UNIVERSITÄT OSNABRÜCK

Institute of Computer Science, Software Engineering Group

Bachelorarbeit

Die automatisierte Bewertung von Data-Clumps mit Hilfe von Tracing-Daten

Lucas Nagelsmann
Immatrikulationsnummer: 973116

04.06.2024
Erstbetreuerin: Prof. Dr.-Ing. Elke Pulvermüller
Zweitbetreuer: Dennis Ziegenhagen, M.Sc.

Abstract

Deutsch Im Rahmen dieser Bachelorarbeit wird ein Konzept zur automatisierten Bewertung von Data-Clumps entwickelt. Aufbauend auf den Ergebnissen des Data-Clump Detektors von Nils Baumgartner sollen die erhaltenen Data-Clumps eines Projekts anhand verschiedener Kriterien automatisch bewertet werden. Diese Bewertung basiert sowohl auf zugehörigen Tracing-Daten als auch auf dem Source-Code des Beispiel-Projekts *GanttProject* selbst.
Das Ziel dieser Arbeit ist es, eine automatisierte Bewertung für jeden Data-Clump zu schaffen, damit diese nach Relevanz in eine Prioritätenliste eingeordnet und für das anschließende Refactoring übergeben werden können. Mit Hilfe dieser Lösung sollen sowohl Zeit als auch Ressourcen für ein manuelles Sortieren und Prüfen der Data-Clumps eingespart sowie die generelle Möglichkeit einer Bewertung aufgezeigt werden.

English For this bachelor thesis a concept for the automated evaluation of data clumps is developed. Based on the results of Nils Baumgartner's data clump detector, the data clumps received from a project are to be evaluated automatically using various criteria. This evaluation is based both on associated tracing data and on the source code of the example project *GanttProject* itself.
The aim of this work is to create an automated evaluation for each data clump so that they can be prioritized according to relevance and transferred for subsequent refactoring, saving time and resources for manual sorting and checking of the data clumps.

Inhaltsverzeichnis

1 Einleitung

Diese Bachelorarbeit thematisiert die automatisierte Bewertung von Data-Clumps. Dabei sollen die durch einen von Nils Baumgartner an der Universität Osnabrück entwickelten Data-Clump Detektor [1] gefundenen Data-Clumps anhand von verschiedenen Kriterien einzeln bewertet werden. Als Basis der Bewertung dienen sowohl diverse Tracing-Daten als auch der Quellcode des Open-Source-Projekts *GanttProject* [2] als Beispiel. Bisher mussten die einzelnen Data-Clumps manuell betrachtet und nach Relevanz sortiert werden, um eine sequentielle Verbesserung nach der Priorität dieser möglich zu machen. Mit Hilfe der Bewertung soll automatisch eine Prioritätenliste der Data-Clumps für ein späteres Refactoring festgelegt werden können.

Neben der Ausarbeitung des Konzepts werden konkrete Kriterien für eine automatische Bewertung umgesetzt sowie mögliche Schwierigkeiten und Hindernisse - nicht zuletzt auf Grund der jeweiligen Datenlage - durchleuchtet.

1.1 Motivation und Hintergrund

Der Hintergrund der Arbeit ist das notwendige Sortieren der durch den Data-Clump-Detektor [1] gefundenen Data-Clumps, da nicht alle Data-Clumps gleich wichtig und ähnlich komplex sind. So können nicht alle Datenklumpen direkt zusammen beseitigt werden und einige können derart komplex sein, dass eine Beseitigung dieser eine Änderung des ganzen Projekts zur Folge hätte. Der Bewertungsprozess und die Prüfung der Data-Clumps muss – Stand jetzt – manuell durchgeführt werden, bevor die Data-Clumps an das anschließende Refactoring übergeben werden können. Eine automatisierte Bewertung auf Basis der Tracing-Daten und des Source-Codes sollen somit den Aufwand und die Zeit einer solchen Überprüfung einsparen. Tracing-Daten eignen sich besonders gut für eine solche Einschätzung, da diese während des Softwareentwicklungsprozesses oftmals bereits existieren und beispielsweise für Anforderungen des Projekts sogar benötigt werden. Darüber hinaus können durch Tracing-Daten auch Architekturen ersichtlich werden, auf Grund derer direkt auf die Verbreitung sowie Verbindung einzelner Module geschlossen werden kann. Des Weiteren kann mit der Bewertung direkt überprüft werden, ob eine Verbesserung für den jeweiligen Data-Clump technisch überhaupt möglich und sinnvoll ist. Damit einhergehend entsteht ebenfalls eine grobe Aufwandsschätzung, welche für eine Prüfung der Wirtschaftlichkeit einer Data-Clump-Änderung nützlich sein kann.

1.2 Ausgangssituation

Dabei soll die hier entwickelte automatisierte Bewertung von Data-Clumps Teil einer automatisierten Pipeline sein, bei der bereits erkannte Datenklumpen durch einen Data-Clump-Detektor [1] [3] im *JavaScript Object Notation* (JSON)-Format an das Programm übergeben werden. Durch weitere Eingaben, wie Tracing-Daten und des Source-Codes des Projekts, wird dann eine automatisierte Bewertung jedes einzelnen Data-Clumps getroffen und mit einem Wert zurückgegeben. Dieser Wert gibt die Wahrscheinlichkeit an, wie notwendig und vor allem wie wichtig ein Refactoring ist. Mit Hilfe dieses Wertes kann das nachfolgende Refactoring starten, welches den Quellcode - ebenfalls automatisiert - durch Entfernen der Data-Clumps verbessert. Das Refactoring ist nicht Teil der Arbeit. Der zu berechnende Wert gibt somit an, wie wichtig ein Refactoring für das Projekt ist. Von besonderer Bedeutung dabei ist, dass nur auf Grund der Relevanz nicht auf die Möglichkeit einer Beseitigung geschlossen werden kann. Ein Data-Clump kann folglich durch den ermittelten Wert als wichtig zu ändern angesehen werden, wobei noch keine direkte Abschätzung bezüglich der Umsetzung gegeben ist.

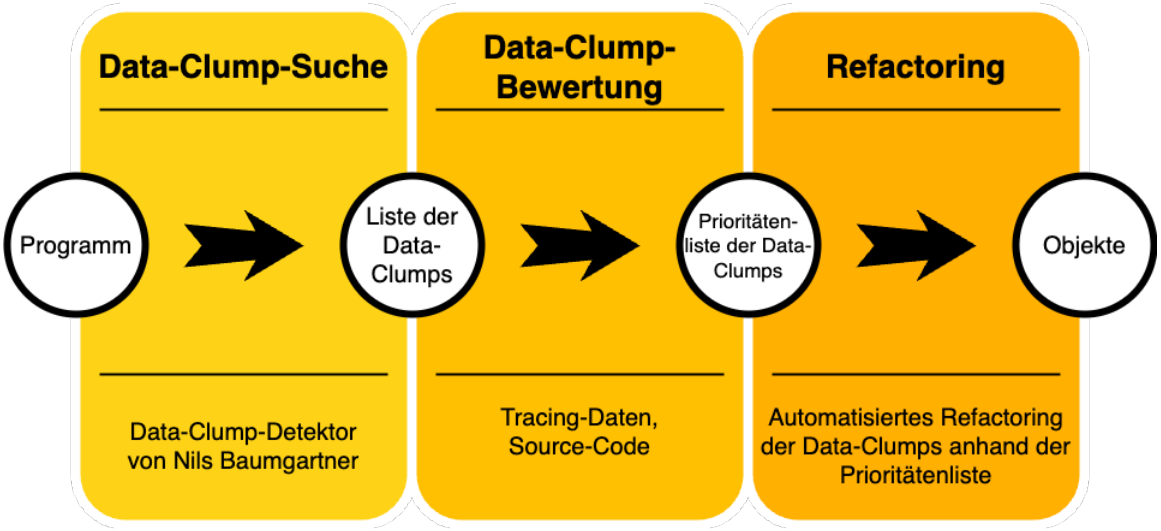


Abbildung 1: Die automatisierte Bewertung von Data-Clumps innerhalb der Pipeline

1.3 Anforderungen an die automatisierte Bewertung

Um die zeitintensive Prüfung der Data-Clumps zu reduzieren und die Integration in die automatisierte Pipeline zu gewährleisten, bestehen die Anforderungen des Programms aus drei Bereichen: die Eingabe, die Verarbeitung der Daten und die Ausgabe. Als Eingabe sollen sowohl die gefundenen Data-Clumps im JSON-Format als auch die zugehörigen Tracing-Daten dienen. Diese sollen für jeden Data-Clump Werte für die einzelnen Bewertungskriterien berechnen, welche sich darauffolgend zu einem Gesamtwert zusammensetzen. Dabei soll auch die Gewichtung der Kriterien innerhalb der Bewertung be-

rücksichtigt werden. Ausgegeben werden sollen diese Werte wahlweise entweder direkt im Programm visualisiert, über die Konsole oder in einer zusätzlichen Datei - ebenfalls im JSON-Format.

1.4 Aufbau der Arbeit

In Kapitel zwei dieser Bachelorarbeit werden Grundlagen erläutert und wichtige Begriffe wie die der Data-Clumps oder des Refactorings für die nachfolgende Arbeit definiert.

Im dritten Kapitel folgt der konzeptionelle Teil der Arbeit, welcher die Möglichkeiten einer automatisierten Bewertung thematisiert und auf Grundlage der Eigenschaften der Data-Clumps in Verbindung mit den Tracing-Daten mögliche Kriterien einer Bewertung entwickelt sowie abwägt, welche Kriterien für eine Umsetzung beispielsweise anhand der Datenlage ungeeignet sind.

Das vierte Kapitel begleitet die Umsetzung des Konzepts und zeigt Möglichkeiten, Techniken sowie besondere Hindernisse bei der Implementierung der automatisierten Bewertung.

Das fünfte und letzte Kapitel dieser Arbeit fasst die Ergebnisse zusammen und gibt einen kurzen Ausblick auf die nachfolgenden Arbeiten.

2 Grundlagen

Im Folgenden werden die Grundlagen der Arbeit erläutert und die Fragen beantwortet, was Data-Clumps und Refactoring sind und warum diese Begriffe in der Softwareentwicklung von Bedeutung sind. Des Weiteren werden die Tracing-Daten als Datenquelle der Bewertung der Data-Clumps beschrieben und es wird erörtert, inwiefern diese hilfreich für die Bewertung sein können.

2.1 Die Data-Clumps

Der Hauptbestandteil dieser Arbeit sind Data-Clumps, welche eine Unterkategorie von Code-Smells darstellen [4]. Code-Smells sind Verstöße gegen Design-Prinzipien [4]. Oftmals treten diese auf, wenn neue Funktionalitäten zu bestehendem Code hinzugefügt werden und somit die Komplexität des bereits existierenden Codes erhöht wird [5]. Ebenfalls beeinträchtigen Code-Smells die Entwicklung [6] [7], die Erweiterbarkeit, die Wartbarkeit sowie die Testbarkeit des Softwareprojekts [7] [8] [9]. Außerdem erschweren diese das Verständnis des Codes für weitere Entwickler, können bei folgender Arbeit zu Fehlern führen [10] und somit das Projekt sowohl zeitlich als auch finanziell stark beeinflussen [3]. Deshalb ist es besonders wichtig, Code-Smells frühzeitig zu erkennen und zu beseitigen. Zu den bekanntesten Code-Smells gehören unter anderem *God-Klassen*, *Lava Flow*, *Spaghetti-Code* und das *Schweizer-Taschenmesser* [4]. Auch die in dieser Arbeit behandelten Data-Clumps sind sehr verbreitet und ein Bestandteil *Fowlers* Liste der bekanntesten Code-Smells [4] [5]. Besonders im Kontext der Webentwicklung sind Data-Clumps die am zweithäufigsten auftretenden Design-Verstöße [11]. Data-Clumps sind eine Gruppe von Variablen, welche jeweils immer zusammen an andere Bereiche des Codes weitergereicht werden [4]. Dabei werden zwischen zwei Arten von Data-Clumps unterschieden: Field- und Parameter-Data-Clumps [7].

Field-Data-Clumps bestehen aus Feldern, welche Variablen innerhalb einer Klasse darstellen und in dieser direkt deklariert sind [12]. Sie umfassen mindestens drei Felder gleicher Signatur und müssen mindestens mit zwei Klassen geteilt werden, aber nicht in der gleichen Reihenfolge auftreten [13] [3]. Darüber hinaus müssen stets beide Richtungen der Ausprägung von Data-Clumps beachtet werden [14].

Parameter-Data-Clumps bestehen hingegen aus Variablen, welche als Argumente an Methoden übergeben werden [15]. Diese Art der Data-Clumps umfasst mindestens drei Parameter in mindestens zwei Methodensignaturen [13] [3]. Die Reihenfolge, in welcher diese auftreten, ist ebenfalls irrelevant.

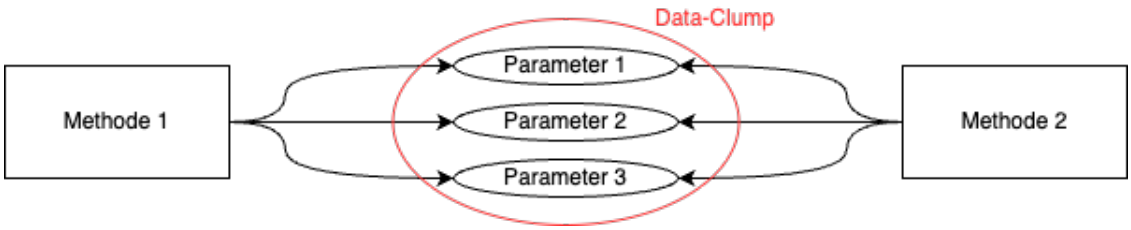


Abbildung 2: Beispielhafte Abbildung eines Parameter-Data-Clumps

Data-Clumps gehören als Unterkategorie der Code-Smells zu schlechten Programmierpraktiken mit den oben genannten Nachteilen [4]. Ebenso ist eine Verletzung des Single Responsibility Principle [16] theoretisch möglich, wenn diese auch durch die Ähnlichkeit der in dem Data-Clump vorkommenden Parameter/Felder unwahrscheinlich erscheint. Folglich ist es sinnvoll, die entstandenen Data-Clumps schnellstmöglich zu überarbeiten und somit einen qualitativ hochwertigen Code zu gewährleisten.

2.2 Das Refactoring

Refactoring bezeichnet den Prozess zur Überarbeitung und Verbesserung von Code-Abschnitten, wobei dessen Funktionalität vollständig erhalten bleiben soll [17]. Somit können Code-Smells durch gezieltes Refactoring beseitigt und die Qualität des Source-Codes deutlich verbessert werden [4]. Bezüglich der hier vorliegenden Data-Clumps werden die zugehörigen Felder beziehungsweise Parameter zu entsprechenden Objekten zusammengefasst. Die Felder/Parameter, welche vorher Teil des Data-Clumps waren, werden nach dem Refactoring als Attribute dieses Objekts zwar ebenfalls zusammen weitergegeben, jedoch stets als zusammenhängende Einheit innerhalb des Objekts [4].

2.3 Ziel des Refactorings

Die Parameter/Felder eines Data-Clumps sollen somit anstatt einzeln zusammen als Objekt weitergeben werden [4]. Dieses Vorgehen erhöht nicht nur die Übersichtlichkeit durch kürzere Methodensignaturen, da die Parameterliste sich verkürzt - es wird nur ein Objekt anstatt mehrere Parameter übergeben - sondern damit auch die Wartbarkeit des Codes. Neue Entwickler können die entsprechenden Abschnitte schneller verstehen und beschleunigen somit den kompletten Softwareentwicklungsprozess. Auch kann durch einen verbesserten Zugriff auf die einzelnen Variablen des Objekts die Performance der Software möglicherweise gesteigert werden. Das Refactoring für Data-Clumps besteht in der Regel aus dem Zusammenfassen der Parameter/Felder zu einer weiteren Klasse [4]. Somit kann ein Objekt dieser Klasse erzeugt werden und die Parameter können innerhalb dieses Objekts an andere Methoden und Klassen weitergegeben werden. Der detaillierte Ablauf des Refactorings ist jedoch nicht Gegenstand der Arbeit.

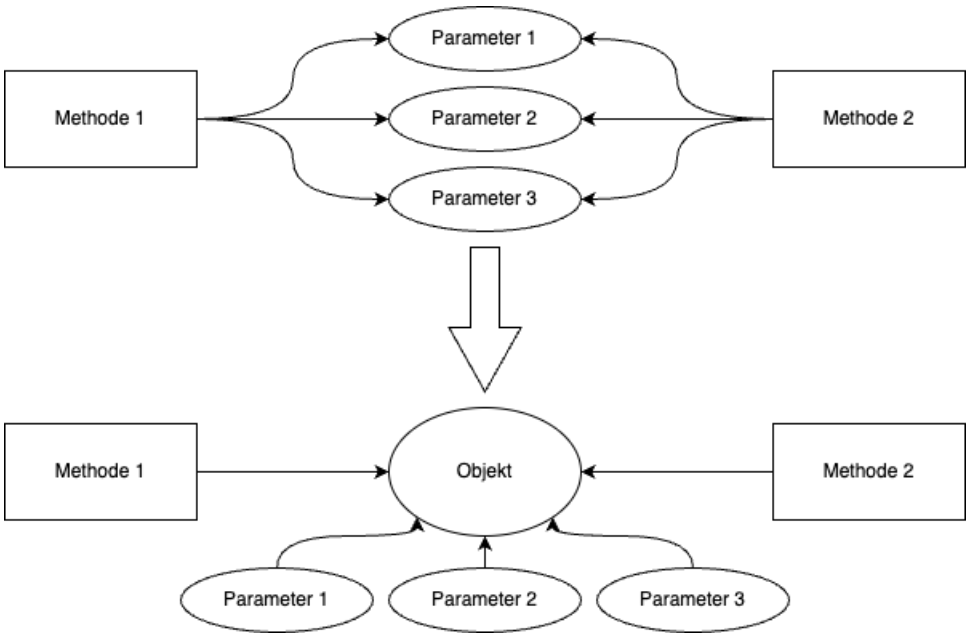


Abbildung 3: Beispielhafte Auflösung eines Parameter-Data-Clumps durch gezieltes Refactoring der Parameter

2.4 Die Tracing-Daten

Tracing-Daten bilden Beziehungen zwischen einzelnen Artefakten des Projektes ab [18]. Artefakte können in diesem Fall verschiedene Klassen oder Methoden im Quellcode selber oder aber externe Dokumente wie beispielsweise Anforderungen, Ablaufpläne oder auch *Unified Modeling Language* (UML)-Diagramme sein. Mit den Tracing-Daten können sowohl Beziehungen als auch Abhängigkeiten zwischen verschiedenen Teilen des Programms ausfindig gemacht und hinsichtlich dieser analysiert werden. Die Verknüpfung zweier Artefakte wird Trace-Link genannt, wobei diese Verbindung sowohl nur in eine als auch in beide Richtungen möglich ist [18]. Dadurch können ebenfalls Beziehungen über mehrere Ebenen hergestellt und genutzt werden. Durch diese Verbindungen wird die Reichweite einer Änderung eines betroffenen Abschnitts innerhalb des Codes besser sichtbar und kann schneller bewertet werden. Tracing-Daten tragen somit durch die Darstellung der Zusammenhänge der Artefakte und deren Analyse zur Verbesserung der Qualität des Source-Codes bei und erleichtern die Wartung sowie Weiterentwicklung [19] [20]. Für die Bewertung von Data-Clumps können Tracing-Daten genutzt werden, um die Verbreitung des Data-Clumps innerhalb des Source-Codes abschätzen zu können und um die Herkunft sowie die eventuell sogar notwendige Existenz dieser zu analysieren.

Als Basis der Bewertung der Data-Clumps dienen die eingangs beschriebenen Tracing-Daten. Die zum Beispielprojekt GanttProject gehörenden Tracing-Daten stammen aus einem Datensatz der Data Show Case Study von Mouna Hammoudi, Chritoph Mayr-Dorn sowie Alexander Egyed aus dem Jahr 2020 [21] [22]. Dieses Datenset beinhaltet

in der ersten Version die Dateien classes.json, methods.json, requirements.json und traces.json. Die Tracing-Daten orientieren sich dabei am Stand des Source-Codes vom 15. Juli 2010 [2]. Seitdem wurden die Tracing-Daten des Projekts inhaltlich nicht aktualisiert. Auch bei anderen Beispielprojekten lassen sich aktuelle Tracing-Daten kaum in einem für eine Bewertungsgrundlage angemessenen Umfang finden.

Die classes.json Datei gibt dabei im JSON-Format eine Liste von Klassen des Gantt-Projekts an. Zusätzlich zu den nummerierten Klassen mit eindeutigen Ids kann dem Klassennamen theoretisch auch der Pfad entnommen werden, welcher eine genaue Lokalisierung innerhalb der verschiedenen Dateien des Projekts möglich macht. Ob die jeweilige Klasse sich tatsächlich noch an dem durch den Pfad angegebenen Ort befindet, wird durch die Quelle nicht bestätigt. Der folgende Auszug zeigt eine von insgesamt 666 Klassen samt der Attribute `id` und `classname`:

```
1       {
2         "id":1,
3         "classname":"net.sourceforge.ganttproject.
4         ChartComponentBase$AbstractChartImplementation"
5       }
6
```

Listing 1: Auszug aus der classes.json Datei

Die methods.json Datei beinhaltet eine Liste der zugehörigen Methoden. Wichtig ist hierbei, dass diese sowohl mit dem Namen als auch mit dem kompletten Pfad angegeben werden, was die spätere Verknüpfung von Daten erleichtern kann. Außerdem können durch die Datei bereits Methoden und Klassen miteinander verknüpft werden, da jede Methode auch die eindeutige Klassen-Id und den Klassennamen beinhaltet. Auch ist für jede Methode des Projekts unter der Eigenschaft `method` der Source-Code sichtbar, was die Analyse von Variablenzuweisungen oder anderen Methodenaufrufen ermöglicht.

```
1       {
2         "id":11,
3         "methodname":"getChartModel()",
4         "methodnamerefined":"getChartModel",
5         "methodabbreviation":"net.sourceforge.ganttproject.
6         ChartComponentBase.getChartModel",
7         "fullmethod":"net.sourceforge.ganttproject.
8         ChartComponentBase.getChartModel()",
9         "classid":9,
10        "classname":"net.sourceforge.ganttproject.
11        ChartComponentBase",
12        "method":"protected abstract ChartModelBase
13        getChartModel();"
14      }
15
```

Listing 2: Auszug aus der methods.json Datei

Die Requirements.json Datei beinhaltet dagegen nur eine Liste mit den vorgegebenen Anforderungen an das Gantt-Programm. Diese sind ebenfalls durch eindeutige Ids durchnummeriert und beinhalten unter dem Attribut `requirementname` den Namen der jeweiligen Anforderung.

```
1     {
2       "id":1,
3       "requirementname":"01: Create Tasks"
4     }
5
```

Listing 3: Auszug aus der requierements.json Datei

Die wichtigste Datei in diesem Schritt ist die traces.json Datei. Diese verbindet die Listen der Klassen, Methoden und Anforderungen miteinander und gibt damit an, welche Klassen und Methoden für welche Anforderung von Relevanz ist. Gerade für eine Anforderungsanalyse ist dies von besonderem Wert, da somit direkt ersichtlich ist, welche Änderungen des Source-Codes welche Anforderung des Programms betreffen. Jeder Trace verbindet somit die Attribute `classid`, `methodid` sowie `requirementid`. Dadurch kann nicht nur festgestellt werden, welche Methoden und Anforderungen zu welcher Klasse gehören, sondern auch, welche Klassen und Methoden relevant für welche Anforderungen sind.

```
1     {
2       "id":19,
3       "requirement":"11: Change Task Begin/End Times manually
4       with user changes",
5       "requirementid":11,
6       "method":"net.sourceforge.ganttproject.
7       ChartComponentBase.-init-",
8       "methodname":"-init-()",
9       "fullmethod":"net.sourceforge.ganttproject.
10      ChartComponentBase.-init-()",
11      "methodid":2,
12      "classname":"net.sourceforge.ganttproject.
13      ChartComponentBase",
14      "classid":"9",
15      "goldfinal":"E",
16      "SubjectT":"1",
17      "SubjectN":"6"
18    }
19
```

Listing 4: Auszug aus der traces.json Datei

Ergänzt werden diese Daten durch die Tracing-Daten der dritten Version der Data Show Case Study [23]. Diese Version beinhaltet zusätzlich folgende Dateien:

- superclasses.json
- interfaces.json
- methodcalls.json
- parameters.json
- fieldclasses.json
- fieldmethods.json

Die superclasses.json Datei gibt dabei eine Liste von Superklassen und Subklassen an. Die Superklasse vererbt in der Regel Methoden und Attribute an die Unterklasse, wodurch ein Abhängigkeitsverhältnis zwischen diesen beiden Klassen entsteht [24]. Folglich stellt jeder Trace der Datei eine Verbindung zwischen einer Superklasse und einer Subklasse dar. Die Superklasse wird durch die Attribute `superclassid` und `superclassname`, die Subklasse durch `ownerclassid` sowie `childclassname` abgebildet. Die Ids und Namen beziehen sich dabei stets auf die Ids der Klassen aus der anfangs gezeigten classes.json Datei. Für eine einfache Zuweisung genügt jedoch ausschließlich das Attribut `id`.

```
1     {
2       "id":1,
3       "superclassid":646,
4       "superclassname":"net.sourceforge.ganttproject.time.
5       gregorian.GregorianCalendar",
6       "ownerclassid":24,
7       "childclassname":"net.sourceforge.ganttproject.
8       GanttCalendar"
9     }
10
```

Listing 5: Auszug aus der superclasses.json Datei

Die interfaces.json Datei beinhaltet Traces zwischen Klassen und Interfaces und gibt an, welche Klassen welches Interface implementieren. Die Interfaces sind dabei Teil der classes.json Liste, sodass die entsprechenden Verknüpfungen zwischen dem Interface mit `interfaceclassid` sowie `interfacename` und der implementierenden Klasse mit `ownerclassid` und `classname` hergestellt werden können. Interfaces bilden an dieser Stelle eine weitere Möglichkeit der Verbindung von Feldern und Klassen ab und können somit für die Bewertung von Data-Clumps sehr wertvoll sein – siehe Abbildung 4.

```
1     {
2       "id":7,
3       "interfaceclassid":185,
4       "interfacename":"net.sourceforge.ganttproject.
5       chart.GanttChart",
6       "ownerclassid":45,
7       "classname":"net.sourceforge.ganttproject.
8       GanttGraphicArea"
9     }
10
```

Listing 6: Auszug aus der interfaces.json Datei

Die Datei methodcalls.json stellt die Verbindung zwischen den Methoden dar [22]. So ist es beispielsweise möglich, dass eine Methode eine andere Methode aufruft. Gerade für die Analyse von Verschachtelungen ist dies besonders wichtig, da dadurch auch die zugehörigen Klassen und Anforderungen in Verbindung gebracht werden können. Die Traces bestehen aus einem `Caller`, dem Aufrufer, und einem `Callee`, die Klasse oder auch Methode, welche aufgerufen wird. Die Klassen stützen sich, wie bereits geschildert, auch hier auf die Klassen der classes.json Datei. Somit können beispielsweise Ketten von Aufrufen oder aber auch mögliche Kreise analysiert werden.

```
1     {
2       "id":14,
3       "callermethodid":8,
4       "callername":"createCopy()",
5       "callerclass":"net.sourceforge.ganttproject.
6       ChartComponentBase",
7       "callerclassid":"9",
8       "fullcaller":"net.sourceforge.ganttproject.
9       ChartComponentBase.createCopy()",
10      "calleemethodid":11,
11      "calleename":"getChartModel()",
12      "calleeclass":"net.sourceforge.ganttproject.
13      ChartComponentBase",
14      "calleeclassid":"9",
15      "fullcallee":"net.sourceforge.ganttproject.
16      ChartComponentBase.getChartModel()"
17    }
18
```

Listing 7: Auszug aus der methodcalls.json Datei

Auch die einzelnen Parameter werden in der parameters.json Datei analysiert. Diese gibt eine Liste von 3843 Parametern an, welche innerhalb des Gantt-Programms benutzt und übergeben werden. Wichtige Eigenschaften der Parameter sind hier der Parametername `parametername` und der zugehörige Typ des Parameters `parametertype`. Ebenfalls

besitzt jeder Parameter eine eindeutige Id. Um einen Parameter einer Methode beziehungsweise Klasse zuordnen zu können, ist jeder Parameter über die Attribute `classid/methodid` über die eindeutige Id mit der jeweiligen Methode beziehungsweise Klasse verbunden. Zusätzlich dazu geben die Attribute `isreturn` und `sourcecode` weitere Informationen über den Code des Programms, welche für die Analyse der Methoden und dessen Parameter von großer Wichtigkeit sein können. Wenn das Attribut `isreturn` auf true, also 1, gesetzt ist, fungiert der entsprechende Parameter als Rückgabe, sonst als Eingabeparameter [22].

```
{
  "id":13,
  "parametername":"Graphics g",
  "parametertype":"java.awt.Graphics",
  "parameterclass":0,
  "classid":3,
  "classname":"net.sourceforge.ganttproject.
  ChartComponentBase$MouseInteraction",
  "methodid":64,
  "methodname":"net.sourceforge.ganttproject.
  ChartComponentBase$MouseInteraction.paint(
  java.awt.Graphics)",
  "isreturn":0,
  "sourcecode":"void paint(Graphics g);"
}

```

Listing 8: Auszug aus der parameters.json Datei

Die Datei fieldclasses.json gibt eine Liste von Feldern an, welche zur Speicherung von Daten innerhalb einer Klasse benutzt werden [12]. Dabei kann jedes Feld mit einer Klasse durch das Attribut `ownerclassid` und `classname` verbunden werden. Jedes Feld besitzt einen Typ `fieldtype` und einen Namen `fieldname`. Durch diese Verknüpfung können auch Data-Clumps bestehend aus Feldern erkannt und analysiert werden.

```
{
  "id":31,
  "fieldname":"V2_0_1",
  "fieldtypeclassid":0,
  "fieldtype":"java.lang.String",
  "ownerclassid":21,
  "classname":"net.sourceforge.ganttproject.GPVersion"
}

```

Listing 9: Auszug aus der fieldclasses.json Datei

In der fieldmethods.json können die oben genannten Felder mit Hilfe der Ids und des Namens einer Methode zugeordnet werden. Das Attribut `ownermethodid` bezieht sich dabei wieder auf die Ids der Methoden aus der methods.json Datei. Zudem gibt das Attribut

`read` an, ob die Felder innerhalb der Methode gelesen werden, was ein entscheidender Faktor für die Komplexität sein und Auswirkungen auf ein Refactoring haben kann.

```
1       {
2         "id":1,
3         "fieldclassid":"422",
4         "fieldname":"treetable",
5         "ownerclassname":"net.sourceforge.ganttproject.
6         GanttTree2",
7         "ownerclassid":"73",
8         "ownermethodname":"setActions()",
9         "ownermethodid":"869",
10        "read":"1"
11      }
12
```

Listing 10: Auszug aus der fieldmethods.json Datei

Durch die sechs weiteren Tracing-Daten der dritten Version der Data Show Case Study [23] lassen sich insgesamt mehr Eigenschaften verknüpfen und somit mehr Informationen über die Beziehungen der einzelnen Tracing-Daten gewinnen, welche als Grundlage der Bewertung unabdingbar sind. Ebenfalls werden die zugehörigen MySQL-Skripte analog zu jeder JSON-Datei zur Verfügung gestellt. Da diese Arbeit jedoch die Bewertung von Data-Clumps ohne die Nutzung von Datenbanken fokussiert, werden diese Skripte zur Filterung von Eigenschaften nicht benötigt. In der nachfolgenden Grafik werden alle Tracing-Daten samt Verbindungen untereinander dargestellt.

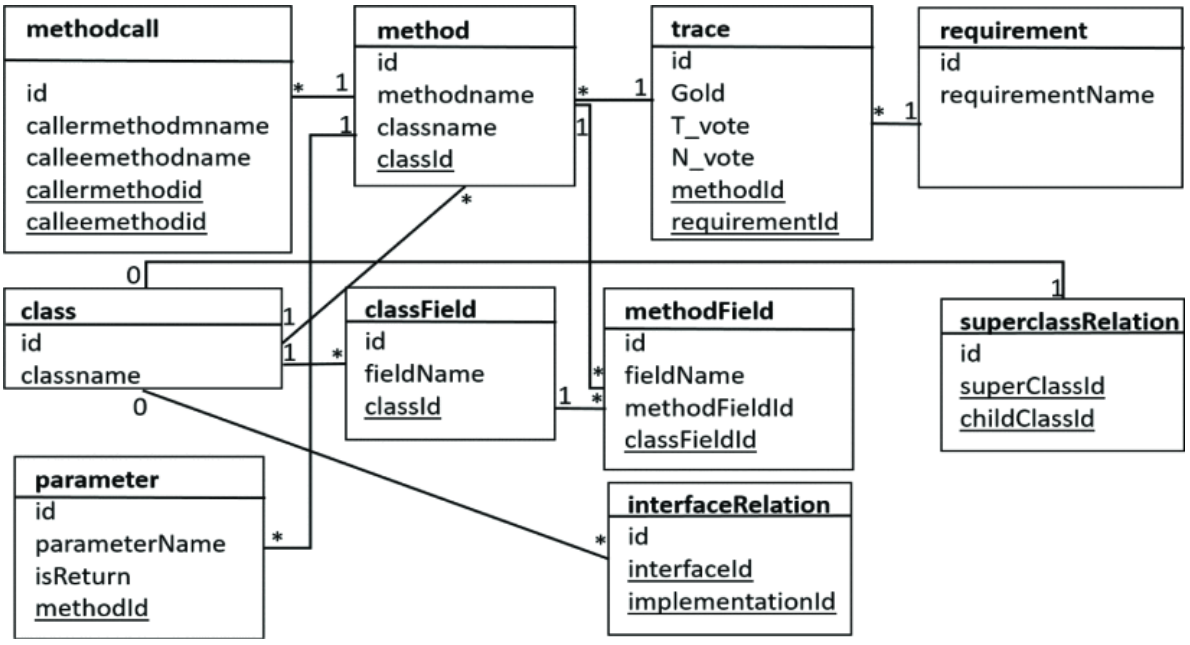


Abbildung 4: Verknüpfung aller verfügbaren Tracing-Daten [22]

2.5 Open-Source-Projekt GanttProject

Die vorliegende Arbeit orientiert sich am open-source Projekt GanttProject von bardsoftware, welches eine Desktop-App für das oftmals im Bereich des Projektmanagement eingesetzten Gantt-Diagramms darstellt [2]. Dabei ermöglicht das Gantt-Diagramm eine Visualisierung von zeitlichen Abläufen und Aufgaben innerhalb eines Projekts und trägt nicht nur zur besseren Übersichtlichkeit, sondern durch die Kontrolle von Meilensteinen auch zur Einhaltung beziehungsweise Überwachung von Teilzielen bei [25]. Dieses Projekt wurde als Grundlage der Arbeit gewählt, da es als open-source Projekt frei verwendet werden darf und ein entsprechendes Datenset der Tracing-Daten durch die Data Show Case Study von Mouna Hammoudi, Christoph Mayr-Dorn, Atif Mashkoor und Alexander Egyed [23] [22] bereits vorliegt. Des Weiteren weist das Programm auf Grund seiner Länge einen geeigneten Umfang für die Art dieser Abschlussarbeit auf.

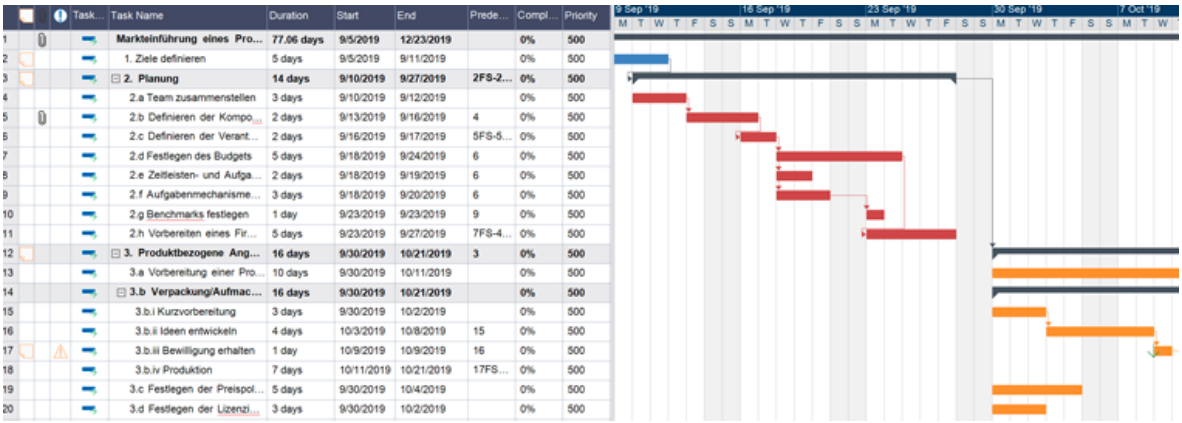


Abbildung 5: Beispielhafte Darstellung eines Gantt-Diagramms [25]

3 Konzept

Im folgenden Hauptteil wird ein theoretisches Konzept erarbeitet, welches die automatisierte Bewertung von Data-Clumps mit Hilfe von Tracing-Daten als Ziel hat. Dabei werden anhand der Eigenschaften von Data-Clumps verschiedene Bewertungskriterien sowie ein Algorithmus zur Berechnung eines Gesamtwertes pro Data-Clump entwickelt. Ebenfalls wird die Aussagekraft der Bewertungskriterien samt Umsetzungsmöglichkeiten thematisiert. Die Arbeit orientiert sich dabei an dem open-source Projekt GanttProject, welches eine Visualisierung des im Bereich des Projektmanagement eingesetzten Gantt-Diagramms darstellt [2].

3.1 Die vorliegenden Tracing-Daten

Aufbauend auf den in den Grundlagen genannten Tracing-Daten soll für eine Bewertung und damit einhergehend für eine mögliche Rückverfolgbarkeit eine Verknüpfung dieser geschaffen werden. Dabei ist es möglich, die einzelnen Tracing-Daten anhand verschiedener Attribute miteinander derart zu verbinden, sodass ein Netz aus Daten für jeden Data-Clump entsteht. Damit hat der Data-Clump entsprechend Zugriff auf alle Attribute dieses Netzes und diese können zur Bewertung anhand verschiedener Kriterien genutzt werden. Die nachfolgende Abbildung zeigt die entwickelte Verknüpfung der Attribute der Data-Clumps mit denen der Tracing-Daten der ersten Version [21] und dient somit als Basis des Konzepts. So kann ein Data-Clump mit Hilfe des Methodennamens `from_method_name` nicht nur mit der entsprechenden Methode der methods.json Datei, sondern auch mit einem Trace-Link der traces.json Datei verbunden werden. Dieser Trace-Link ist wiederum durch die Attribute `requirementid` sowie `requirement` mit der entsprechenden Anforderung, welche durch diese Methode teils umgesetzt wird, verknüpft. Durch diese Zusammenhänge kann folglich anhand der Tracing-Daten ermittelt werden, welche Data-Clumps welche Anforderungen betreffen und wie weit diese innerhalb des Source-Codes verbreitet sind. Diese beiden Aspekte können zur automatisierten Bewertung von Data-Clumps genutzt werden. Analog dazu können ebenfalls die To-Attribute der Data-Clumps analysiert werden, sofern diese von den From-Attributen abweichen. From- und To-Attribute bilden den Verlauf sowie die Zugehörigkeit von Parametern zwischen zwei Methoden beziehungsweise von Feldern zwischen zwei Klassen ab. Ergänzend zu Abbildung 6 können ebenfalls alle Attribute der durch die dritte Version der Data Show Case Study [23] hinzugefügten Tracing-Daten mit den Data-Clumps verknüpft werden. Insgesamt erhält jeder Data-Clump durch das Hinzufügen der Tracing-Daten zu einem Netz eine erweiterte Liste von Attributen anhand derer eine automatisierte Bewertung vorgenommen werden kann.

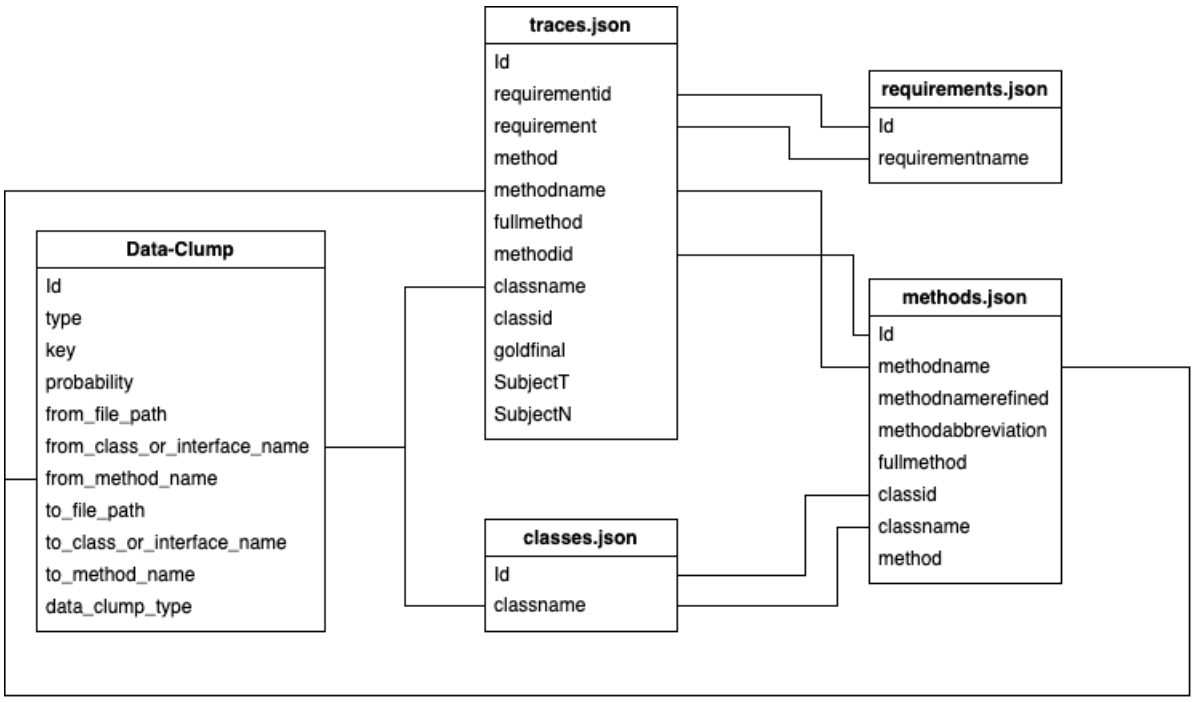


Abbildung 6: Verknüpfung der Tracing-Daten (Version 1,[21]) mit den Eigenschaften der Data-Clumps

3.2 Die Bewertungskriterien

Die Bewertung der Data-Clumps soll mit Hilfe verschiedener Kriterien erfolgen. Um diese Kriterien zu erarbeiten, müssen auch die Eigenschaften der Data-Clumps genauer betrachtet werden. Wie bereits beschrieben, besteht ein Datenklumpen aus einer festen Anzahl Parametern oder Feldern. Diese Parameter befinden sich dabei stets in der Signatur einer Methode, welche wiederum zu einer bestimmten Klasse innerhalb des Quellcodes gehören. Somit kann alleine durch die zugehörigen Parameter mit Hilfe der vorliegenden Tracing-Daten die Zugehörigkeit zu diversen Methoden und damit auch Klassen bestimmt und getestet werden. Dies führt dazu, dass jeder Data-Clump ebenfalls mit allen Eigenschaften der zugehörigen Methoden und Klassen verbunden werden kann und diese auf den Data-Clump anwendbar sind.

3.2.1 Mögliche Kriterien zur Bewertung

Aus den Eigenschaften der Data-Clumps sowie verschiedenen Tracing-Daten ergeben sich folgende Kriterien für eine Bewertung:

Die Verbreitung des Data-Clumps

Die Verbreitung des Data-Clumps gibt an, über wie viele Module / Klassen - aber auch Methoden - sich der Data-Clump samt zugehöriger Parameter/Felder erstreckt. Je mehr

Klassen und Methoden diesen Data-Clump enthalten, umso schwieriger ist es, diesen zu bereinigen, ohne unerwünschte Nebeneffekte auszulösen. Auch die Dauer des Refactorings durch die reine Anzahl an notwendigen Änderungen spielt hier eine Rolle. Es muss ausdrücklich erwähnt werden, dass die reine Anzahl an Data-Clump-Vorkommnissen in Klassen nicht zwangsläufig ein Gradmesser für deren Komplexität sein muss, jedoch ist die Anzahl der Änderungen für eine Abschätzung des Aufwandes eines Refactorings durchaus als sinnvoll zu betrachten. Die Information, in wie vielen Klassen der Data-Clump auftritt, stammt unmittelbar aus den Tracing-Daten der Methoden, Klassen und Traces. Außerdem gibt der Data-Clump-Detektor an, in welchen Methoden und Klassen der Data-Clump gefunden wurde. Ein weiterer Weg der Datengewinnung kann es sein, über die verwendeten Parameter in den Parameter-Tracing-Daten die zugehörigen Parameter-Klassen zu filtern und durch diese wiederum auf die Klassen und Methoden zu schließen. Somit sind alle notwendigen Daten und Eigenschaften der Data-Clumps für dessen Bewertung anhand der Verbreitung im Source-Code vorhanden (siehe Abbildung 4).

An dieser Stelle muss die absolute Anzahl der Klassen in einen Wert zwischen null und 100 konvertiert werden. So könnten die Häufigkeiten klassifiziert werden und einen festen Wert je nach Klassenzugehörigkeit zurückgeben:

Anzahl Vorkommen Klassen	**Bewertung (Punkte)**
≤ 1	20
2 - 3	40
4 - 5	60
6 - 7	80
≥ 8	100

Tabelle 1: Beispielhafte Kategorisierung der Bewertung anhand der Verbreitung von Data-Clumps innerhalb des Source-Codes

Jedoch können in diesem Fall nur sehr grobe Abschätzungen getroffen werden und die einzelnen Klassen auf Grund einer zu stark variierenden Spanne zwischen dem Data-Clump mit den wenigsten und dem mit den meisten Vorkommen besonders groß werden. Damit werden die Klassen als solche sehr ungenau, da diese mit wachsendem Vorkommen auch einen größeren Werte-Bereich abdecken müssen. Deshalb könnten zwei Data-Clumps - obwohl diese sich in der Häufigkeit der Klassen, in denen diese vorkommen, stark unterscheiden - derselben Kategorie eingeordnet und somit gleich bewertet werden. Folglich wird jedes Vorkommen einzeln bei der Bewertung berücksichtigt. Nach der Ermittlung des Data-Clumps mit der weitesten Verbreitung wird der größtmögliche Wert durch diese Anzahl geteilt, um einen Anteil pro Vorkommen zu erhalten. Danach wird der Wert pro Data-Clump mit der jeweiligen Anzahl multipliziert. Der Data-Clump mit der größten Verbreitung hat damit den Wert 100 und alle anderen Data-Clumps können relativ zu diesem bewertet werden. Die Berechnung setzt sich dann wie folgt zusammen:

$$Wert = \frac{100}{Anzahl_{MaxKlasse}} * Anzahl_{betroffeneKlassen} + \frac{100}{Anzahl_{MaxMethoden}} * Anzahl_{betroffeneMethoden}$$

Diese Bewertung ist auf Grund der Genauigkeit und der Möglichkeit der relativen Betrachtung untereinander deutlich geeigneter als die zuvor genannte Klassifizierung. Ebenfalls stellt sich die Frage, ob die Bewertung anhand der Verbreitung linear ist oder sogar exponentiell betrachtet werden kann, da die Schwierigkeit eines Refactorings mit zunehmender Verbreitung stark ansteigt. Für den Fall der Bewertung bezüglich der Notwendigkeit eines Refactorings reicht eine lineare Betrachtung jedoch aus.

Das Wissen über den Data-Clump

Das Wissen über den Data-Clump gibt an, wie viel Information das entwickelnde Team in Bezug auf den Data-Clump und den betroffenen Code hat. Viel Vorwissen erleichtert hier das Refactoring und vermeidet unter Umständen die Entstehung von unerwünschten Nebeneffekten, welche einen anderen Teil des Codes unbeabsichtigt negativ beeinflussen. Auch die Existenz einer geeigneten Dokumentation oder Kommentaren innerhalb des Source-Codes sind hier von großer Bedeutung, da dadurch mögliche Fehler schneller entdeckt und behoben werden können. Unzureichende Dokumentation kann den Prozess der Beseitigung des Data-Clumps sowie die Abschätzung der Komplexität erschweren. Ein großer Nachteil dieses Kriteriums ist, dass - abgesehen von der Dokumentation - das Wissen über den Code als solches rein subjektiver Natur ist und keiner objektiven Skala zugeordnet werden kann. Auch verhält sich dieser Aspekt relativ zwischen den einzelnen Entwicklern, sodass nicht von einem einheitlichen Wissen ausgegangen werden kann. Folglich lassen sich Data-Clumps zwar anhand der Information der Entwickler bewerten, jedoch ist dieses Kriterium keine objektive Grundlage für eine automatisierte Bewertung.

Häufigkeit von Änderungen

Ebenfalls wichtig ist, wie oft die Parameter des Data-Clumps ihren Wert während der Ausführung des Programms ändern. Parameter, die bei der Initialisierung einen Wert zugewiesen bekommen und diesen über die gesamte Lebensdauer des Objekts nicht verändern, sind einfacher zu bereinigen als Parameter, welche ihren Wert sehr häufig wechseln. Die Auswertung gestaltet sich für dieses Kriterium allerdings sehr schwierig, da es keine direkte Möglichkeit gibt, den Zugriff auf den Wert der Parameter/Felder zu messen, ohne eine entsprechende Kontrollstruktur direkt im Source-Code zu verankern. Da der schreibende Zugriff auf den Source-Code nicht gegeben ist, ist eine Messung für die Bewertung des Data-Clumps an dieser Stelle nicht möglich, aber für weitere Projekte durchaus denkbar. Im Hinblick auf die Notwendigkeit eines Refactorings sollte ein häufig wechselnder Parameter einen höheren Wert zugewiesen bekommen.

Komplexität des betroffenen Codes

Die Komplexität gibt an, in welchem Kontext die Parameter/Felder innerhalb einer Methode genutzt werden. Dabei gibt es mehrere Möglichkeiten diese zu bestimmen. Zum einen lässt sich die Komplexität durch den Source-Code der Methode an sich bestimmen, zum anderen können die Datentypen der Parameter zwischen benutzerdefinierten Datentypen und einfachen Java-Datentypen – wie beispielsweise Integer – unterschieden werden – siehe Kapitel 3.2.1. Der Source-Code jeder Methode kann dem Attribut `method` der methods.json Datei entnommen werden. Innerhalb des Source-Codes kann ebenfalls auf etwaige Rückgaben geprüft werden, welche somit samt der Komplexität der Datenstruktur in die Bewertung mit einbezogen werden können. Die im Data-Clump enthaltenen Parameter werden außerdem durch das Attribut `isreturn` der parameters.json-Datei beschrieben, welche angibt, ob ein Parameter ein Eingabeparameter oder ein zurückgegebener Parameter ist. Darüber hinaus kann mit dem Attribut `sourcecode` auf den Source-Code zugegriffen werden, welcher den entsprechenden Parameter enthält. Um den Source-Code an sich als Bewertungsgrundlage nutzen zu können, muss ein Konzept zur Bewertung dessen samt Einteilung in verschiedene, abgrenzbare Kategorien entwickelt werden. So könnte die Bewertung ähnlich zur O-Notation mittels Abschätzung einzelner Abschnitte oder Verschachtelungen erfolgen. Dafür wird der Source-Code der Methode nach verschiedenen Worten, wie if, while und for durchsucht, durch welche Bedingungen oder Schleifen angegeben werden. Dabei sind sowohl die Anzahl als auch die Schleifen selbst von großer Bedeutung. Auch an dieser Stelle wird der Unterschied zwischen verschiedenen Zielsetzungen besonders deutlich. Ein Data-Clump, welcher Bestandteil eines komplexen Code-Abschnitts ist, kann sowohl besonders refactoringbedürftig als auch auf Grund der Komplexität sehr ungeeignet bezüglich einer Verbesserung sein. Da der detaillierte Ablauf des Refactorings aber keinen Teil dieser Arbeit darstellt, werden mögliche Hindernisse auf Grund der Komplexität außer Acht gelassen. Somit ist ein Data-Clump mit einer hohen Komplexität am dringendsten zu verbessern und bekommt eine maximal hohe Bewertung. In den nachfolgenden Tabellen sind beispielhafte Bewertungen anhand der Anzahl und der genannten keywords abgebildet.

Anzahl Keywords	**Bewertung**
≤ 1	20
2 - 3	40
4 - 5	60
6 - 7	80
≥ 8	100

Tabelle 2: Bewertung Anzahl

Keyword	**Bewertung**
≤ 1	20
2 - 3	40
4 - 5	60
6 - 7	80
≥ 8	100

Tabelle 3: Bewertung Keywords

Mit Hilfe dieser Bewertung kann die Gesamtkomplexität des betroffenen Abschnitts mit folgender Formel berechnet werden:

$$Komplexität_{Parameter} = Keywords_{Anzahl}/10 + Codelänge/100$$

Testabdeckung für Code (Unit und Integration)

Ein weiteres mögliches Kriterium zur Bewertung von Data-Clumps ist die Frage nach einer ausreichenden Testabdeckung für den durch einen Data-Clump betroffenen Bereich. Eine vollständige Testabdeckung hilft dabei, eventuelle Fehler oder Seiteneffekte, welche durch das Refactoring auftreten können, zu identifizieren. Die Testabdeckung hilft dabei, einen fehlerfreien und sicheren Ablauf des Programms zu gewährleisten und ist im Bereich der Qualitätssicherung von großer Bedeutung [26]. Relevant dafür ist sowohl die Abdeckung durch Unit-Tests für die Gewährleistung der Funktionalität in dem betroffenen Abschnitt als auch die Abdeckung durch Integrations-Tests, um zu gewährleisten, dass etwaige Änderungen auch im Gesamtsystem ohne unerwünschte Nebeneffekte bleiben. Jedoch kann auf die Testabdeckung nur mit Hilfe des Source-Codes durch eventuell vorhandene Testklassen eingegangen werden. Da diese Testklassen in dem vorliegenden Beispielprojekt GanttProject nicht existieren und auch keine weiteren Informationen bezüglich der Testabdeckung vorliegen, kann dieses Kriterium nur theoretisch behandelt werden und eignet sich für die Umsetzung innerhalb des Projektes an dieser Stelle nicht.

Zukünftige Anforderungen

Auch können zukünftige Anforderungen die Eignung eines Refactorings der Data-Clumps beeinflussen. So kann der durch die Data-Clumps betroffene Bereich des Codes bereits zum Bewertungszeitpunkt schon refactoringbedürftig sein oder darf bis zu den geplanten Änderungen nicht angepasst werden. Im letzten Fall ist ein Refactoring der Data-Clumps folglich nicht zulässig und somit ungeeignet. Eine automatisierte Bewertung scheint jedoch auf Grund mangelnder Daten in diesem Fall nicht möglich.

Alter des Data-Clumps

Das Alter eines Data-Clumps kann theoretisch über das Datum des passenden Commits, also das Hinzufügen des relevanten Code-Abschnitts zum Repository, ermittelt werden. Anhand des Alters können die betroffenen Abschnitte schnell in Kategorien eingeteilt werden, ob es sich dabei um neuen Code handelt, welcher innerhalb der letzten Monate entwickelt wurde, oder ob es sich um einen mehrere Jahre alten Code handelt, welcher bereits zu der Kategorie Legacy-Code zählt. Der Legacy-Code ist in zweierlei Hinsicht nicht sinnvoll zu verbessern. Zum einen ist der Code so alt, dass das aktuelle Entwicklerteam wahrscheinlich nicht über ausreichendes Wissen bezüglich des relevanten Source-Codes verfügt (siehe Kriterium Wissen) und zum anderen, ob der Code nicht bereits durch besseren Code ersetzt wurde und somit nicht mehr relevant ist, was ein Refactoring überflüssig macht. Auch hier erfolgt die Bewertung mit einer Kategorisierung der Zeit in mehrere Monatsabschnitte beginnend bei dem Start der Bewertung. Je älter der Code ist, desto geringer die Punktzahl, welche dem enthaltenen Data-Clump zugewiesen wird.

Rechtliche und Sicherheits-Aspekte

Rechtliche und sicherheitsrelevante Aspekte können ebenfalls ein wichtiges Kriterium zur Bewertung von Data-Clumps darstellen. So können rechtliche Faktoren wie beispielsweise bezüglich der Datenschutzgrundverordnung in besonderen Fällen für eine Änderung des von den Data-Clumps betroffenen Codes ein großes Hindernis darstellen. Damit einhergehend ist auch die Frage nach der Sicherheitsrelevanz des betroffenen Codes zu untersuchen. Greifen die Parameter/Felder - aus denen der Data-Clump besteht - in sicherheitsrelevante Systeme ein, erfordert ein Refactoring sowie dessen Prüfung einen erheblichen Mehraufwand. Auch muss die Funktionalität zu jedem Zeitpunkt gewährleistet werden, was ein Refactoring zusätzlich erschwert. In diesem Fall wäre eine Veränderung des Data-Clumps nicht sinnvoll. Diese Fragen müssen zunächst gründlich geprüft werden und sind durch die vorliegenden Tracing-Daten und den Source-Code nur schwer zu beantworten, da diese keine Hinweise auf genannte Kriterien liefern. Folglich sind rechtliche und Sicherheitsaspekte ein theoretisch sehr gutes Kriterium zur Bewertung von Data-Clumps, es kann an dieser Stelle jedoch nicht verwendet werden, da die notwendigen Informationen durch die zugrundeliegenden Daten nicht gegeben sind.

Anforderungen an das Programm (Requirements)

Ein weiteres wichtiges Kriterium anhand dessen eine Bewertung der Data-Clumps stattfinden kann, sind die aktuellen Anforderungen an das Gesamtsystem. Bereiche des Programms, welche unmittelbar wichtige Anforderungen betreffen, sind tendenziell wichtiger zu verbessern als andere Bereiche, da der Code durch die Beseitigung des Data-Clumps nicht nur einfacher, sondern auch verständlicher für andere Entwickler wird. Die Tracing-Daten des hier verwendeten Beispielprojekts GanttProject beinhalten in der requirements.json Datei [23] insgesamt 18 Requirements, wovon 17 Anforderungen an das Programm selbst gerichtet sind und ein Aspekt den generell für die Ausführung des Programms wichtigen Code beschreibt. Diese Anforderungen lassen sich in verschiedene Kategorien einteilen, welche die Anforderungen nach Relevanz beinhalten. Durch die Verknüpfung der Anforderungen mit den Methoden, Klassen und Data-Clumps kann somit schnell festgestellt werden, ob ein Data-Clump im Bereich einer wichtigen oder durchschnittlichen Anforderung liegt. Für eine genauere Betrachtung können auch diese Gruppen wieder geteilt werden, bis jede Anforderung eine eigene Wichtigkeit erhält. Der Score für dieses Kriterium orientiert sich somit an der jeweiligen Gruppe der Anforderungen. In der nachfolgenden Tabelle wird dieser Aspekt mit Hilfe einer beispielhaften Einteilung der Anforderungen an das Gantt-Projekt nochmal verdeutlicht. Wichtig ist, dass es sich hierbei um eine subjektive Einschätzung der Anforderungen handelt, welche stets variieren kann und nur in Verbindung mit der Berechnung sowie dessen Veranschaulichung gültig ist. Aufgeführt werden hier bewusst nur die highlevel Requirements. Die dazugehörigen lowlevel Requirements, in welche jedes highlevel Requirement unterteilt werden kann, enthält das Daten-Set von Holbrook, Hayes und Dekhtyar [27] [28]. Diese können ebenfalls weiter zur Einordnung der betroffenen Anforderungen und somit zur Bewertung der Data-Clumps beitragen. Allerdings müssen auch diese Teilanforderungen

wieder manuell geschätzt werden, weshalb eine Bewertung anhand von Anforderungen im Folgenden nur mit den highlevel Requirements durchgeführt wird. Die folgende Bewertung soll eine beispielhafte Kategorisierung der einzelnen Anforderungen darstellen. Dabei muss jede Anforderung nach dessen Priorität innerhalb des Projektes mit einem festen Wert verknüpft werden. So bilden beispielsweise sowohl das Erstellen als auch das Löschen von Aufgaben elementare Funktionen innerhalb des Projekts GanttProject ab und werden deshalb mit dem höchstmöglichen Wert auf einer Skala von 0 bis 10 bewertet. Analog dazu kann das Hinzufügen oder Entfernen von Urlaubstagen als weniger wichtig im Hinblick auf den Gesamtumfang des Programms angesehen werden und wird folglich nur mit dem Wert 2 verknüpft.

Anforderungen (Requirements)	**Bewertung (Punkte)**
01: Create Tasks	10
02: Delete Tasks	10
03: Maintain Task Properties	6
04: Add/Remove Tasks as Subtasks	6
05: Handle Milestones	6
06: Create Resources (person)	10
07: Delete Resources (person)	10
08: Maintain Resource Properties	8
09: Add/Remove Task Links, inner	6
10: Add/Remove Resources to Tasks Dependencies	6
11: Change Task Begin/End Times manually with user changes	6
12: Change Task Begin/End Times automatically with dependency changes	4
13: Change Task Begin/End Times automatically with subtask changes	4
14: Prevent Circular Dependencies	6
15: Show Critical Path	8
16: Add / Remove Holidays and Vacation Days	2
17: Show Resource Utilization (underused or overused person)	2
XXX: General Purpose Code	10

Tabelle 4: Beispielhafte Bewertung der Anforderungen

Um den Gesamtwert für dieses Kriterium zu berechnen, wird zunächst der Wert pro Anforderung ermittelt. Da in vielen Fällen pro Data-Clump nur eine Anforderung betroffen

ist, wird der Wert durch die Schätzung in der Tabelle bestimmt. Damit das Ergebnis im Zahlenraum zwischen 0 und 100 liegt, wird das Ergebnis mit 10 multipliziert. Alternativ ist auch eine direkt Bewertung von 0 bis 100 möglich. Daraus ergibt sich die folgende Berechnungsformel:

$$Wert_{Anforderungen} = Bewertung_{Max} * 10$$

An dieser Stelle muss jedoch auch das `goldfinal` Attribut der Traces beachtet werden. Dieses gibt an, ob ein Trace, welcher zur Bestimmung der Verbindung mit den Anforderungen notwendig ist, eindeutig einem Requirement zugeordnet werden kann. Dies kann die Ausprägungen T für Trace, N für NoTrace sowie E für ein nicht eindeutiges Ergebnis haben. Der Wert wird durch die Mehrheit der Attribute `SubjectT` und `tSubjectN` bestimmt, welche die Bewertung der Entwickler des zu den Tracing-Daten gehörenden Projekts widerspiegelt [22]. So ist das Ergebnis der einschlägigen Anforderungen stark mit der Wahl des `goldfinal`-Attributes verknüpft. Bei nur eindeutigen Traces fällt die Anzahl der Requirements deutlich geringer aus als bei allen möglichen Anforderungen.

Performance

Ebenfalls ist eine Überprüfung der Performance sinnvoll, ob der durch das Refactoring des Data-Clumps betroffene Abschnitt die Leistung des Programms verbessert oder beeinträchtigt. Wie viel die relevante Methode zur Gesamtperformance des Programms beiträgt kann etwa durch Zeitmessungen – beispielsweise mit Hilfe einer Stopwatch bei mehrmaliger Ausführung des Code-Abschnitts – errechnet werden. Für die Umsetzungen werden jedoch verschiedene externe Tools benötigt. Vor allem müssen einzelne Teile des Programms gesondert ausgeführt werden können, was den Umfang dieser Arbeit überschreitet. Somit bleibt dieses Kriterium eine rein theoretische – wenn auch wichtige – Möglichkeit der Bewertung von Data-Clumps.

Mögliche alternative Behandlung des Data-Clumps

Innerhalb dieses Kriteriums stellt sich die Frage, ob es eine Alternative zum Refactoring des Data-Clumps gibt. Welche alternativen Möglichkeiten zur Behandlung von Data-Clumps existieren und wann diese sinnvoll einsetzbar sind, wird in dieser Arbeit nicht weiter thematisiert. Der Vollständigkeit halber sollte jedoch erwähnt werden, dass verschiedene Möglichkeiten der Behandlung existieren und anhand dieser sowie der jeweiligen Komplexität und Aufwandsschätzung ebenfalls eine Bewertung des Data-Clumps möglich ist. Ein Data-Clump, welcher alternativ behandelt werden kann, ist nicht dringend zu verbessern. Für das vorliegende Gantt-Projekt ist dieses Kriterium folglich nicht relevant.

Data-Clump als Ergebnis der Architektur

Möglicherweise ist der Data-Clump auch ein Ergebnis der verwendeten Architektur, wie es beispielsweise bei mehreren Schichten oder einer Trennung von Benutzeroberfläche

und Logik der Fall sein kann. Hier muss sorgfältig abgewägt werden, ob ein Refactoring überhaupt erfolgen kann, ohne die vorliegende Architektur zu beeinträchtigen. In diesem Beispiel wird die benutzte Architektur nur durch den Source-Code, nicht aber durch die Tracing-Daten ersichtlich, was eine Analyse dahingehend deutlich erschwert. Somit müsste der Quellcode extern analysiert werden, um Ergebnisse hinsichtlich der Fragestellung zu erhalten. Für die Bewertung dieses Kriteriums gäbe es folglich nur zwei Werte - nämlich 0 und 1 - welche angeben, ob der Data-Clump das Ergebnis der Architektur ist oder nicht. Auch die Flexibilität und Anpassungsfähigkeit einer möglichen Änderung könnten an dieser Stelle berücksichtigt werden, jedoch erfordert auch dieser Schritt die externe Analyse des Source-Codes.

Teil einer Superklasse / Anzahl der Ownerklassen

Die superclasses.json Datei gibt anhand der sich aus der classes.json ergebenden Klassen des Gantt-Projekts an, welche Klassen Superklassen beziehungsweise als Subklasse einer Superklasse zugehörig sind. Die Traces der superclasses.json verbinden mit dem Attribut `superclass` und `ownerclass` die Superklasse mit jeweils einer Subklasse. Superklassen als solche stellen zwar einen eigenen Code-Smell dar, werden aber in dieser Arbeit nicht eigenständig behandelt und dienen an dieser Stelle zusätzlich der Bewertung von Data-Clumps. Mit Hilfe der Namen und Ids der Klassen lassen sich diese gezielt filtern, sodass jede Klasse in eine Kategorie Superklasse oder Subklasse eingeteilt werden kann. Liegt ein Data-Clump in einem Super-Sub-Klassen-Konstrukt ist dieser von mehreren Klassen abhängig, was das Refactoring eines solchen Code-Abschnitts deutlich erschwert, da zusätzlich andere Klassen geändert werden müssen. Damit ist ein Eingriff in weite Teile des Codes aus mehren Aspekten, wie beispielsweise Erhaltung der Funktionalität oder Einhaltung von Richtlinien, nicht sinnvoll und sollte daher nicht priorisiert werden. Bezüglich der Notwendigkeit ist ein Refactoring genau dann wichtig.

Anzahl der Parameter/Felder innerhalb des Data-Clumps

Auch die Anzahl der Parameter/Felder (im Folgenden Parameter) eines Data-Clumps können eine wichtige Bewertungsgrundlage liefern. Je mehr Parameter ein Data-Clump enthält, umso wichtiger ist dessen Refactoring, um lange Methodensignaturen zu verkürzen. Die Anzahl der enthaltenen Parameter können direkt dem Ergebnis des hier verwendeten Data-Clump-Detektors entnommen werden. So gibt die Länge der Liste Data_Clump_Data innerhalb eines Data-Clumps der Data-Clump JSON-Datei an, um wie viele Parameter es sich handelt. Um den gewünschten Wert zwischen 0 und 100 zu erhalten, wird die Anzahl der Parameter mit dem Wert 20 multipliziert und vom Maximalwert 100 subtrahiert. Alle Werte unter 0 werden automatisch auf 0 gesetzt, da ein Data-Clump bestehend aus 5 oder mehr Parametern zwar auf jeden Fall dringend eines Refactorings bedarf, jedoch sehr komplex und damit ungeeignet ist.

Typen der Parameter / Felder

Die einzelnen Typen der im Data-Clump enthaltenen Parameter und Felder sind über das Attribut `parametertype` der `parameters.json` Datei aufrufbar. Diese geben an, ob die Parameter aus einfachen Datentypen, wie beispielsweise Intergern oder benutzerdefinierten Datentypen bestehen. Benutzerdefinierte Datentypen sind dabei wesentlich komplexer und müssen mit Vorsicht behandelt werden. Parameter mit einfachen Datentypen hingegen können leichter verbessert werden. Damit bieten diese sich für ein Refactoring mehr an. Die Bewertung kann an dieser Stelle sehr gut in Kombination mit dem zuvor genannten Aspekt der Anzahl an Parametern durchgeführt werden. Nachdem alle Werte der Parameter für einen Gesamtwert der Parametertypen aufsummiert werden, wird die Anzahl jedes Parameters mit diesem Wert - abhängig vom Typen des Parameters - multipliziert. Analog dazu werden auch hier mögliche Werte über 100 wieder auf den Maximalwert 100 reduziert.

Parametertyp	**Bewertung (Punkte)**
Einfache Datentypen	1
UI - abhängige Datentypen (javax)	3
Drittanbieter - Datentypen	5
Benutzerdefinierte Datentypen	8

Tabelle 5: Beispielhafte Kategorisierung und Bewertung der Parametertypen

Aufrufe weiterer Methoden

Oftmals rufen Methoden in sich wieder andere Methoden auf und geben entsprechende Parameter an diese weiter. Dieses Kriterium untersucht die einzelnen Methodenaufrufe mit Hilfe der Datei methodcalls.json. Diese gibt durch die Attribute `Caller` und `Callee` anknüpfend an die Methoden der Methode.json Datei an, welche Methode als Caller eine andere Methode (Callee) aufruft. Wenn eine von einem Data-Clump betroffenen Methode eine weitere - unter Umständen mit den gleichen oder ähnlichen Parametern - aufruft, erhöht dies nicht nur die Komplexität des Codes sowie dessen Verbreitung, sondern beeinträchtigt auch die Möglichkeiten eines Eingriffs in den Code, da dieser an allen weiterreichenden Stellen geändert werden muss. So kann es zu langen Ketten von Methodenaufrufen oder sogar - bei ungünstigen Bedingungen der Entscheidungslogik während der Laufzeit - zu vorübergehenden Schleifen oder Kreisen kommen. Diese sind oftmals sehr unübersichtlich, was einen Eingriff hinsichtlich der Gewährleistung der Funktionalität und Einhaltung diverser Richtlinien erschwert. Ist ein Data-Clump Teil eines solchen Konstrukts wird anhand der Anzahl der verknüpften Methoden sowie einer geeigneten Kategorisierung ein Wert bestimmt, welcher - je näher dieser an 0 liegt - von einem Refactoring abrät.

3.2.2 Starke Kriterien

Nach der Aufzählung der verschiedenen - theoretisch möglichen - Kriterien zur Bewertung von Data-Clumps werden diese in zwei Kategorien unterteilt - starke sowie schwache Kriterien. Die erste Kategorie umfasst dabei die für diese Arbeit relevanten Kriterien, welche objektiv mit Hilfe von Tracing-Daten bewertbar sind. Die Möglichkeit der objektiven Bewertung führt zur Kategorisierung innerhalb der Kriterien, an welchem sich der Bewertungsscore letztendlich orientiert. Somit kann durch die eindeutig definierten Grenzen ein Punktewert ermittelt werden, welcher nur von den Tracing-Daten abhängig ist und keine Schätzungen erfordert. Beispielsweise kann die Verbreitung des Data-Clumps anhand der Anzahl der betroffenen Klassen / Methoden ermittelt werden. Diese Anzahl wird in eindeutige Kategorien eingeteilt und ergibt einen Wert für die jeweilig betroffenen Kategorien oder kann direkt mit Hilfe der Anzahl bestimmt werden. Folglich sind die objektiven Kriterien für die automatisierte Bewertung von Data-Clumps von großer Bedeutung, da diese Einteilung direkt durch die Tracing-Daten erfolgen kann und nicht durch den Zeitaufwand sowie Ungenauigkeit externer Schätzungen, wie dies beispielsweise bei einer Analyse anhand des Wissens des Entwicklerteams der Fall ist, beeinflusst wird. Zu den objektiven Kriterien zählen somit:

- Die Verbreitung des Data-Clumps
- Die Häufigkeit von Änderungen
- Die Komplexität
- Die Testabdeckung
- Das Alter des Data-Clumps
- Die Anforderungen an das Programm (Requirements)
- Die Performance
- Teil einer Superklasse / Anzahl Ownerklassen
- Die Anzahl der Parameter / Felder
- Die Typen der Parameter / Felder
- Die Aufrufe weiterer Methoden

3.2.3 Schwache Kriterien

Schwache Kriterien sind Kriterien, mit denen ein Data-Clump nicht objektiv analysiert werden kann. Diese erfordern häufig externe Schätzungen und einen damit verbundenen hohen Grad an Subjektivität. Für eine einheitliche Bewertung von Data-Clumps ist dieses Vorgehen ungeeignet, da viele für diese Kriterien notwendigen Grundlagen stets

variieren. So kann das Wissen bezüglich eines Data-Clumps innerhalb des Entwicklerteams von Person zu Person unterschiedlich sein. Auch rechtliche Aspekte können nicht direkt objektiv bewertet werden, sondern müssen jeweils im Voraus von zuständigen Personen abgeschätzt werden. Ein weiterer Aspekt, warum subjektive Kriterien für die automatisierte Bewertung von Data-Clumps ungeeignet sind, ist die hohe Fehleranfälligkeit, Ungenauigkeit des Schätzens sowie Abhängigkeit von externen Personen. Besonders innerhalb der oben beschriebenen Pipeline soll eine Bewertung möglichst genau, schnell und unabhängig von Dritten geschehen. Folgende Kriterien basieren auf subjektiven Faktoren:

- Wissen / Informationen bezüglich des Data-Clumps
- Rechtliche und Sicherheitsaspekte
- Data-Clump als Ergebnis der Architektur
- Mögliche alternative Behandlung des Data-Clumps
- Zukünftige Anforderungen

Ebenfalls wichtig ist, dass eine Bewertung anhand dieser Kriterien keinesfalls gänzlich ausgeschlossen wird. Vielmehr soll diese Kategorisierung in starke und schwache Kriterien verdeutlichen, welche Kriterien für den konkreten Anwendungsfall der automatisierten Bewertung geeigneter sind.

3.3 Welche Kriterien lassen sich Stand jetzt nicht umsetzen?

Obwohl es viele Kriterien für die Bewertung von Data-Clumps gibt, lassen sich einige der genannten Kriterien aus verschiedenen Gründen nicht umsetzen. Der dabei wichtigste Aspekt ist die Datenlage. Viele Projekte verfügen nicht über ausreichende Tracing-Daten, die Dokumentation fehlt oder wichtige Zugriffsrechte sind nicht vorhanden. So ist zum Beispiel das Alter eines Data-Clumps, welches über die verknüpften Methoden und Klassen durch eine Suche in dem Repository nach dem passenden Commit, welcher diese hinzugefügt hat, bestimmt werden kann, nur schwer zurückzuverfolgen und für die Analyse des Data-Clumps in der Form nicht zielführend zu nutzen, da sowohl ein passendes Werkzeug zum Suchen sowie Auslesen des Commits als auch die zugehörigen Zugriffsrechte in dem vorliegenden Beispiel nicht gegeben sind. Die subjektiven Kriterien - Know-How, rechtliche und Sicherheitsaspekte, Ergebnis der Architektur, mögliche alternative Behandlung des Data-Clumps sowie zukünftige Anforderungen - sind auf Grund der bereits beschriebenen Faktoren nicht nur ungeeignet, sondern lassen sich auch durch die Datenlage in diesem konkreten Beispiel nicht umsetzen. Es fehlen wichtige Informationen bezüglich des Entwicklerteams, der Architektur sowie rechtliche und sicherheitsrelevante Daten. Aber auch Angaben zur Nutzung von Drittanbieter-Diensten

oder bezüglich der Testabdeckung durch Unit- und Integrationstests liegen dem open-source Projekt GanttProject nicht bei. Die Kriterien Häufigkeit von Änderungen der Data-Clumps, die Performance sowie Aufrufe von weiteren Methoden sind zwar prinzipiell möglich zu bewerten, jedoch werden für die Analyse stets externe Tools benötigt, die entweder die Ausführung eines bestimmten Teils des Codes überprüfen oder den Source-Code als solchen direkt hinsichtlich speziell vorkommender Zeichenketten analysieren.

Auf Grund der Beschränkung auf die vorliegenden Tracing-Daten des Projekts Gantt und den Verzicht auf externe Tools können die genannten Kriterien bei der in dieser Arbeit entwickelten automatisierten Bewertung von Data-Clumps nicht berücksichtigt werden und sind somit als rein theoretische Grundlage für weitere Projekte anzusehen. Analog zu der Kategorisierung in starke und schwache Kriterien ist deshalb zu beachten, dass die Bewertung anhand dieser Kriterien nicht generell ausgeschlossen wird und die Bewertungskriterien zukünftig oder mit anderer Technologie bei passender Datenlage ebenfalls gut für eine automatisierte Bewertung von Data-Clumps geeignet sein können.

3.4 Kriterien für das open-source Projekt GanttProject

Damit ergeben sich die folgenden Bewertungskriterien für Data-Clumps aus dem open-source Projekt GanttProject, anhand dessen eine Bewertung - auf Grundlage der vorliegenden Daten - realisiert werden kann:

- Anforderungen (Requirements)
- Komplexität des Data-Clumps
- Teil einer Superklasse / Anzahl Ownerklassen
- Anzahl und Typen der Parameter
- Verbreitung des Data-Clumps

Dabei werden die Kriterien Anzahl und Typen der in den Data-Clumps enthaltenen Parametern zusammengelegt, da diese ähnliche Eigenschaften untersuchen und um eine präzisere Bewertung innerhalb dieses Bewertungskriteriums zu erhalten. So kann beispielsweise ein Refactoring von fünf Parametern mit primitiven Datentypen zwar einfacher sein als bei Data-Clumps bestehend aus drei Parametern mit benutzerdefinierten Datentypen, diese sind jedoch auf Grund der geringeren Größe nicht dringender zu beseitigen. Somit werden beide Aspekte der enthaltenen Parameter/Felder bei der Berechnung des Wertes in folgender Formel berücksichtigt:

$$Wert_{Gesamt} = \frac{100}{Anzahl_{M}axParameter} * \sum_{1}^{ParameterKategorie} AnzahlParameter_{ProKategorie} * \frac{}{\sum_{WerteDerParameterProKategorie}}$$

Ebenfalls kann das Kriterium der Methodenaufrufe in das Kriterium der Komplexität integriert werden, sodass sich der berechnete Wert aus beiden Kriterien zusammensetzt.

3.5 Die Bewertung

Wie den Anforderungen an das Programm zu entnehmen ist, ist es das Ziel der automatisierten Bewertung, für jeden Data-Clump einen geeigneten Wert zu ermitteln, der angibt, wie wichtig eine Verbesserung dessen ist. Dieser Wert soll sowohl jeweils für die einzelnen Kriterien der Data-Clumps als auch als Gesamtwert - an welchem sich das anschließende Refactoring orientiert - einsehbar sein. Wie die einzelnen Werte berechnet werden hängt stark von dem jeweiligen Kriterium ab. Der Gesamtwert ergibt sich aus den Werten der einzelnen Kriterien.

3.5.1 Die Ziele der Bewertung

Das Ziel der Berechnung des Gesamtwertes soll es sein, für jeden Data-Clump eine Orientierung beziehungsweise Abschätzung zu geben, wie komplex dieser ist und ob sich ein mögliches Refactoring überhaupt lohnt. Die Ausgabe ist folglich eine aus den einzelnen Kriterien zusammengesetzte Empfehlung, ob der Data-Clump überarbeitet werden sollte oder der Datenklumpen notwendigerweise existent bleiben muss. Dabei wird für jedes verwendete Kriterium ein Wert zwischen 0 und 100 berechnet. Zusammengesetzt und durch die Anzahl der für die Bewertung relevanten Kriterien dividiert ergeben diese Teilwerte den Gesamtwert des jeweiligen Data-Clumps.

$$Wert_{Gesamt} = \frac{K_1+K_2+K_3+K_4+K_5}{Anzahl_{Bewertungskriterien}}$$

Da alle Kriterien gleich gewichtet sind, ergibt sich der Gesamtwert aus dem Durchschnitt der fünf Kriterien K_1 bis K_5. Je höher der jeweilige Teilwert ist, desto notwendiger ist ein Refactoring für diesen Data-Clump nach dem zugrundeliegenden Kriterium.

3.5.2 Die Ausgabe der Bewertung

Die Ausgabe der Bewertung soll ein prozentualer Wert pro Data-Clump sein. Dabei gibt 0 an, dass der jeweilige Data-Clump sich nicht für ein Refactoring eignet und dieses sich auf Grund der der Bewertung zugrundeliegenden Kriterien nicht lohnt oder möglicherweise gar nicht möglich ist. Wichtig ist hier zu beachten, dass die Prüfung der reinen Möglichkeit eines Refactorings nicht Gegenstand der Arbeit ist und daher an dieser Stelle nicht detaillierter behandelt wird. Der Wert 100 hingegen gibt an, dass ein Data-Clump dringend beseitigt werden muss. Auch kann es zu Null-Werten - etwa auf Grund fehlender Daten - innerhalb der Kriterien kommen, welche in diesem Fall nicht als 0 interpretiert werden, da diese Bewertung das Ergebnis verfälschen kann. Stattdessen wird das nicht bewertbare Kriterium nicht mit in die Berechnung aufgenommen, sodass der Gesamtwert nur durch vier der fünf Kriterien berechnet wird. Der entsprechende Teilwert wird mit 0 in der Visualisierung angegeben.

Wichtig zu beachten ist außerdem, dass es sich bei der Bewertung lediglich um eine Empfehlung für das Refactoring handelt und technische Details bezüglich der Umsetzung des Refactorings hier bewusst außer Acht gelassen werden. Außerdem können mit

Hilfe dieses Wertes auch Data-Clumps desselben Projekts untereinander verglichen und relativ zueinander bewertet werden. Die Ausgabe der Werte - sowohl der Gesamtwert als auch die einzelnen Werte der Kriterien - sollen entweder über die Konsole oder das User Interface einsehbar sein oder in einer Datei im JSON-Format zusammen mit den Data-Clumps zurückgeschrieben werden. Dabei dienen die ersten beiden Methoden der Visualisierung sowie des Abgleiches der Werte, das Schreiben in eine JSON-Datei ermöglicht hingegen die weitere Verarbeitung der Daten innerhalb der zukünftig automatisierten Pipeline. Damit dient diese Datei als Eingabe für das anschließende automatisierte Refactoring.

3.5.3 Gewichtung der Kriterien

Abschließend sollen die einzelnen Kriterien gewichtet werden können, sodass nicht alle Eigenschaften gleichermaßen in die Bewertung einfließen. Dies kann viele Vorteile haben, da sich die Berechnung so an den konkreten Anwendungsfall besser anpassen lässt oder wenn die Datenlage für einige Kriterien bei anderen Projekten nicht ausreichend ist. Ebenfalls kann die Relevanz der Kriterien besser bestimmt werden. So ist etwa die Anzahl der Methoden, in denen der Data-Clump vorkommt, wesentlich einflussreicher für die Bestimmung der Notwendigkeit des Refactorings als die Wortlänge eines Parameters. Für die Gewichtung enthält jedes Kriterium ebenfalls einen prozentualen Wert zwischen 0 und 100, welcher als Wert zwischen 0 und 1 ($Gewicht_G = 100/10 = 1.00$) angegeben wird. Bei dem Wert 0 wird das entsprechende Kriterium nicht berücksichtigt – beispielsweise bei nicht bewertbaren Kriterien – und bei 1 fließt der errechnete Punktestand vollständig in die Berechnung des Gesamtwertes des Data-Clumps ein. Durch diese Gewichtung sind auch Abstufungen zwischen den einzelnen Kriterien möglich, sodass diese relativ zueinander bewertet werden können. Damit ergibt sich folgende Formel zur Berechnung des Gesamtwertes eines Data-Clumps hinsichtlich der Notwendigkeit eines Refactorings mit fünf Kriterien:

$$Ergebnis_{Gewichtet} = \frac{K_1*G_1+K_2*G_2+K_3*G_3+K_4*G_4+K_5*G_5}{G_1+G_2+G_3+G_4+G_5}$$

4 Umsetzung

Im zweiten Teil der Arbeit wird die konkrete Umsetzung des konzeptionellen Teils anhand des open-source Projekts GanttProject thematisiert. Dabei werden die durch den Data-Clump Detektor gefundenen Data-Clumps hinsichtlich der in Abschnitt 3.4 genannten Kriterien Anforderungen, Komplexität, Superklassen, Anzahl und Typen der Parameter, Verbreitung analysiert und die Ergebnisse sowohl innerhalb des Programms visualisiert als auch in einer Datei im JSON-Format gespeichert. Auch Probleme, Schwierigkeiten sowie etwaige Fehlerbehandlungen werden am Ende des zweiten Hauptteils - der Umsetzungsdokumentation - genauer beschrieben.

4.1 Idee und Werkzeuge

Das Programm für die automatisierte Bewertung von Data-Clumps wird innerhalb der Softwareplattform .Net 8.0 von Microsoft mit Hilfe der Programmiersprache C# und dem *User Interface* (UI)-Framework *Windows Presentation Foundation* (WPF) erstellt. .Net ist eine plattformübergreifende open-source Entwicklerplattform, welche verschiedene Komponenten wie eine Runtime, Bibliotheken sowie Compiler beinhaltet, welche für die Entwicklung eines Programms unabdingbar sind [29]. Auch wenn .Net mehrere Programmiersprachen wie F# oder Visual Basic beinhaltet, wird für das vorliegende Projekt die gängigste Sprache C# gewählt. Eine geeignete Benutzeroberfläche wird mit Hilfe des von Microsoft zur Verfügung gestellten WPF-Frameworks erstellt. Die Windows Presentation Foundation ist auflösungsunabhängig und verwendet eine vektorbasierte Rendering-Engine mit *Extensible Application Markup Language* (XAML)-Seiten für die Darstellung der einzelnen Ansichten [30]. Außerdem werden verschiedene Pakete des NuGet Paketmanagers [31] verwendet, um diverse Prozesse, wie beispielsweise das Übersetzen der JSON-Objekte in C#-Objekte, zu vereinfachen. Die genaue Vorgehensweise der Übersetzung wird in Kapitel 4.3 detailliert erläutert.

Grundsätzlich soll das Programm nach dem Start durch einen angegebenen Dateipfad die notwendigen Dateien - die Liste der Data-Clumps sowie die Tracing-Daten - laden und in entsprechende C#-Objekte übersetzen. Jeder Data-Clump beinhaltet somit als C#-Objekt seine Beziehungen zu den Tracing-Daten in Form von Listen als Objekt-Attribute. Über diese Listen können die Data-Clumps mit allen Attributen verknüpft werden und die Werte können anhand dieser für jeden Data-Clump berechnet werden. Folglich besitzt jedes Data-Clump-Objekt Attribute sowohl für die Werte der Kriterien als auch für den entsprechenden Gesamtwert. Während die Attribute der Data-Clump-Objekte über eine Liste angezeigt werden können, können diese ebenfalls zurück in das

JSON-Format übersetzt und in eine neue Datei geschrieben werden, welche an einen angegebenen Ort im Filesystem gespeichert werden kann. Somit ergibt sich eine sortierte Liste der Data-Clumps nach den Werten der Eignung eines Refactorings.

4.2 Grundlegende Architektur und sekundäre Anforderungen

In diesem Abschnitt wird die grundlegende Architektur des Programms beschrieben und auf weitere Anforderungen an das Programm zur automatisierten Bewertung von Data-Clumps anhand von Tracing-Daten eingegangen.

4.2.1 Model-View-ViewModel

Grundlegend wird das *Model-View-ViewModel* (MVVM)-Pattern benutzt, welches das Programm in drei verschiedene Ebenen aufteilt. Die Model-Ebene beinhaltet dabei die wichtigen Programmstrukturen wie beispielsweise die Klasse der Data-Clumps oder die Berechnung der Werte pro Data-Clump nach den jeweiligen Kriterien. Die View-Ebene stellt die Benutzeroberfläche samt Textboxen, Schaltflächen, Tabellen und Eingabefeldern dar. Die Viewmodel-Ebene bildet die Verbindung zwischen Model und View und sorgt dafür, dass in der View getätigte Eingaben an das Model weitergeleitet werden und Änderungen am Model in der View für den Benutzer sichtbar sind. Diese Trennung der Ansicht und Logik sorgt dafür, dass das Programm auch ohne die verwendete Benutzeroberfläche über die Konsole ausführbar ist und keine Abhängigkeiten zwischen Model und View bestehen. Außerdem können damit theoretisch mehrere Benutzeroberflächen einfach verändert werden, ohne dass dies einen direkten Einfluss auf das Model hat [32] [33]. Für die Umsetzung im .Net Kontext wird das MVVM Community-Toolkit als Paket verwendet [34] [35].

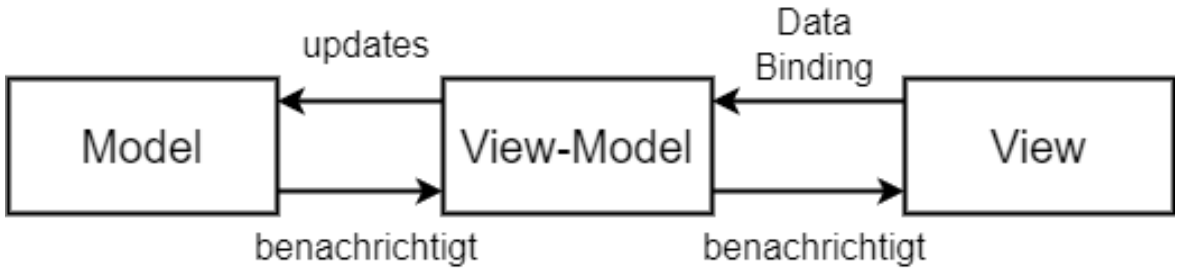


Abbildung 7: Aufbau des Model-View-ViewModel-Patterns

4.2.2 Persistente Optionen

Eine weitere Anforderung an das Programm sind speicherbare Optionen. Als Optionen werden hier beispielsweise die Dateipfade zu den Tracing-Daten, der Data-Clump-Liste oder der Ausgabe für die sortierte Data-Clump-Liste angesehen. Aber auch weitere

Aspekte wie die Anzeige der Data-Clumps innerhalb des Programms oder die in Kapitel 3.5.3 beschriebene Gewichtung der einzelnen Kriterien für die Bewertung der Data-Clumps können unter den Optionen zusammengefasst werden. Persistente Optionen sind dahingehend wichtig, dass gerade bei einer großen Datenmenge Zeit und Ressourcen für die wiederholte Eingabe der Parameter gespart werden soll. So soll das Programm beim Start der Berechnung die notwendigen Parameter aus den gespeicherten Optionen ableiten. Für die persistente Speicherung von Optionen gibt es verschiedene Möglichkeiten, wie den Einsatz von Appsettings oder Datenbanken. In dem vorliegenden Fall werden die speicherbaren Optionen über einen Speicher-Button in eine Datei im JSON-Format innerhalb des Projektordners geschrieben. Vor dem Start der Berechnung kann der Nutzer somit entscheiden, ob die in dieser Datei gespeicherten Daten geladen werden oder eigene Eingaben in den dafür vorgesehenen Feldern getätigt werden sollen. Die Übersetzung der Optionen in das JSON-Format und zurück in C-#-Attribute wird im nachfolgenden Kapitel genauer erläutert.

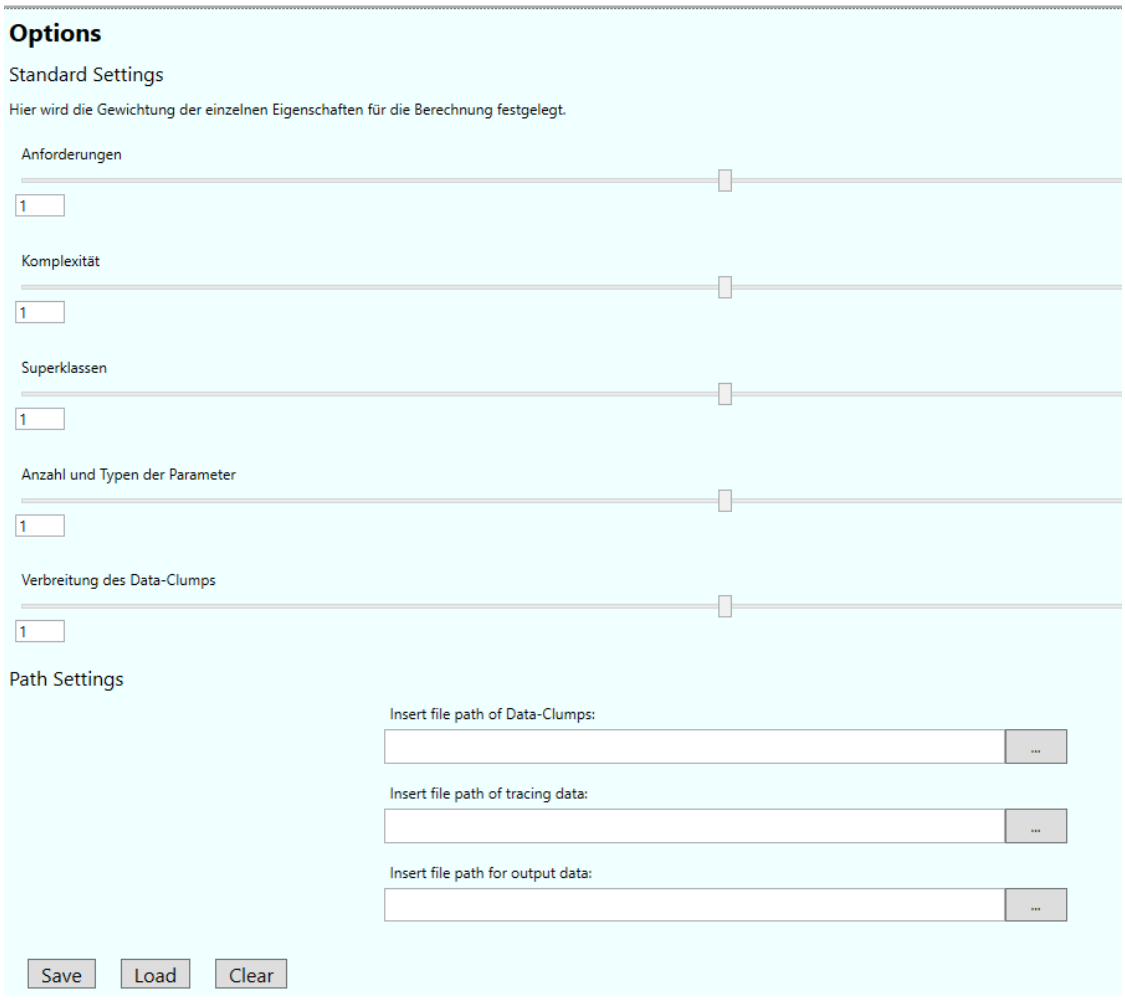


Abbildung 8: Die Optionen des Programms samt Möglichkeit des Speicherns und Ladens

Die Optionen werden in folgendem Format in einer options.json-Datei im Programmverzeichnis gespeichert und können somit auch direkt in der Datei manipuliert werden. Das von der grafischen Benutzeroberfläche unabhängige Programm *DataClumpRating.Core* benötigt entweder in diesem Format angegebene Optionen oder erwartet bei Ausführung entsprechende Parameter. Für die Parameter `PathDataClumps`, `PathOutputData`, `PathTracingData` sind die absoluten Dateipfade anzugeben.

```
{
  "OptionsEigenschaft1SliderValue":1.0,
  "OptionsEigenschaft2SliderValue":1.0,
  "OptionsEigenschaft3SliderValue":1.0,
  "OptionsEigenschaft4SliderValue":1.0,
  "OptionsEigenschaft5SliderValue":1.0,
  "PathDataClumps":"",
  "PathOutputData":"",
  "PathTracingData":"",
}

```

Listing 11: Auszug aus der Datei options.json

4.3 Parser und Dateiformate

Um die im JSON-Format vorliegenden Data-Clumps innerhalb des Programms gut verarbeiten zu können, müssen diese in Objekte umgewandelt werden. Für die Deserialisierung zu Objekten und die auf die Bewertung folgende Serialisierung in das JSON-Format wird das NuGet Paket *Newtonsoft.Json* verwendet [36].

4.3.1 JSON zu C#-Objects

Für die Deserialisierung der Data-Clumps und Tracing-Daten stellt das Newtonsoft.Json Paket die Methode JsonConvert.DeserializeObject<T>() zur Verfügung. Durch die Methode werden die Attribute der Objekte im JSON-Format auf die Attribute des C#-Objekts zugewiesen. Dafür müssen die Attribute der Objekte innerhalb des Programms genau wie in der JSON-Form benannt werden. Andernfalls müssen die Attribute entsprechend gesondert sortiert und verbunden werden, was den Übersetzungsaufwand erhöhen würde. Eine mögliche Funktion zur Übersetzung der classes.json Datei ist in Listing 12 dargestellt.

```
public static List<TracingClasses>
LoadClassesFromJSONToList(string filePath)
{

  string classesFromJSON =
  File.ReadAllText(filePath+"/classes.json");
```

```
7
8           List<TracingClasses> classesList =
9           JsonConvert.DeserializeObject<List<TracingClasses>>
10          (classesFromJSON);
11
12          return classesList;
13
14      }
15
```

Listing 12: Die Methode LoadClassesFromJSONToList()

Die Methode LoadClassesFromJSONToList bekommt den Pfad zur classes.json Datei als Parameter filePath übergeben. Durch den Aufruf der Methode File.ReadAllText wird der Inhalt der classes.json Datei in einen einzigen String classesFromJSON geschrieben. Die JsonConvert.DeserializeObject()-Methode lädt – im Gegensatz zu der Methode JsonConvert.Deserialize() – die einzelnen Objekte der JSON-Datei in die generische Liste classesList, welche aus Objekten der TracingClasses Klasse besteht. Ein Objekt der Klasse TracingClasses enthält dabei alle Informationen, die in der JSON-Datei für ein entsprechendes Objekt vorgesehen sind. Analog dazu werden auch die übrigen Tracing-Daten sowie die Data-Clumps in geeignete C-#-Objekte übersetzt.

4.3.2 C#-Objects zu JSON

Um die Data-Clump Objekte aus der Data-Clump-Liste wieder in eine JSON-Datei zu schreiben, wird die JsonConvert.SerializeObject()-Methode aus dem Newtonsoft.Json Paket [36] verwendet, indem in der StartCalculation()-Methode SerializeDataClumpObjectsToJson() aufgerufen wird. Diese Methode bekommt sowohl die Liste der Data-Clumps als auch den Ausgabepfad, welcher auf Grund des Aufrufs des Windows-Dialogs ebenfalls den Dateinamen beinhaltet, übergeben. Der zweite Parameter der Funktion Formatting.Intendet bewirkt, dass der im String dataClumpString gespeicherte Text in der finalen JSON-Datei die für das JSON-Format übliche Formatierung erhält. Um nicht alle Attribute der Data-Clumps, wie beispielsweise die zugehörigen Listen, in denen die Verknüpfungen der Tracing-Daten angegeben sind, in die Ergebnis-Datei zu schreiben, werden diese Attribute mit der Annotation [JSONIgnore] versehen. Mit dem Tag [JsonProperty=()] kann ein Attributname, wie die der einzelnen Kriterien, für das Ergebnis verändert werden. Beide Annotationen sind Teil des Newtonsoft.Json Pakets und ermöglichen die Steuerung der Serialisierung. Mit Hilfe der WriteAllText()-Methode wird eine neue Textdatei in dem angegebenen Verzeichnis erstellt und mit dem bereits formatierten String befüllt. Nachfolgend ist die Methode SerializeDataClumpObjectsToJson() abgebildet.

```
1       public void SerializeDataClumpObjectsToJson
2         (List<DataClump> dataClumpList,
3         string outputDataString)
4       {
```

```
5           string dataClumpString = JsonConvert.SerializeObject
6           (dataClumpList, Formatting.Indented);
7
8           File.WriteAllText(outputDataString, dataClumpString);
9       }
10
```

Listing 13: Die Methode SerializeDataClumpObjectsToJson

4.4 User Interface

Sowohl für eine bessere Übersicht als auch zum Testen der Tracing-Daten enthält dieses Programm eine mit dem bereits vorgestellten UI-Framework WPF implementierte Benutzeroberfläche. Am linken Fensterrand befindet sich ein Menü, mit welchem zwischen den einzelnen Fenstern gewechselt werden kann. Dies beinhaltet die Menüpunkte Home, Data-Clump rating, Options, Results, Save and Quit. Der Home-Button führt zum Startbildschirm und der Save and quit-Button beendet das Programm. Unter Options können die Pfade der Tracing-Daten, Data-Clumps und der Ausgabepfad für die Ergebnis-Datei angegeben und für die weitere Nutzung gespeichert werden. Außerdem ist die Gewichtung der einzelnen Bewertungskriterien - welche ebenfalls gespeichert werden können - an dieser Stelle möglich. Die Tabelle der Ergebnisse wird unter Results angezeigt und kann mit dem Refresh-Button aktualisiert werden. Die eigentliche Bewertung der Data-Clumps anhand der vorliegenden Tracing-Daten kann im Reiter Data-Clump rating nach dem Laden oder Einfügen der Dateipfade mit dem Button Calculate gestartet werden.

4.5 Die Klasse Data-Clump

Die Data-Clump-Klasse bildet den Bauplan für die einzelnen Data-Clump-Objekte. Dabei bildet jedes Objekt einen einzelnen Data-Clump ab und besitzt folgende Attribute, welche unmittelbar bei der Erstellung mit den Werten des Data-Clump-Detektors gefüllt werden.

```
1       String Type;
2       String Key;
3       Int Probability;
4       String From\_File\_Path;
5       [...]
6
```

Listing 14: Auszug der Attribute der Data-Clump Klasse, welche unmittelbar befüllt werden

Zusätzlich werden für jeden Data-Clump die Attribute für die Ergebnisse der Berechnung sowie eine eindeutige Id erstellt. Die EigenschaftScores beinhalten die Teilergebnisse und das Attribut RefactorScore – das Gesamtergebnis pro Data-Clump.

```
1       Int Id;
2       Double Eigenschaft1Score;
3       Double Eigenschaft2Score;
4       Double Eigenschaft3Score;
5       Double Eigenschaft4Score;
6       Double Eigenschaft5Score;
7       Double RefactorScore;
8
```

Listing 15: Die Attributnamen der berechneten Werte pro Data-Clump

Durch den Import der Tracing-Daten werden zusätzlich den folgenden Attributen entsprechende Werte zugewiesen:

```
1       List<DataClumpData> DataClumpDataList;
2       List<TracingRequirements> TracingRequirementsList;
3       List<TracingClasses> TracingClassesList;
4       List<TracingMethods> TracingMethodsList;
5       List<TracingParameters> TracingParametersList;
6       List<Traces> TracesList;
7       List<TracingSuperClasses> TracingSuperClassesList;
8       List<TracingSuperClasses> TracingChildClassesList;
9
```

Listing 16: Die zu befüllenden Listen, in denen die Verbindungen der Tracing-Daten gespeichert werden

Dabei verbinden die Listen jeweils Objekte der Tracing-Dateien. So beinhaltet beispielsweise die Liste List<TracingClasses> TracingClassesList alle Klassen als einzelne Objekte der TracingClasses-Klasse, welche über die Klassen-Id diesem Data-Clump zugeordnet werden können. Analog dazu werden die anderen Listen mit den Tracing-Daten befüllt. Folglich sind jedem Data-Clump Listen der Traces der Tracing-Daten zugeordnet, welche für die Verknüpfung und Analyse von Abhängigkeiten unabdingbar sind.

Des Weiteren besitzt jedes Objekt der Data-Clump-Klasse Methoden zur Berechnung der Kriterien, welche im folgenden Abschnitt beschrieben werden.

4.6 Der Bewertungsalgorithmus und die Berechnung

Die eigentliche Bewertung der Data-Clumps findet in der Calculation-Klasse mit der Methode StartCoreCalculation() statt. Innerhalb der Methode wird zunächst ein neues Objekt der Klasse Calculation mit den notwendigen Dateipfaden und den Gewichtungen der Kriterien erstellt. Durch den Aufruf der Methode FillTracingDataLists() werden die globalen Listen der Traces mit den verschiedenen Tracing-Daten gefüllt. Das nachfolgende Listing zeigt, wie die globalen Listen der Tracing-Daten durch die Methode FillTracingDataLists() gefüllt werden. Dabei wird die LoadFromJSONToList-Methode der jeweiligen Tracing-Daten-Klasse – wie in Kapitel 4.3.1 – aufgerufen.

```
1       public void FillTracingDataLists()
2       {
3         GlobalTracingData.DataClumpListGLobal = DataClump.
4         LoadDataCLumpsFromJSONToList(DataCLumpPath);
5
6         GlobalTracingData.TracingClassesListGLobal =
7         TracingClasses.
8         LoadClassesFromJSONToList(TracingDataPath);
9
10        GlobalTracingData.TracingRequirementsListGLobal =
11        TracingRequirements.
12        LoadRequirementsFromJSONToList(TracingDataPath);
13
14        GlobalTracingData.TracesListGlobal = Traces.
15        LoadTracesFromJSONToList(TracingDataPath);
16
17        GlobalTracingData.TracingMethodsListGlobal =
18        TracingMethods.
19        LoadMethodsFromJSONToList(TracingDataPath);
20
21        GlobalTracingData.TracingSuperClassesListGlobal =
22        TracingSuperClasses.
23        LoadSuperClassesFromJSONToList(TracingDataPath);
24
25        GlobalTracingData.TracingParametersListGlobal =
26        TracingParameters.
27        LoadParametersFromJSONToList(TracingDataPath);
28      }
29
```

Listing 17: Die Methode FillTracingDataLists der Klasse Calculation

Die Methode FillDataClumpListsWithTracingData() befüllt die einzelnen Listen der Data-Clumps mit den entsprechenden Traces. So wird die Liste der Data-Clumps durchlaufen und anhand der Abhängigkeiten aus Abbildung 4 erweitert. Im folgenden Beispiel werden die betroffenen Methoden eines Data-Clumps ermittelt, indem der Name der Methode in der Data-Clump Datei mit dem Attribut *From_Method_Name* mit dem Attribut *MethodRefinedName* der Tracing-Methoden verglichen wird. Alle übereinstimmenden Elemente werden somit in der für Methoden vorgesehenen Liste jedes Data-Clumps gespeichert.

```
1       dataClumpItem.TracingMethodsList =
2       GlobalTracingData.TracingMethodsListGlobal
3       .Where(e => e.MethodNameRefined == dataClumpItem.
4       From\_Method\_Name).ToList();
5
```

Listing 18: Auszug aus der Methode FillDataClumpListsWithTracingData() zum Zuordnen der Methoden eines Data-Clumps

Für den Fall, dass sich die Start- und Zielmethoden des Data-Clumps unterscheiden, müssen mehrere Verknüpfungen beachtet werden. Somit wird folgender Zusatz zur Methode FillDataClumpListsWithTracingData() hinzugefügt:

```
1       if (dataClumpItem.From\_Method\_Name != dataClumpItem.
2         To\_Method\_Name){
3         List<TracingMethods> liste = GlobalTracingData.
4         TracingMethodsListGlobal
5         .Where(e => e.MethodNameRefined == dataClumpItem.
6         To\_Method\_Name).ToList();
7         dataClumpItem.TracingMethodsList.AddRange(liste);
8       }
9
```

Listing 19: Erweiterung der Methode FillDataClumpListsWithTracingData()

Nachdem alle Listen der Data-Clumps vollständig befüllt sind, wird die Berechnung ausgeführt und die Ergebnisse werden in die dafür vorgesehenen Attribute der Data-Clumps sowie in die JSON-Ergebnis-Datei geschrieben. Die Berechnung orientiert sich dabei größtenteils an den im Konzept vorgestellten Methoden. Wichtig ist hier jedoch die Unterscheidung der Data-Clump Typen. So können Parameter-Data-Clumps über die Methoden und Parameter die erforderlichen Listen füllen und Berechnungen anhand der vorgesehenen Variablen durchführen; Field-Data-Clumps hingegen können an dieser Stelle nur unzureichend über die Methoden verknüpft werden. Folglich müssen speziell für diese Art der Data-Clumps die Klassen genutzt werden. Auch die Nutzung der Felder für die Berechnung der Komplexität ist auf Grund fehlender Tracing-Daten mit der gewählten Methode nicht möglich. Daher kann das Kriterium der Komplexität für Field-to-Field-Data-Clumps nicht berücksichtigt werden. Der erste Field-Data-Clump beispielsweise hat seinen Ursprung in der Klasse BottomUnitLineRendererImpl mit der Klassen-Id 156. Diese lässt sich jedoch nicht in den für Felder vorgesehenen Tracing-Daten finden. Somit ist die automatisierte Bewertung von Field-Data-Clumps an dieser Stelle mit den vorliegenden Tracing-Daten nicht möglich. Die übrigen Kriterien lassen sich wie im konzeptionellen Abschnitt beschrieben umsetzen. Eine entsprechende Fehlerbehandlung für beispielsweise null-Rückgaben wurde ebenfalls implementiert.

4.7 Ausgabe der bewerteten Data-Clumps

Wie in den Anforderungen an das Programm bereits geschildert gibt es drei verschiedene Möglichkeiten zur Ausgabe der nach Notwendigkeit eines Refactorings sortierten Liste der Data-Clumps: die Ausgabe als Text über die Konsole, die Visualisierung innerhalb des Programms in der Ergebnisübersicht sowie der Export in eine gesonderte Datei im JSON-Format. Die Ausgabe über die Konsole erfolgt dabei durch die Auflistung der einzelnen Data-Clumps samt der Attribute.

4.7.1 Visualisierung innerhalb des Programms

Die Visualisierung der Ergebnisse innerhalb der Data-Clumps findet mit Hilfe einer Tabelle statt, welche die Data-Clump-Liste darstellt. Diese Tabelle kann im Anschluss an die Berechnung unter dem Tab Results erstellt werden. Jede Zeile bildet dabei einen Data-Clump ab. Die Spalten orientieren sich an den Attributen der Data-Clump-Objekte. In der Tabelle werden bewusst mit der Option AutoGenerateColumns=False nur ausgewählte, die für die Ergebnisse wichtigen Attribute, wie die einzelnen Werte der Bewertungskriterien und der Gesamtwert sowie die Id und der Data-Clump-Type, angezeigt. Damit alle Data-Clumps mit den entsprechenden Attributen für den Benutzer sichtbar sind, werden diese in der View-Ebene in eine ScrollView mit den XAML-Tags <ScrollViewer></ScrollViewer> eingebettet, um sowohl vertikales als auch horizontales Scrollen innerhalb der Tabelle - des DataGrids - zu ermöglichen. Mit Hilfe eines innerhalb der Spalten eingebunden ValueConverters können einzelne Werte unterschiedlich farblich hinterlegt werden. Alle Zellen mit Werten kleiner als 25 werden grün hinterlegt, Werte zwischen 50 und 75 gelb und Werte ab 75 rot. Dies soll zur besseren Übersicht der Ergebnisse beitragen, da höhere Werte der Skala von 0 bis 100 somit direkt erkennbar sind.

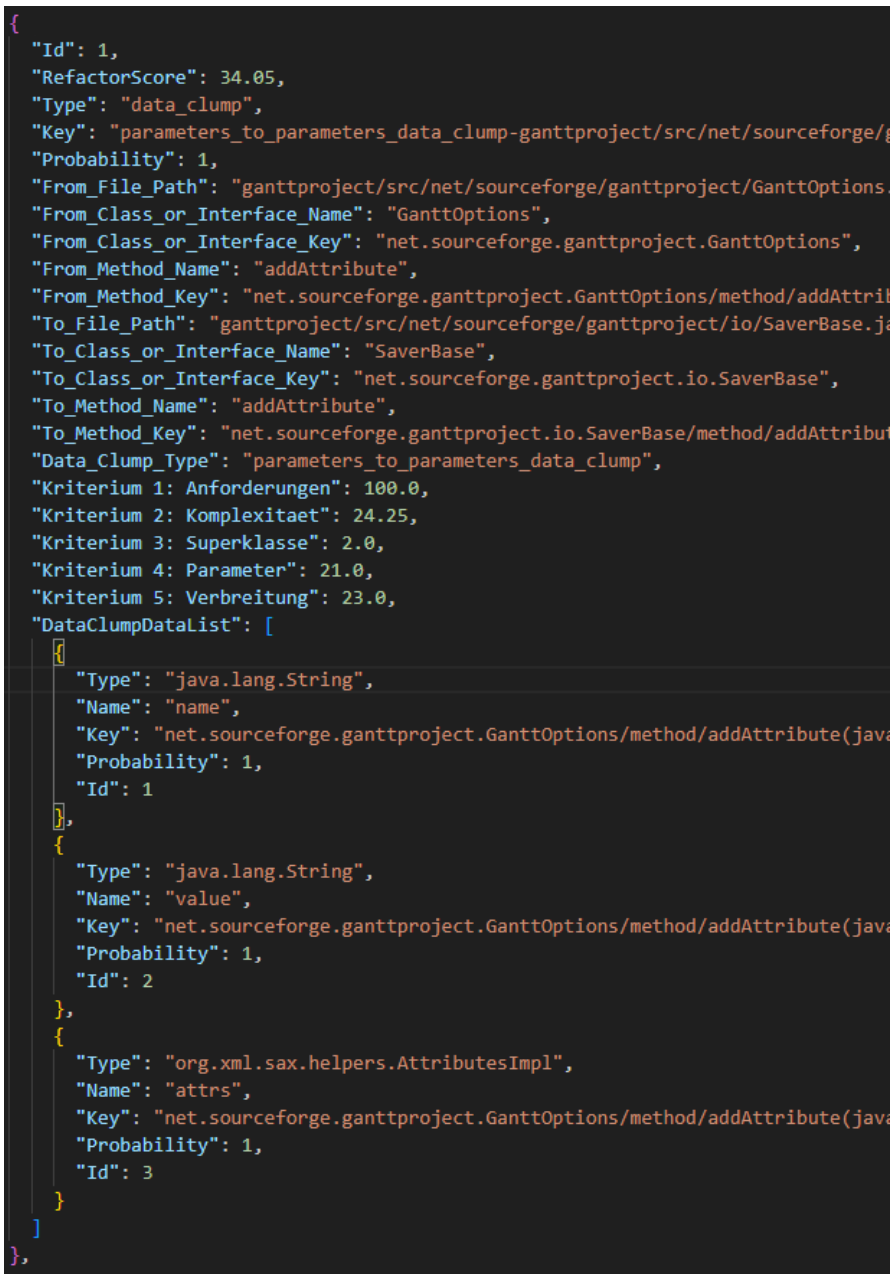
ID	Typ	Anforderungen	Komplexität	Superklasse / Subklasse	Anzahl und Typen der Parameter	Verbreitung	RefactorScore (gesamt)
1	parameters_to_parameters_data_clump	100	24.25	2	21	23	34.05
2	parameters_to_parameters_data_clump	100	24.25	2	21	13	32.05
3	parameters_to_parameters_data_clump	100	21.6	2	27	22	34.52
4	parameters_to_parameters_data_clump	100	100	2	27	48	55.4
5	parameters_to_parameters_data_clump	100	24.25	2	27	23	35.25
6	parameters_to_parameters_data_clump	100	21.6	2	27	12	32.52
7	parameters_to_parameters_data_clump	100	21.6	2	27	12	32.52
8	parameters_to_parameters_data_clump	100	21.6	2	27	12	32.52
9	parameters_to_parameters_data_clump	100	100	2	27	38	53.4

Abbildung 9: Ausschnitt der Visualisierung der Ergebnisse innerhalb eines DataGrids

Mit Hilfe des Buttons Adjust values können darüber hinaus an dieser Stelle die Werte entsprechend der Data-Clumps innerhalb des Projekts angepasst werden. So werden die höchsten und niedrigsten Werte ermittelt und alle Werte anhand dieser Berechnung mit einem Faktor multipliziert, sodass alle Data-Clumps den Bereich von 0 bis 100 entsprechend dem höchsten und niedrigsten Data-Clump abbilden. Dies ermöglicht einen besseren Vergleich der Data-Clumps untereinander innerhalb desselben Projekts.

4.7.2 Ausgabe im JSON-Format

Für die Ausgabe im JSON-Format wird die Data-Clump-Liste mit den neuen Attributen - die einzelnen Kriterien sowie der Gesamtwert - nach der Bewertung mit Hilfe der in Abschnitt 4.3 beschriebenen Übersetzer als JSON-String in eine neue Datei geschrieben. Dafür wird eine neue Datei an der Stelle des angegebenen Ausgabepfads erstellt und mit der Liste der Data-Clumps als Zeichenkette befüllt. Die für das JSON-Format übliche Strukturierung innerhalb der Datei erzeugt dabei der Befehl Format.Intended. Die relevanten Attribute werden entsprechend des in Abschnitt 4.3.2 genannten Verfahrens gefiltert.



```
{
  "Id": 1,
  "RefactorScore": 34.05,
  "Type": "data_clump",
  "Key": "parameters_to_parameters_data_clump-ganttproject/src/net/sourceforge/
  "Probability": 1,
  "From_File_Path": "ganttproject/src/net/sourceforge/ganttproject/GanttOptions
  "From_Class_or_Interface_Name": "GanttOptions",
  "From_Class_or_Interface_Key": "net.sourceforge.ganttproject.GanttOptions",
  "From_Method_Name": "addAttribute",
  "From_Method_Key": "net.sourceforge.ganttproject.GanttOptions/method/addAttri
  "To_File_Path": "ganttproject/src/net/sourceforge/ganttproject/io/SaverBase.j
  "To_Class_or_Interface_Name": "SaverBase",
  "To_Class_or_Interface_Key": "net.sourceforge.ganttproject.io.SaverBase",
  "To_Method_Name": "addAttribute",
  "To_Method_Key": "net.sourceforge.ganttproject.io.SaverBase/method/addAttribu
  "Data_Clump_Type": "parameters_to_parameters_data_clump",
  "Kriterium 1: Anforderungen": 100.0,
  "Kriterium 2: Komplexitaet": 24.25,
  "Kriterium 3: Superklasse": 2.0,
  "Kriterium 4: Parameter": 21.0,
  "Kriterium 5: Verbreitung": 23.0,
  "DataClumpDataList": [
    {
      "Type": "java.lang.String",
      "Name": "name",
      "Key": "net.sourceforge.ganttproject.GanttOptions/method/addAttribute(java
      "Probability": 1,
      "Id": 1
    },
    {
      "Type": "java.lang.String",
      "Name": "value",
      "Key": "net.sourceforge.ganttproject.GanttOptions/method/addAttribute(java
      "Probability": 1,
      "Id": 2
    },
    {
      "Type": "org.xml.sax.helpers.AttributesImpl",
      "Name": "attrs",
      "Key": "net.sourceforge.ganttproject.GanttOptions/method/addAttribute(java
      "Probability": 1,
      "Id": 3
    }
  ]
},
```


Abbildung 10: Auszug der Ergebnisse im JSON-Format

4.7.3 Ausgabe über die Konsole

Die Ausgabe in der Konsole erfolgt über die Console.Writeline()-Methode. Dabei wird mit mehreren foreach-Schleifen durch die Liste der Data-Clumps sowie deren Attribute iteriert und diese formatiert ausgegeben. Als vereinfachte Ausgabe dienen dabei die einzelnen Werte der Kriterien sowie der Gesamtwert, die ID, der Typ und der Schlüssel des jeweiligen Data-Clumps. Die Ausgabe eines Data-Clumps sieht wie folgt aus:

```
Data-Clump Nr. 1:
Type: parameters_to_parameters_data_clump
Anforderungen: 100
Komplexität: 24,25
Superklasse / Anzahl Ownerklassen: 2
Anzahl und Typen des Data-Clumps: 21
Verbreitung des Data-Clumps: 23
Gesamtbewertung: 34,05
-----------------------------------------

Data-Clump Nr. 2:
Type: parameters_to_parameters_data_clump
Anforderungen: 100
Komplexität: 24,25
Superklasse / Anzahl Ownerklassen: 2
Anzahl und Typen des Data-Clumps: 21
Verbreitung des Data-Clumps: 13
Gesamtbewertung: 32,05
-----------------------------------------

Data-Clump Nr. 3:
Type: parameters_to_parameters_data_clump
Anforderungen: 100
Komplexität: 21,6
Superklasse / Anzahl Ownerklassen: 2
Anzahl und Typen des Data-Clumps: 27
Verbreitung des Data-Clumps: 22
Gesamtbewertung: 34,52
-----------------------------------------
```

Abbildung 11: Auszug aus der Ausgabe der bewerteten Data-Clumps über die Konsole

4.8 Hindernisse und Fehlerbehandlung sowie Ausblick auf die Verarbeitung generischer Tracing-Daten

Das größte Hindernis bei der automatisierten Bewertung von Data-Clumps anhand von Tracing-Daten sind die Tracing-Daten selbst, da der Erfolg der Analyse hinsichtlich der erarbeiteten Kriterien hauptsächlich von diesen als Bewertungsgrundlage abhängt. Ohne die entsprechenden und vollständigen Tracing-Daten eines Projekts ist eine etwaige Analyse sowie Bewertung von Data-Clumps nicht möglich. Auch die Auswahl des Projektes gestaltet sich auf Grund der zugehörigen Datenlage sowie diversen Anforderungen an das Projekt selbst als besonders begrenzt, da viele Projekte nicht öffentlich zur Verfügung stehen und die notwendigen Ressourcen nicht beinhalten. Da auch innerhalb eines Projektes Daten zu bestimmten Data-Clumps fehlen, können möglicherweise nicht alle Daten miteinander verglichen werden, was das Gesamtergebnis der Bewertung durch die Missachtung der fehlenden Daten teilweise stark beeinflussen kann.

Ein weiterer Aspekt der eigentlichen Bewertung ist die Formulierung des genauen Bewertungsziels. So kann das Refactoring für einen Data-Clump auf Grund verschiedener Faktoren besonders wichtig sein und dennoch durch andere Faktoren begrenzt werden.

Ebenso kann das Refactoring eines Data-Clumps auf Grund sehr geringer Komplexität sehr gut geeignet sein, jedoch ist dieser durch seine geringe Verbreitung nicht so dringend zu verbessern wie ein stark verbreiteter Data-Clump. An dieser Stelle ist ein Data-Clump sowohl bezüglich der Fragen nach der Notwendigkeit als auch nach der Dringlichkeit, der Eignung und der Relevanz zu bewerten. Je nach Bewertungsziel können die Ergebnisse folglich gegensätzlich sein, was eine einheitliche Bewertung stark erschwert oder gar nahezu unmöglich machen kann. Eine Lösung des Problems sieht vor, die Bewertung von dem Bewertungsziel zu trennen und einzeln zu implementieren, sodass jede Bewertung mit einem individuellen Bewertungsziel ausgeführt werden kann.

In dem Beispiel dieser Arbeit wurde die Bewertung anhand des Gantt-Projekts sowie den zugehörigen Tracing-Daten durchgeführt. Jede Bewertung ist dabei abhängig von den Attributen des Data-Clump-Detektors und der Tracing-Daten, welche von Projekt zu Projekt variieren können. Um diese Bewertung auf eine Vielzahl von Projekten anwenden zu können, muss sichergestellt werden, dass die für die Übersetzer benötigten Attribute sich in den Namen nicht unterscheiden. Hierfür müsste innerhalb der Pipeline eine Zwischenstufe implementiert werden, welche die individuellen Attribute den Attributen der Übersetzer des Programms zuordnet. Dementsprechend muss die Berechnung der einzelnen Bewertungen angepasst werden.

Die Fehlerbehandlung des Programms sieht vor, dass Kriterien, welche auf nicht vorhandenen Tracing-Daten basieren, automatisch auf 0 gesetzt und somit in der Berechnung des Gesamtwertes nicht berücksichtigt werden. Entsprechend wird auch die Gewichtung des Kriteriums ignoriert und fließt nicht in die Berechnung mit ein. Starke Ausreißer über 100 werden dahingehend verarbeitet, dass diese genau auf den Maximalwert 100 gesetzt werden, da ein höherer Wert die Notwendigkeit beziehungsweise Eignung in vollem Maße erfüllt.

Auch hängen viele Details der Berechnungen von konkreten Anforderungen ab. Sollen beispielsweise wie in Kapitel 3.1 alle möglichen Traces für die Berechnung hinzugezogen werden oder nur die durch die Entwickler garantierten Traces, was die Datenlage weiter einschränkt und eine Bewertung auf Basis der gewählten Kriterien zusätzlich erschwert.

Trotz der Hindernisse ist das Programm in der Lage, eine gegebene Liste von Data-Clumps mit Hilfe von Tracing-Daten automatisiert anhand der Kriterien Anforderungen, Komplexität, Superklassen, Anzahl und Typen der Parameter/Felder sowie Verbreitung innerhalb des Codes zu bewerten und als Ausgabe eine Prioritätenliste für die eingangs beschriebene Pipeline zur Verfügung zu stellen.

5 Zusammenfassung

Das Ziel der Arbeit ist es, die generelle Möglichkeit einer automatisierten Bewertung von Data-Clumps auf Grundlage von Tracing-Daten und des Source-Codes zu prüfen und ein entsprechendes Konzept am Beispiel des open-source GanttProject zu entwickeln.

Alles in allem zeigt sich dabei, dass die Möglichkeit einer automatisierten Bewertung von Data-Clumps anhand von Tracing-Daten von vielen verschiedenen Faktoren abhängt. Bei dem dieser Arbeit vorliegenden Beispielprojekt GanttProject sind nicht alle theoretisch möglichen Kriterien gut bewertbar. Somit müssen einzelne Kriterien ausgewählt werden, welche sich anhand der vorliegenden Tracing-Daten oder weiteren objektiven Eigenschaften ausreichend analysieren lassen. Die in dem Gantt-Projekt verwendeten Kriterien sind somit schwer zu generalisieren und auf andere Projekte zu übertragen, da auf Grund der fest implementierten Eigenschaften und Attribute das gesamte Programm verändert werden müsste. Eine komplette Implementierung aller Kriterien sowie eine dynamische Analyse der Tracing-Daten anhand derer entschieden werden kann, welche Kriterien sich für eine automatisierte Bewertung eignen, wäre zwar - durch die Datenvielfalt bedingt - schwer umzusetzen, aber dennoch denkbar. Nichtsdestotrotz können alle Data-Clumps anhand dieser Kriterien sowie den generellen Eigenschaften von Code-Smells immerhin theoretisch – entweder manuell oder bei entsprechender Datenlage automatisch – bewertet werden. Mit Hilfe zusätzlicher Technologie kann dieses Bewertungsverfahren erweitert werden, sodass viele der oben genannten Kriterien, welche für die Bewertung nicht berücksichtigt werden konnten, in den Prozess mit aufgenommen werden könnten. Somit sind sowohl dieses Verfahren als auch die Bewertungskriterien selber auch auf andere Arten von Code-Smells wie beispielsweise God-Klassen übertragbar.

Des Weiteren ist die Trennung zwischen Eignung und Notwendigkeit eines Refactorings von großer Bedeutung, da diese auf verschiedene Eigenschaften abzielen. Eine eindeutige Differenzierung ist mit einer automatisierten Bewertung jedoch nicht gänzlich möglich, da sich diese beiden Eigenschaften weder automatisch ausschließen noch vollends überschneiden. So kann das Refactoring eines Data-Clumps zwar sehr wichtig, jedoch auf Grund verschiedener Faktoren durchaus ungeeignet sein.

Ein weiteres Hindernis bei der Umsetzung der automatisierten Bewertung von Data-Clumps bildet jedoch die bereits beschriebene Datenlage, da nur sehr wenige Tracing-Daten für verschiedene Projekte zur Verfügung stehen und diese auch nicht immer vollständig sind. Daher ist eine geeignete Fehlerbehandlung notwendig und nicht jedes der speziell für dieses Projekt ausgewählten Kriterien kann für jeden einzelnen Data-Clump bewertet werden, weshalb Data-Clumps untereinander nur schwer vergleichbar sind.

Jedoch können Differenzen der Data-Clumps hinsichtlich der Kriterien Anforderungen, Komplexität, Superklassen, Typen und Anzahl der Parameter sowie Verbreitung des Data-Clumps mit Hilfe der im Rahmen dieser Bachelorarbeit entwickelten Anwendung sehr gut visualisiert und im Kontext der eingangs beschriebenen Pipeline verarbeitet werden. Das Ziel der Anwendung, eine priorisierte Liste der Data-Clumps für das Refactoring zurückzugeben, wird auch durch die verschiedenen Möglichkeiten der Ausgabe der Ergebnisse sehr gut erreicht.

Durch diese Arbeit wird trotz aller Schwierigkeiten und Hindernisse ersichtlich, dass eine automatisierte Bewertung von Data-Clumps anhand von Tracing-Daten nicht nur möglich, sondern durch die erzeugte Prioritätenliste sowohl Zeit als auch Ressourcen einer manuellen Abschätzung bezüglich eines Refactorings einspart, den Gesamtaufwand innerhalb der Pipeline maßgeblich verringert und folglich in jedem Fall sogar ökonomisch sinnvoll ist.

Literatur

[1] Baumgartner, N., *GitHub - NilsBaumgartner1994/data-clumps-doctor*. Adresse: `https://github.com/NilsBaumgartner1994/data-clumps-doctor` (besucht am 28. März 2024).

[2] bardsoftware, *GitHub - bardsoftware/ganttproject: Official GanttProject repository*. Adresse: `https://github.com/bardsoftware/ganttproject` (besucht am 25. März 2024).

[3] Baumgartner, N., Adleh, F. und Pulvermüller, E., „Live Code Smell Detection of Data Clumps in an Integrated Development Environment," in *Proceedings of the 18th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE,*, INSTICC, SciTePress, 2023, S. 64–76, ISBN: 978-989-758-647-7. DOI: 10.5220/0011727500003464.

[4] Fowler, M., *Refactoring*. Addison-Wesley Professional, Nov. 2019.

[5] Lacerda, G., Petrillo, F., Pimenta, M. und Guéhéneuc, Y. G., „Code smells and refactoring: A tertiary systematic review of challenges and observations," *Journal of Systems and Software*, Jg. 167, S. 110 610, Sep. 2020, ISSN: 0164-1212. DOI: 10.1016/j.jss.2020.110610. Adresse: http://dx.doi.org/10.1016/j.jss.2020.110610.

[6] Abbes, M., Khomh, F., Guéhéneuc, Y.-G. und Antoniol, G., „An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, On Program Comprehension," März 2011, S. 181–190. DOI: 10.1109/CSMR.2011.24.

[7] Hall, T., Zhang, M., Bowes, D. und Sun, Y., „Some Code Smells Have a Significant but Small Effect on Faults," *ACM Trans. Softw. Eng. Methodol.*, Jg. 23, Nr. 4, Sep. 2014, ISSN: 1049-331X. DOI: 10.1145/2629648. Adresse: https://doi.org/10.1145/2629648.

[8] Sjøberg, D., Anda, B. und Mockus, A., „Questioning software maintenance metrics: A comparative case study," Sep. 2012, S. 107–110, ISBN: 978-1-4503-1056-7. DOI: 10.1145/2372251.2372269.

[9] Yamashita, A. und Moonen, L., „Do developers care about code smells? An exploratory survey," Okt. 2013, S. 242–251. DOI: 10.1109/WCRE.2013.6671299.

[10] Khomh, F., Penta, M. D., Guéhéneuc, Y.-G. und Antoniol, G., „An exploratory study of the impact of antipatterns on class change- and fault-proneness.," *Empirical Software Engineering*, Jg. 17, Nr. 3, S. 243–275, 2012. Adresse: http://dblp.uni-trier.de/db/journals/ese/ese17.html#KhomhPGA12.

[11] Delchev, M. und Harun, M. F., „Investigation of Code Smells in Different Software Domains," *Full-scale Software Engineering*, S. 31, Feb. 2015.

[12] Wagner, B., *Felder -Programmierhandbuch*, Juni 2023. Adresse: `https://learn.microsoft.com/de-de/dotnet/csharp/programming-guide/classes-and-structs/fields` (besucht am 2. Apr. 2024).

[13] Zhang, M., Baddoo, N., Wernick, P. und Hall, T., „Improving the Precision of Fowler's Definitions of Bad Smells," in *2008 32nd Annual IEEE Software Engineering Workshop*, 2008, S. 161–166. DOI: 10.1109/SEW.2008.26.

[14] Baumgartner, N. und Pulvermüller, E., „The Lifecycle of Data Clumps: A Longitudinal Case Study in Open-Source Projects," in *Proceedings of the 12th International Conference on Model-Based Software and Systems Engineering - MODELSWARD*, INSTICC, SciTePress, 2024, S. 15–26, ISBN: 978-989-758-682-8. DOI: 10.5220/0012313900003645.

[15] Wagner, B., *Methodenparameter werden nach Wert übergeben. Modifizierer ermöglichen die Semantik „Übergeben-nach-Verweis", einschließlich Unterscheidungen wie schreibgeschützt und „out"-Parametern.* Nov. 2023. Adresse: `https://learn.microsoft.com/de-de/dotnet/csharp/language-reference/keywords/method-parameters` (besucht am 2. Apr. 2024).

[16] Martin, R. C., *Clean code*. Pearson Education, Aug. 2008.

[17] Wake, W. C., *Refactoring workbook*. Addison-Wesley Professional, Jan. 2004.

[18] Gotel, O., Cleland-Huang, J., Hayes, J. H. et al., „Traceability Fundamentals," in *Software and Systems Traceability*, J. Cleland-Huang, O. Gotel und A. Zisman, Hrsg. London: Springer London, 2012, S. 3–22, ISBN: 978-1-4471-2239-5. DOI: 10.1007/978-1-4471-2239-5_1. Adresse: `https://doi.org/10.1007/978-1-4471-2239-5_1`.

[19] Ziegenhagen, D., Speck, A. und Pulvermueller, E., „Expanding Tracing Capabilities Using Dynamic Tracing Data," in Feb. 2020, S. 319–340, ISBN: 978-3-030-40222-8. DOI: 10.1007/978-3-030-40223-5_16.

[20] SPANOUDAKIS, G. und ZISMAN, A., „SOFTWARE TRACEABILITY: A ROADMAP," in *Handbook of Software Engineering and Knowledge Engineering*, S. 395–428. DOI: 10.1142/9789812775245_0014. eprint: `https://www.worldscientific.com/doi/pdf/10.1142/9789812775245_0014`. Adresse: `https://www.worldscientific.com/doi/abs/10.1142/9789812775245_0014`.

[21] Hammoudi, M., Mayr-Dorn, C. und Egyed, A., *Traceability Dataset for Open Source Systems*, Zenodo, Sep. 2020. DOI: 10.5281/zenodo.4032476. Adresse: `https://doi.org/10.5281/zenodo.4032476` (besucht am 28. März 2024).

[22] Hammoudi, M., Mayr-Dorn, C., Mashkoor, A. und Egyed, A., „A Traceability Dataset for Open Source Systems," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, S. 555–559. DOI: 10.1109/MSR52588.2021.00073.

[23] Hammoudi, M., Mayr-Dorn, C., Mashkoor, A. und Egyed, A., *Data Show Case Study*, Zenodo, Jan. 2021. DOI: 10.5281/zenodo.4453526. Adresse: https://doi.org/10.5281/zenodo.4453526 (besucht am 28. März 2024).

[24] Schader, M. und Schmidt-Thieme, L., „Subklassen, Superklassen und Vererbung," in *Java™: Eine Einführung*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, S. 119–147, ISBN: 978-3-662-08044-3. DOI: 10.1007/978-3-662-08044-3_9. Adresse: https://doi.org/10.1007/978-3-662-08044-3_9.

[25] Gantt.com, *Was ist ein Gantt Diagramm?* Adresse: https://www.gantt.com/ge/ (besucht am 4. Apr. 2024).

[26] Bertolino, A., „Software Testing Research: Achievements, Challenges, Dreams," in *Future of Software Engineering (FOSE '07)*, 2007, S. 85–103. DOI: 10.1109/FOSE.2007.25.

[27] *Datasets*. Adresse: http://sarec.nd.edu/coest/datasets.html (besucht am 30. März 2024).

[28] Holbrook, E., Hayes, J. und Dekhtyar, A., „Toward Automating Requirements Satisfaction Assessment," Okt. 2009, S. 149–158. DOI: 10.1109/RE.2009.10.

[29] Gewarren, *Introduction to .NET - .NET*, Jan. 2024. Adresse: https://learn.microsoft.com/en-us/dotnet/core/introduction#free-and-open-source (besucht am 10. Apr. 2024).

[30] Adegeo, *Was ist Windows Presentation Foundation? - WPF .NET*, Okt. 2023. Adresse: https://learn.microsoft.com/de-de/dotnet/desktop/wpf/overview/?view=netdesktop-8.0 (besucht am 10. Apr. 2024).

[31] *NuGET Gallery | Home*. Adresse: https://www.nuget.org/ (besucht am 10. Apr. 2024).

[32] Kouraklis, J., „MVVM as Design Pattern," in Okt. 2016, ISBN: 978-1-4842-2213-3. DOI: 10.1007/978-1-4842-2214-0_1.

[33] Gaudioso, V., „MVVM: Model-View-ViewModel," in *Foundation Expression Blend 4 with Silverlight*, T. Brown, B. Renow-Clarke, C. Collins et al., Hrsg. Berkeley, CA: Apress, 2010, S. 341–367, ISBN: 978-1-4302-2974-2. DOI: 10.1007/978-1-4302-2974-2_15. Adresse: https://doi.org/10.1007/978-1-4302-2974-2_15.

[34] Sergio, *Einführung in das MVVM-Toolkit - Community Toolkits for .NET*, März 2024. Adresse: https://learn.microsoft.com/de-de/dotnet/communitytoolkit/mvvm/ (besucht am 10. Apr. 2024).

[35] *CommunityToolkit.MVVM 8.2.2*. Adresse: https://www.nuget.org/packages/CommunityToolkit.Mvvm (besucht am 10. Apr. 2024).

[36] *NewtonSoft.Json 13.0.3*. Adresse: https://www.nuget.org/packages/Newtonsoft.Json/ (besucht am 12. Apr. 2024).

Abkürzungsverzeichnis

JSON *JavaScript Object Notation*
MVVM *Model-View-ViewModel*
UI *User Interface*
UML *Unified Modeling Language*
WPF *Windows Presentation Foundation*
XAML *Extensible Application Markup Language*

Erklärung zur selbstständigen Abfassung der Bachelorarbeit

Name: .....

Geburtsdatum: .....

Matrikelnummer: .....

Fach, in welchem die Arbeit angefertigt wird: .....

Titel der Bachelorarbeit: .....

.....

Ich versichere, dass ich die eingereichte Bachelorarbeit / die entsprechend gekennzeichneten Teile der Bachelorarbeit selbstständig und ohne unerlaubte Hilfe verfasst habe. Anderer als der von mir angegebenen Hilfsmittel und Schriften habe ich mich nicht bedient. Alle wörtlich oder sinngemäß den Schriften anderer Autoren entnommenen Stellen habe ich kenntlich gemacht.

.....

Ort, Datum Unterschrift

[1]Bei einer Gruppenarbeit gilt o. für den entsprechend gekennzeichneten Anteil der Arbeit.