_cluster_1(var_in_1 := var_in_1, var_out_1 := var_out_1);
  instance_of_cluster_1();
```

Hierbei kann zwischen zwei Fällen unterschieden werden. Entweder die Hauptmethode eines Cluster-FBs enthält die Funktionalität der Hauptmethode des ursprünglichen Funktionsbausteins, dann wird der Aufruf auch an die Instanz des Cluster-FBs durchgereicht und der Code mit einer Indirektion ausgeführt. Oder die Hauptmethode des Cluster-FBs ist leer, dann wird beim Aufruf der Instanz eine leere Methode aufgerufen und der Aufruf hat keinen Effekt. Somit gibt es keinen negative Auswirkungen auf die Funktionalität des FBs.
Außerdem kann gesehen werden, dass auch die Input- und Output-Variablen des ursprünglichen FBs weiterhin verwendet werden müssen. Darum dürfen diese beim

Refactoring nur in einen neuen Funktionsbaustein kopiert werden, nicht aber im ursprünglichen FB entfernt werden. Ein weiterer Grund, warum zumindest ein Teil der Instanz der Cluster-FBs aufgerufen werden muss, sind die Input-Variablen verbunden mit der zyklischen Arbeitsweise von Applikationen.

Auch beim Verschieben von Methoden gibt es eine Einschränkung. Die Information, welche Variable in welchen Bausteinen verwendet wird, hängt im Precompile an der jeweiligen Variablen.

5 Evaluierung

5.1 Visualisierung

In diesem Abschnitt werden die unterschiedlichen Visualisierungsalgorithmen gegenübergestellt. Für alle drei folgenden Grafiken wurden die selben Berechnungsmodelle verwendet und alle zeigen die selbe Heap-Implementierung. Abbildung 5.1 zeigt den Spring-Embedder-Algorithmus. In Abbildung 5.2 wird der Fruchterman-Reingold-Algorithmus dargestellt und in Abbildung 5.3 ist der modifizierte Fruchterman-Reingold-Algorithmus zu sehen. Man kann deutlich erkennen, dass beim Spring-Embedder die Knoten deutlich enger zusammenliegen, während beim modifizierten Fruchterman-Reingold-Algorithmus die Struktur des Zusammenhangsgraphen deutlich hervortritt.

5.2 Datenbasis

Wie bereits in Unterabschnitt 2.4.1 erwähnt ist Objektorientierung in der SPS-Programmierung noch nicht so verbreitet. Aus diesem Grund ist auch der Bestand an Projekten mit denen der Effekt des Konzepts dieser Arbeit gemessen werden relativ gering. Deshalb wurde auf eine größere CODESYS-Bibliothek zurückgegriffen, die bereits objektorientiert programmiert ist.

	FBs	Methoden	Properties	Variablen
Gesamt	264	2450	153	1306
Datenrefs mit Refactoring	96	1425	68	646
Datenrefs nach Refactoring	148	1970	109	890
Alle Refs mit Refactoring	94	1419	68	642
Alle Refs nach Refactoring	162	2134	110	916

Tabelle 5.1: Datenbasis der Evaluation

Wie Tabelle 5.1 zu entnehmen ist, hat die Bibliothek 264 FBs mit 2450 Methoden, 153 Properties und 1306 Variablen. Dabei konnte mittels der Kohäsion über die Datenreferenzen in 96 FBs ein Code-Smell gefunden werden. Bei der Berechnung

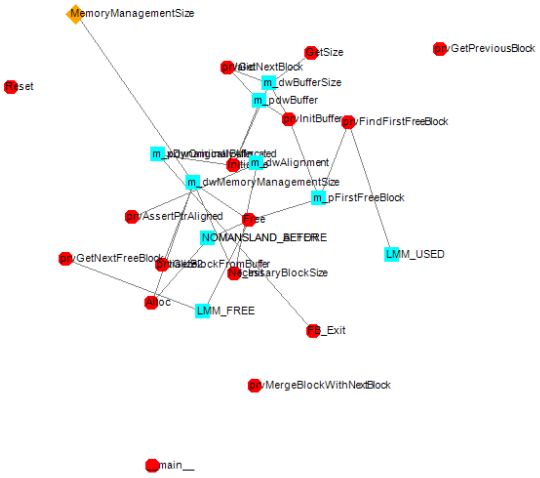

Abbildung 5.1: Die Grafik zeigt den Zusammenhangsgraphen einer Heap-Implementierung. Die Positionen der Knoten wurde mittels des SpringEmbedder-Algorithmus berechnet.

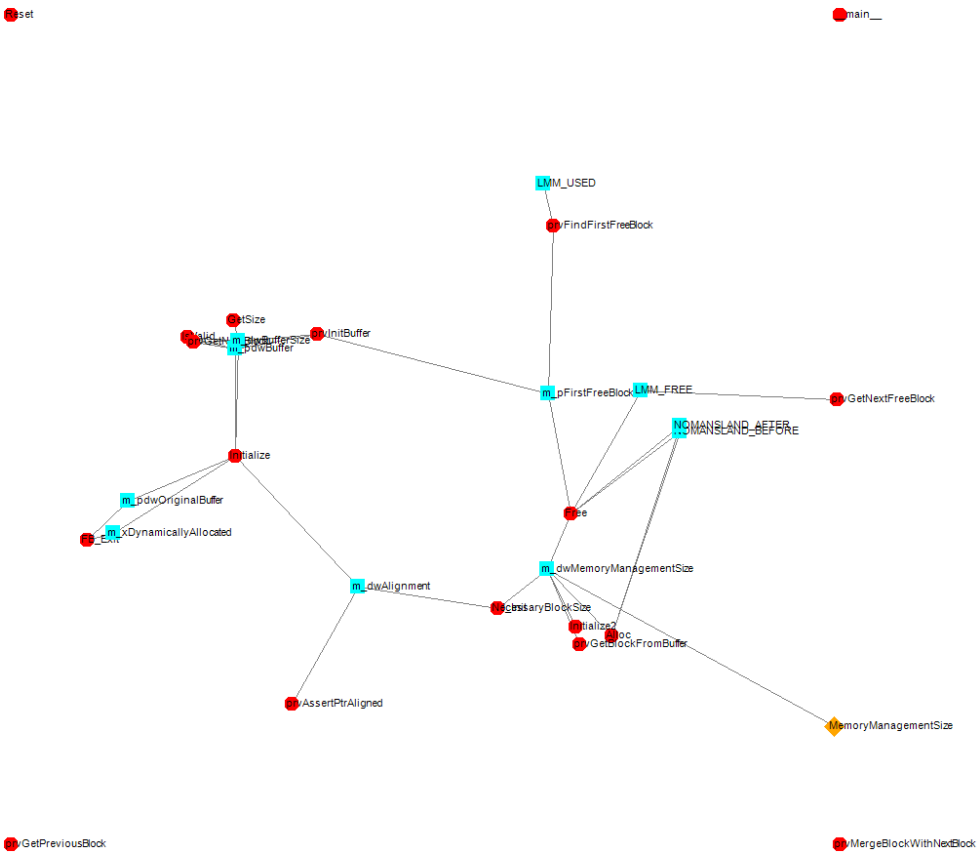

Abbildung 5.2: Die Grafik zeigt den Zusammenhangsgraphen einer Heap-Implementierung. Die Positionen der Knoten wurde mittels des Fruchterman-Reingold-Algorithmus berechnet.

Abbildung 5.3: Die Grafik zeigt den Zusammenhangsgraphen einer Heap-Implementierung. Die Positionen der Knoten wurde mittels des modifizierten Fruchterman-Reingold-Algorithmus berechnet.

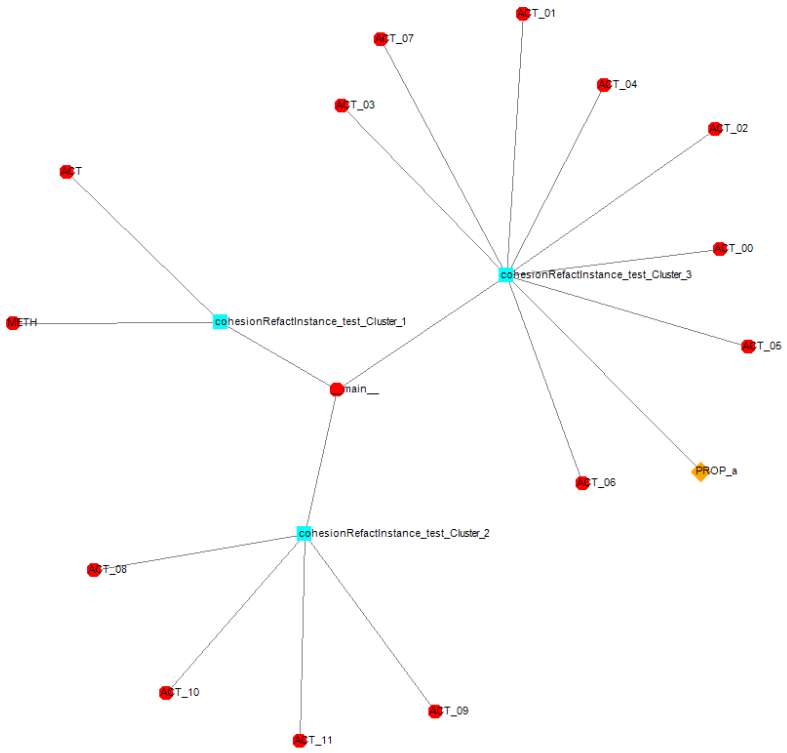

Abbildung 5.4: Die Grafik zeigt den Zusammenhangsgraphen eines FBs nach allen Refactorings. Der Abstand der Methoden-Knoten (rot) wurde dabei mittels des Berechnungsmodells gewichtete direkte Referenzen mit alle Knoten als Basis berechnet.

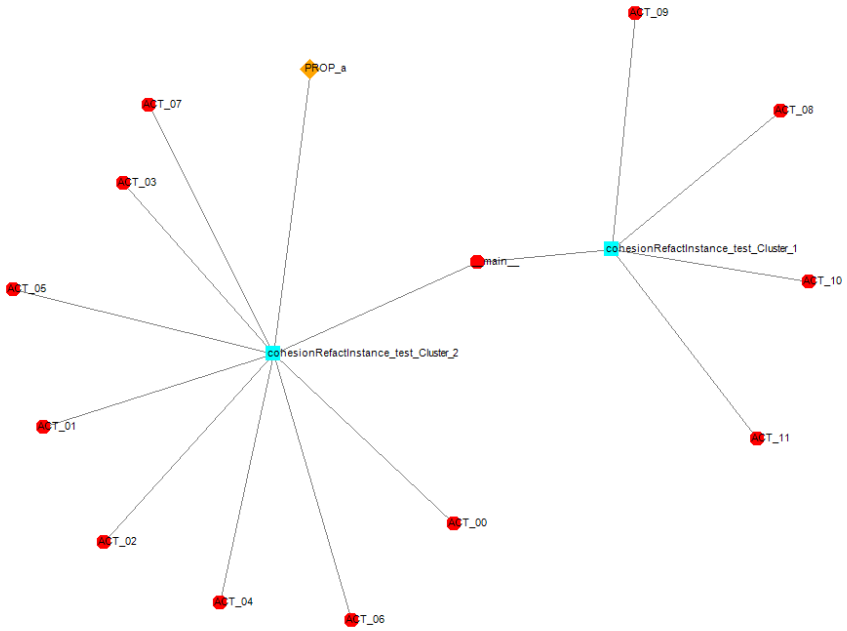

Abbildung 5.5: Die Grafik zeigt den Zusammenhangsgraphen eines FBs nach allen Refactorings. Der Abstand der Methoden-Knoten (rot) wurde dabei mittels des Berechnungsmodells gewichtete direkte Referenzen mit den Variablen-Knoten als Basis berechnet.

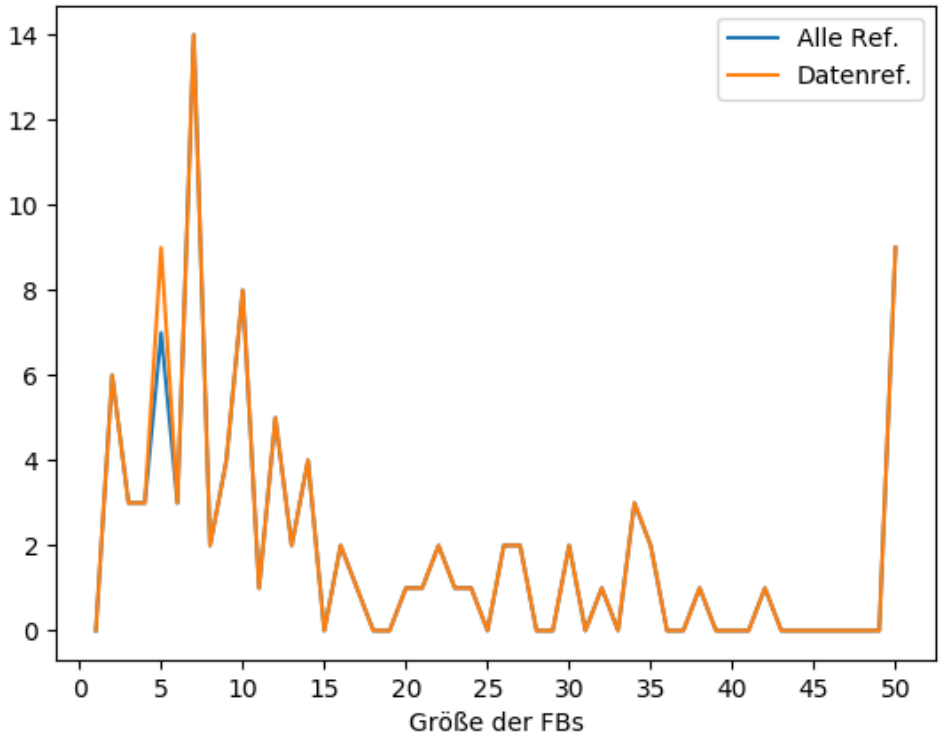

Abbildung 5.6: Die Grafik zeigt die Verteilung der FBs bezüglich ihrer Größe.

über alle Referenzen konnte nur in 94 FBs ein Code-Smell gefunden werden.
?? zeigt dabei, wie sich die FBs bezüglich ihrer Größe verteilen.

5.3 Refactoring und Kohäsion

Abbildung 5.7 zeigt die Anzahl der einzelnen Refactorings für alle Referenzen und Datenreferenzen. Zu erwarten waren, dass Properties nicht betroffen sind, das diese in der Regel immer an einer Variablen hängen, als auch, dass keine Variablen verschoben wurden. Dies sollte aus Gründen der Datenkapselung fast nie passieren. Die Verteilung bei den die Methoden betreffenden Operationen ist überraschend. Das Löschen sollte bei beiden Berechnungsmodellen gleich verteilt sein und die Anzahl der in Utility-Funktionen umgewandelten Methoden sollte bei den Datenreferenzen deutlich höher sein. Allerdings haben sich die Ergebnisse auch bei nochmaligem Messen bestätigt.

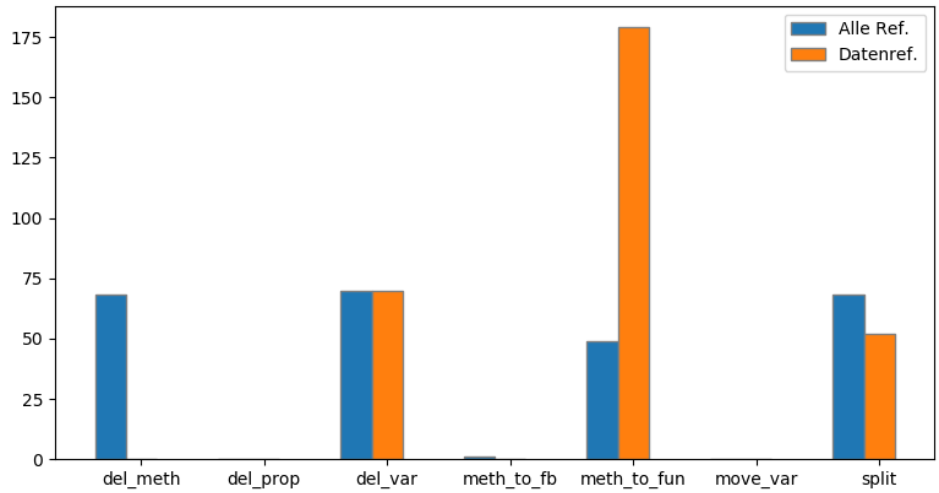

Abbildung 5.7: Die Grafik zeigt die Verteilung der einzelnen Refactorings für die Berechnungsmodelle. Die blauen Balken stehen dabei für die Refactorings basierend auf allen Referenzen innerhalb eines FBs. Die orangenen Balken stehen für Refactorings basierend auf Datenreferenzen. Während sich bei den Refactorings Property entfernen, Methode in anderen FB verschieben und Variable verschieben beide Metriken zu fast gleichen Ergebnissen führen, sind bei den übrigen Operationen teils deutliche Unterschiede zu sehen.

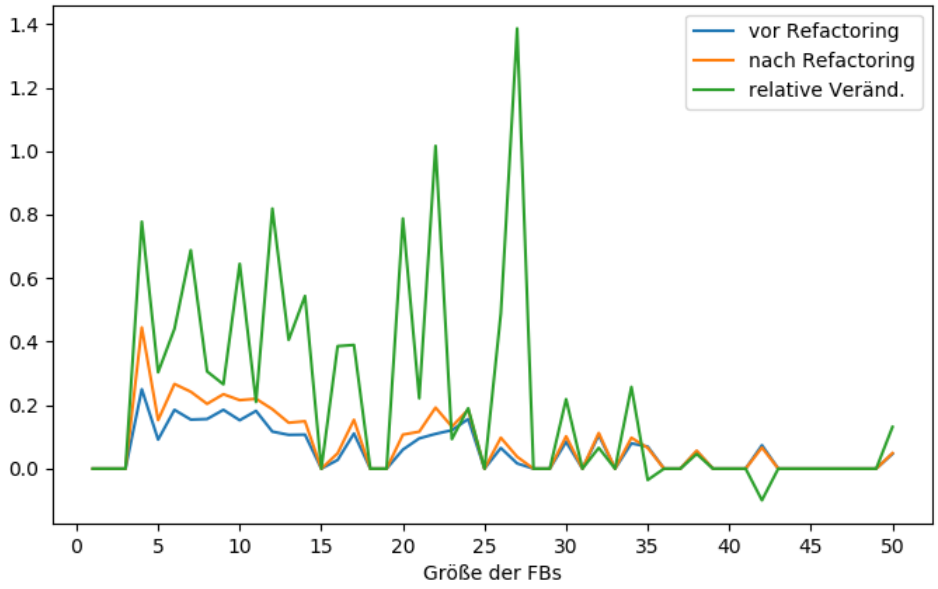

Abbildung 5.8: Die Grafik zeigt die Kohäsion basierend auf allen Referenzen vor dem Refactoring (blau), nach dem Refactoring (gelb) und die relative Veränderung (grün).

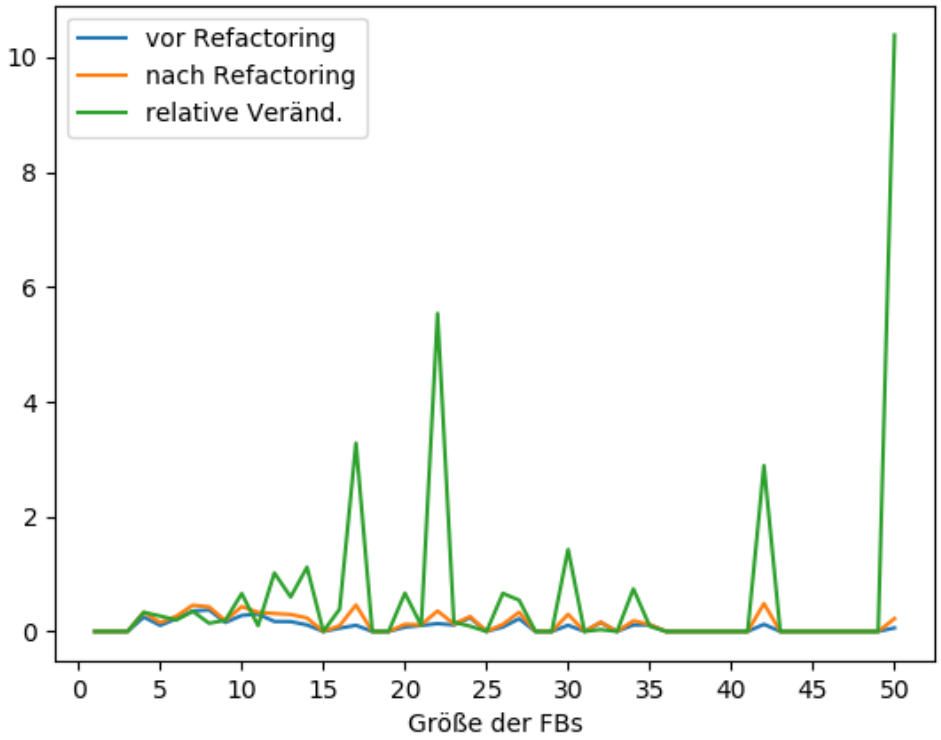

Abbildung 5.9: Die Grafik zeigt die Kohäsion basierend auf den Variablenreferenzen vor dem Refactoring (blau), nach dem Refactoring (gelb) und die relative Veränderung (grün).

Abbildung 5.8, die die Kohäsion für die jeweilige FB-Größe auf Basis aller Referenzen darstellt und Abbildung 5.9, die die Kohäsion basierend auf den Datenreferenzen zeigt, zeigen bis auf eine kleine Verschlechterung bei Abbildung 5.8 fast durchwegs eine deutliche realtive Steigerung der Kohäsion. Die eine Verschlechterung kommt von einem FB, der sehr viele Methoden hat, die keine Variablenreferenzen besitzen. Da diese bei der Daten-basierten Berechnung eine Teilkohäsion von 0 aufweisen, wurden alle in Utility-Funktionen umgewandelt.
Grundsätzlich beträgt die mittlere relative Steigerung der Kohäsion über alle FBs für die alle Referenzen berücksichtigende Berechnung über 20% und für die nur Variablenreferenzen berücksichtigende Metrik 60% und ohne den Ausreißer bei einer FB-Größe von 50 immer noch über 40%.

5.4 Weitere Beobachtungen

Die Kohäsion sollte nicht ausschließlich auf Basis der Methodenreferenzen berechnet werden, da dies bei Variablen, welche nur Referenzen zu anderen Variablen haben, dass diese entfernt werden.

6 Zusammenfassung und Ausblick

Es wurden mit den gewichteten Referenzen der Knoten des Zusammenhangsgraphen und der Hamming-Distanz zwei Metriken ausgewählt, die sich hinsichtlich ihres Verwendungszwecks gut ergänzen. Die gewichteten Referenzen für die Gewichtung der Knoten und beim Finden von Code-Smells, die Hamming-Distanz bei der Visualisierung der Kohäsion als Kantengewicht. Bei der Visualisierung konnte durch eine Modifizierung des Fruchterman-Reingold-Algorithmus die Struktur des Zusammenhangsgraphen klar ersichtlich dargestellt werden. Für das Refactoring wurden zwei Code-Smells herausgearbeitet und formalisiert. Auf Basis dieser kann unter Berücksichtigung weiterer Bedingungen eines von sieben Refactorings ausgewählt werden. Grundsätzlich muss aber festgestellt werden, dass die Kohäsion nur zusammen mit anderen Indikatoren, wie hier dem Zusammenhangsgraph, dazu geeignet ist Code-Smells zu finden und Refactorings zu ermöglichen. Die Umsetzung mittels CODESYS hat zudem gezeigt, dass sich die Refactorings nach Bedarf automatisch ausführen lassen. Bei der Evaluation des Konzepts konnte für die Berechnung auf Basis der Variablenreferenzen eine durchschnittliche relative Steigerung von über 40% und für die Berechnung auf Basis aller Referenzen eine relative Steigerung um duschnittlich über 20% festgestellt werden.

Bezüglich der Metrik wäre für die Zukunft zu prüfen, wie sich das Einbeziehen indirekter Referenzen auf den Wert der Kohäsion auswirkt. Zudem könnte eine Datenflussanalyse als zusätzliche Unterstützung helfen. Dann könnten auch weitere Refactorings, wie etwa das Aufteilen einer Klasse die nicht aus zwei komplett unabhängigen Funktionalitäten besteht, umgesetzt werden. Eine weitere Möglichkeit für ein Refactoring ist zu prüfen, inwiefern ein Interface refaktoriert werden kann, indem man beispielsweise die Kohäsion aller es implementierender Interfaces betrachtet.

Abbildungsverzeichnis

Tabellenverzeichnis

Definitions- und Theoremverzeichnis

Abkürzungsverzeichnis

AS	Ablaufsprache, engl. Sequential Function Chart
FB	Funktionsbaustein, engl. Functionblock
FRA	Fruchtman-Reingold-Algorithm, engl. Fruchtman-Reingold-Algorithm
FUP	Funktionsbausteinsprache, engl. Functionblock
KOP	Kontaktplan, engl. Ladder
ST	Strukturierter Text, engl. Structured Text

Literaturverzeichnis

[AAM19] Alzahrani, Musaad ; Alqithami, Saad ; Melton, Austin: Using Client-Based Class Cohesion Metrics to Predict Class Maintainability. In: *43rd IEEE Annual Computer Software and Applications Conference, COMPSAC 2019, Milwaukee, WI, USA, July 15-19, 2019, Volume 1*, IEEE, 2019, 72–80

[AC94] Abreu, Fernando B. ; Carapuça, Rogério: Object-Oriented Software Engineering: Measuring and Controlling the Development Process. (1994), 10

[BK95] Bieman, James M. ; Kang, Byung-Kyoo: Cohesion and Reuse in an Object-Oriented System. In: Samadzadeh, M. H. (Hrsg.) ; Kazerooni-Zand, Mansour (Hrsg.): *Proceedings of the ACM SIGSOFT Symposium on Software Reusability, SSR@ICSE 1995, April 23-30, 1995, Seattle, WA, USA*, ACM, 1995, 259–262

[BO94] Bieman, J.M. ; Ott, L.M.: Measuring Functional Cohesion. In: *IEEE Transactions on Software Engineering* 20 (1994), aug, Nr. 08, S. 644–657. http://dx.doi.org/10.1109/32.310673. – DOI 10.1109/32.310673. – ISSN 1939–3520

[CK91] Chidamber, Shyam R. ; Kemerer, Chris F.: Towards a Metrics Suite for Object Oriented Design. In: Paepcke, Andreas (Hrsg.): *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91), Sixth Annual Conference, Phoenix, Arizona, USA, October 6-11, 1991, Proceedings*, ACM, 1991, 197–211

[CTC20] Cyr, André ; Thériault, Frédéric ; Chartier, Sylvain: Revisiting the XOR problem: a neurorobotic implementation. In: *Neural Comput. Appl.* 32 (2020), Nr. 14, 9965–9973. http://dx.doi.org/10.1007/s00521-019-04522-0. – DOI 10.1007/s00521–019–04522–0

[Dav77] Davis, Charles H.: Reliable Software through Composite Design. Glenford J. Myers. New York: Petrocelli-Charter, 159p. (1975) (ISBN: 0-88405-284-2). In: *J. Am. Soc. Inf. Sci.* 28 (1977), Nr. 5, 303. http://dx.doi.org/10.1002/asi.4630280511. – DOI 10.1002/asi.4630280511

[Ead84] Eades, Peter: A heuristic for graph drawing. In: *Congressus Numerantium* (1984), S. 149–160

[Fen91] Fenton, Norman E.: *Software metrics - a rigorous approach*. Chapman and Hall, 1991

[Fow20] Fowler, Martin: *Refactoring Wie sie das Design bestehender Software verbessern*. 2. Auflage. mitp Verlag, 2020

[FR91] Fruchterman, Thomas M. J. ; Reingold, Edward M.: Graph Drawing by Force-directed Placement. In: *Softw. Pract. Exp.* 21 (1991), Nr. 11, 1129–1164. http://dx.doi.org/10.1002/spe.4380211102. – DOI 10.1002/spe.4380211102

[GHJV11] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. 6. Auflage. Addison-Wesley, 2011

[GK00] Gajer, Pawel ; Kobourov, Stephen G.: GRIP: Graph dRawing with Intelligent Placement. In: Marks, Joe (Hrsg.): *Graph Drawing, 8th International Symposium, GD 2000, Colonial Williamsburg, VA, USA, September 20-23, 2000, Proceedings* Bd. 1984, Springer, 2000 (Lecture Notes in Computer Science), 222–228

[GM18] Gleich, Ronal ; Munck, Jan C.: *Die richtigen Kennzahlen optimal nutzen*. 1. Auflage. Haufe Verlag, 2018

[Gmb16] GmbH, CODESYS: *Objektorientiert programmieren in der SPS?* https://www.sps-magazin.de/allgemein/objektorientiert-programmieren-in-der-sps, 2016. – [Online; Zugriff am 23.02.2021]

[Gri91] Griswold, William G.: *Program Restructuring as an Aid to Software Maintenance*, University of Washington, Dissertation, 1991

[HCN98] Harrison, Rachel ; Counsell, Steve ; Nithi, Reuben V.: Coupling Metrics for Object-Oriented Design. In: *5th IEEE International Software Metrics Symposium (METRICS 1998), March 20-21, 1998, Bethesda, Maryland, USA*, IEEE Computer Society, 1998, 150–157

[IEE92] IEEE Standard for a Software Quality Metrics Methodology. In: *IEEE Std 1061-1992* (1992), S. 1–96. http://dx.doi.org/10.1109/IEEESTD.1993.115124. – DOI 10.1109/IEEESTD.1993.115124

[KC94] Kemerer, C. ; Chidamber, S.: A Metrics Suite for Object Oriented Design. In: *IEEE Transactions on Software Engineering* 20 (1994), jun, Nr. 06, S. 476–493. http://dx.doi.org/10.1109/32.295895. – DOI 10.1109/32.295895. – ISSN 1939–3520

[Mar09] Martin, Robert C.: *Clean Code: Refactoring, Patterns, Testen und Techniken für sauberen Code*. 1. Auflage. mitp Verlag, 2009

[MC05] Michura, John ; Capretz, Miriam A. M.: Metrics Suite for Class Complexity. In: *International Symposium on Information Technology: Coding and Computing (ITCC 2005), Volume 2, 4-6 April 2005, Las Vegas, Nevada, USA*, IEEE Computer Society, 2005, 404–409

[McC76] McCabe, Thomas J.: A Complexity Measure. In: *IEEE Trans. Software Eng.* 2 (1976), Nr. 4, 308–320. http://dx.doi.org/10.1109/TSE.1976.233837. – DOI 10.1109/TSE.1976.233837

[McE20] McEnery, Sage: *How much computer code has been written?* https://medium.com/modern-stack/how-much-computer-code-has-been-written-c8c03100f459, 2020. – [Online; Zugriff am 26.02.2021]

[Opd92] Opdyke, William F.: *Refactoring Object-Oriented Frameworks*, University of Illinois, Dissertation, 1992

[PM05] Poshyvanyk, D. ; Marcus, A.: The Conceptual Cohesion of Classes. In: *Proceedings of the 21st IEEE International Conference on Software Maintenance*. Los Alamitos, CA, USA : IEEE Computer Society, sep 2005. – ISSN 1063–6773, 133-142

[SLS01] Simon, F. ; Lewerentz, C. ; Steinbrückner, F.: Metrics Based Refactoring. In: *2001 5th European Conference on Software Maintenance and Reengineering*. Los Alamitos, CA, USA : IEEE Computer Society, mar 2001. – ISSN 1534–5351, 30

[Tut63] Tutte, William T.: How to draw a graph. In: *IEEE Transactions on Software Engineering* (1963), S. 743–768

[WDB97] Wuest, J. ; Daly, J. W. ; Briand, L. C.: A Unified Framework for Cohesion Measurement. In: *Software Metrics, IEEE International Symposium on*. Los Alamitos, CA, USA : IEEE Computer Society, nov 1997, 43

[Wol74] Wolverton, Ray W.: The Cost of Developing Large-Scale Software. In: *IEEE Trans. Computers* 23 (1974), Nr. 6, 615–636. http://dx.doi.org/10.1109/T-C.1974.224002. – DOI 10.1109/T–C.1974.224002

[WZ16] Wellenreuther, Günter ; Zastrow, Dieter: Grundzüge der SPS-Norm IEC 61131-3. In: Plassmann, Wilfried (Hrsg.): *Handbuch Elektrotechnik: Grundlagen und Anwendungen für Elektrotechniker*. Wiesbaden : Springer Fachmedien Wiesbaden, 2016. – ISBN 978–3–658–07049–6, 779–785

[Zus98] Zuse, Horst: *A Framework of Software Measurement*. 1. Auflage. Walter de Gruyter, 1998

[ZXZY02] Zhou, Yuming ; Xu, Baowen ; Zhao, Jianjun ; Yang, Hongji: ICBMC: An Improved Cohesion Measure for Classes. In: *18th International Conference on Software Maintenance (ICSM 2002), Maintaining Distributed Heterogeneous Systems, 3-6 October 2002, Montreal, Quebec, Canada*, IEEE Computer Society, 2002, 44–53