JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

Uncertainties in Requirements to Code Trace analysis

BACHELORARBEIT
(Projektpraktikum)

zur Erlangung des akademischen Grades

Bakkalaurea der technischen Wissenschaften

im Bachelorstudium

INFORMATIK

Eingereicht von:
Carina Punz, 0555085

Angefertigt am:
Institut für Systems Engineering und Automation

Betreuung:
Univ.-Prof. Dr. Alexander Egyed, M.Sc.

Linz, Oktober 2009

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Bakkalaureatsarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, Oktober 2009 Carina Punz

Kurzfassung

Die vorliegende Bakkalaureatsarbeit behandelt die Analyse und Auswertung eines Experimentes, welches am Institut für *System Engineering und Automation* durchgeführt wurde.
Den Basisteil dieser Arbeit bildete die Unterteilung der vorhandenen Klassen des Experimentes in Kategorien. Darauf aufbauen konnte dann eine genaue Bearbeitung und intensive Analyse der Ergebnisse erfolgen, um eine vielseitige Interpretation des Experimentes zu ermöglichen.
Einleitend wird der Begriff *Traceability* definiert, danach wird die experimentspezifische Software detailliert beschrieben. Im Anschluss daran erfolgt eine nähere Erläuterung des Experimentes, sowie meines persönlichen Einflusses darauf. Dabei werden alle notwendigen Begriffe und Unterteilungen erklärt. Abschließend wird eine umfangreiche Auswertung der Ergebnisse präsentiert, welche möglichst anschaulich und aufschlussreich gestaltet wurde.

Abstract

This bachelor thesis deals with the analysis and evaluation of an experiment, which was performed at the *Institute for Systems Engineering and Automation*.
The base part of this work was the subdivision of existing classes of the experiments into categories. This should provide a basis for my review and intensive analysis of my results in order to permit many possible interpretations.
Initiatively the term "traceability" is defined and then the experiment specific software is described in detail. Furthermore the experiment will be explained circumstantial, as well as my influence with the above-mentioned categorization. All necessary concepts and subdivisions will be explained. Finally, a substantial evaluation is made of my results, which gives descriptive information and an interesting insight into the experiment.

Inhaltsverzeichnis

1. Einleitung und Zielsetzung

Die Bakkalaureatsarbeit entstand im Rahmen eines Experimentes, welches am Institut für *Systems Engineering und Automation* durchgeführt wurde.
Das Institut realisierte mit Studenten der Johannes-Kepler-Universität und der technischen Universität Wien ein Experiment, das die Nachvollziehbarkeit des Menschen im Traceability-Prozess analysierte.

1.1. Aufgabenstellung

Aufbauend auf die bereits gewonnenen Daten, wurde von mir eine Unterteilung in verschiedenen Kategorien vorgenommen, um die Korrektheit der Nachvollziehbarkeit auf einen bestimmten Teilbereich festlegen zu können. Diese Gruppierung erfolgte manuell nach gewissen Kriterien.
Die Motivation für die Vorgehensweise entstand durch das Paper von [Perr92], wo eine ähnliche Splittung im Paper *Foundations for the Study of Software Architecture* stattfand. Dadurch erhoffte ich mir aufschlussreichere Ergebnisse, als jene, die bereits vorlagen.

1.2. Struktur

Diese Bakkalaureatsarbeit ist wie folgt aufgebaut:

- In Kapitel 2 wird eine kurze Einführung in die *Traceability*, sowie ihre Arten, Voraussetzungen und Einflüsse gegeben.
- Kapitel 3 stellt die Software, die im Rahmen dieser Arbeit Anwendung gefunden hat, vor.
- In Kapitel 4 wird das grundlegende Experiment, auf das meine Arbeit aufbaut, beschrieben, zudem erfolgen notwendige begriffliche Erklärungen.
- Kapitel 5 befasst sich mit meinem Einfluss auf das Experiment und präsentiert anschließend meine Ergebnisse.

2. Traceability

2.1. Begriff

Es gibt mehrere Arten *Traceability* zu beschreiben. Die meiner Meinung nach beste Erklärung liefern Orlena Gotel und Anthony Finkelstein, die folgende allgemein gültige Definition aufgestellt haben: *„Requirements traceability refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction."*
Das bedeutet, dass sowohl vor der Implementierung, als auch danach, die Nachvollziehbarkeit von Softwareanforderungen, gewährleistet ist. Es muss eine Verknüpfung zwischen der Softwaredarstellung und den Anforderungen geben, sogenannte *Traces*. [Gote94] Eine UML Komponente oder auch ein Source Code-Ausschnitt können eine Veranschaulichung davon sein.

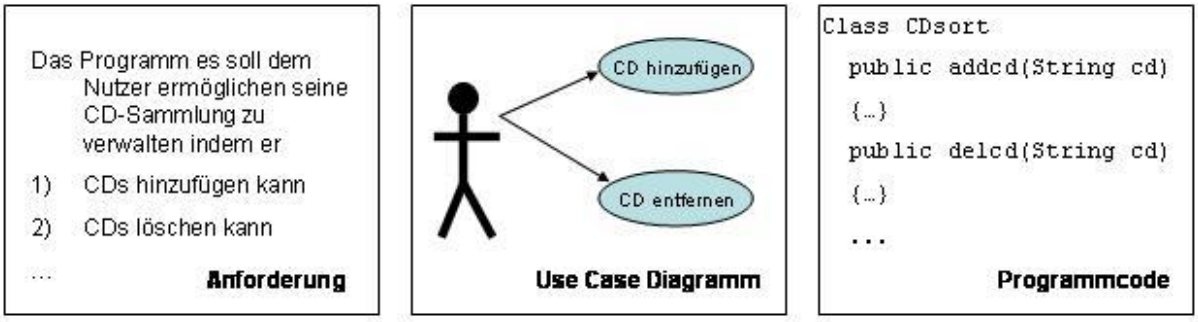

Abbildung 1: Darstellungen der Software

In Abbildung 1 besteht zum Beispiel eine Verbindung zwischen der Anforderung *CD hinzufügen*, dem Use Case mit demselben Namen und der Methode *public addcd(String cd)*. Zwischen der Anforderung und dem Use Case *CD löschen* mit dem Programmcode *public delcd(String cd)* ist ebenfalls eine Verknüpfung ersichtlich. Zum Auffinden solcher *Traces* wurde das sogenannte *Tracing* angewandt. Dies stellt ein Verfahren dar, das vorhandene *Traces* nicht nur ausfindig macht, sondern auch festhält, um sie zurückverfolgen zu können. [Minz04]

2.2. Arten und Darstellungen

2.2.1. Semantik der Traces

Es gibt verschiedenste Arten von Verknüpfungen, wobei jede auf ihre eigene Weise essentiell ist.

Anforderungen in Verbindung mit ihrem Ursprung:

- Anforderungen können mit ihrem Quell-Dokument verbunden sein.
- Ebenfalls kann eine Relation von einer Anforderung zu einer Person entstehen.
- Eine Anforderung kann zu ihrer Begründung verknüpft sein, welche Aufschlüsse darüber gibt, warum es diese Anforderung gibt.
- Ähnlich wie zu der Begründung, kann eine Anforderung zu einer Erklärung verlinkt sein, in der die Anforderung genauer erläutert wird.
- Ein Design-Element kann Bezug zu seinem Designer haben, der es modifiziert hat.

Verknüpfungen während der laufenden Entwicklung:

- *Tracing* zwischen Systemanforderungen und Subsystemen.
- Systemanforderungen verbunden mit den Anforderungen der Subsysteme.
- Verknüpfung von Tests mit den Anforderungen.[Wier95]

Man kann das *Tracing* sicher noch weiter spezfizieren, indem man beispielsweise die Arten der Veränderungen mit der Phase des Projektes in Verbindung setzt. Inwieweit das noch übersichtlich ist, sei dahingestellt.

2.2.2. Darstellung der Beziehungen

Tracing muss nicht nur vorhanden sein, sondern auch dargestellt werden können.

- *Traceability Matrizen*: Eine Matrix mit einer horizontalen und einer vertikalen Dimension listet die Datensätze auf, die miteinander in Beziehung gesetzt werden.

- *Graphische Modelle*

- *Cross Referenzen*: *Traces* zwischen den Datensätzen sind als Hyperlinks in einem Text eingebettet (informale Sprache oder formale Spezifikation). [Knet02]

2.3. Gründe für Traceability

Je nach Stakeholder ergeben sich viele Vorteile durch die Anwendung von *Traceability*.

- *Vorteile im Projekt Management*
  - Wenn Anforderungen mit den implementierten Komponenten verknüpft sind, kann die Wirkung der Anforderungsveränderung eingeschätzt werden.
  - Konflikte zwischen den Anforderungen können früher erkannt werden.
  - Nicht-erfüllte Anforderungen können gesammelt und noch erfüllt werden, um Produktverzögerungen zu verhindern.
  - Zukünftige Systeme profitieren davon, falls eine Code-Wiederverwendung stattfindet.
- *Vorteile für den Kunden*
  - Mit der *Tracing* Information kann die Qualität des Produktes in Bezug auf die Anforderungen evaluiert werden.
  - Akzeptanz-Tests können direkt zu den Anforderungen, für die sie gedacht sind, referenzieren.
  - Da Benutzeranforderungen mit Designanforderungen verknüpft sind, können sich die Entwickler auf die essentiellen Benutzeranforderungen konzentrieren.
- *Designer*
  - Für den Designer ist es leichter ersichtlich, ob das Design die Anforderungen erfüllt. [Wier95]

2.4. Probleme

Ohne die nötigen Hilfsmittel kann ein vollständiges *Tracing* sehr schwierig und zeitaufwändig werden. In weiterer Folge würden sich die Kosten des Verfahrens ebenso erhöhen.

Um diese Probleme zu vermeiden, wäre es hilfreich, geeignete Tools zu verwenden, oder den Detailgrad der Daten dynamisch anzupassen. Man sollte jedoch beachten, dass die

Art oder die Menge der Daten, sowie die Semantik der Darstellung, nicht immer standardisiert ist. Die Anzahl der Daten, die gesammelt werden, stellt sich auch als Problem heraus: Sammelt man zu wenig Daten, ist die Leistungsfähigkeit des *Tracings* eingeschränkt und die Daten wären somit nicht sehr aussagekräftig. Beim umgekehrten Fall, bei zu vielen Daten, wäre die Menge nicht mehr überschaubar. Hauptprobleme bereiten nun:

- *Initiale Datenerhebung:* die Art und Semantik der Daten
- *Inkrementelle Datenerhebung während des Evolutionsprozesses:*
  - *Datenintegrität:* Nicht vorhandene oder unvollständige Dokumentation bei Änderungen des Softwareproduktes
  - *Datenkonsistenz:* Daten von verschiedenen Quellen widersprechen sich
  - *Redundanzfreiheit:* z.B. multiple Erstellung einer Komponente
  - *Komplexität der Daten:* Menge der Daten mit tiefer Verschachtelung
- *Uneinheitlichkeit der Sprache und Form der Dokumentation:* Formalisierung, kontrolliertes Vokabular, text-mining Ansätze
- *fehlende Akzeptanz :* vorhandene Tools decken bereits Teilbereiche ab
  - configuration version management (CVM), z.B. perforce
  - integrated development environment (IDE), z.B. eclipse [Boga05]
- Diverse Management-Probleme aufgrund des immensen Aufwandes
- Manche Anforderungen können keiner Designkomponente wirklich zugeordnet werden (aufgrund der Programmiersprache oder nicht-funktionaler Anforderungen) [Wier95]

Durch Datenpflege und Wartung können zwar nicht alle der genannten Probleme gelöst werden, jedoch ist es eine wichtige Aufgabe im Verfolgbarkeitsprozess, der dazu beiträgt, das *Tracing* konsistenter zu gestalten. [Brci07]

2.5. Pre-Traceability und Post-Traceability

Traceability ist bidirektional und lässt sich somit noch weiter spezifizieren, indem man den Begriff auf folgende Bezeichnungen aufteilt: *Pre-Traceability* (*Pre-T*) und *Post-Traceability* (*Post-T*).
[Gote94] definieren diese Begriffe wie folgt:
„Pre-requirements specification (pre-RS) traceability is concerned with those aspects of a requirement's life prior to its inclusion in the RS (requirement production).“

„Post-requirements specification (post-RS) traceability is concerned with those aspects of a requirement's life that result from its inclusion in the RS (requirement deployment).”

Pre-T beschäftigt sich somit mit den Aspekten einer Anforderung, bevor diese in der Anforderungsspezifikation festgehalten wird. Was mit den Anforderungen in der Spezifikation bis zur endgültigen Software passiert, wird von *Post-T* abgedeckt.
Beide Arten von *Traceability* sind sehr wichtig. Gerade durch den Mangel an ihrer Unterscheidung entstehen die häufigsten Unstimmigkeiten. Abbildung 2 zeigt in welchen Bereichen *Pre-T* und *Post-T* interagieren und wie der Wissenfluss verteilt ist.

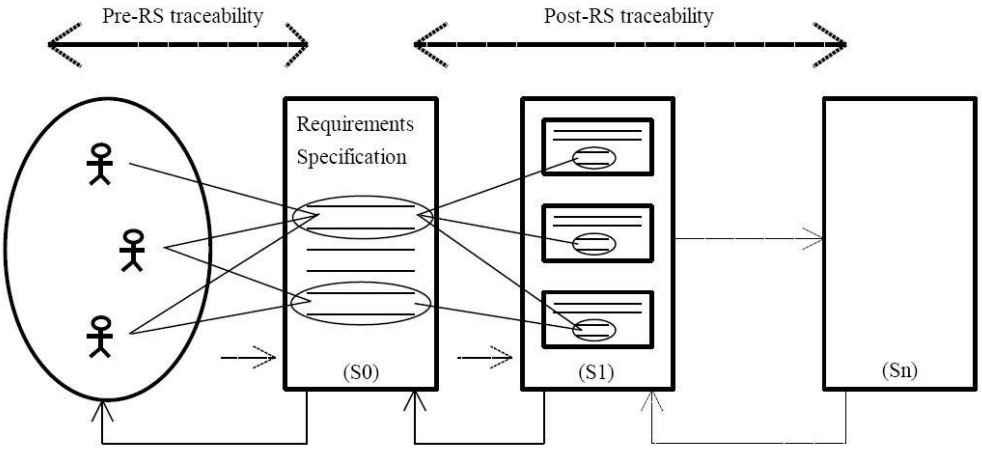

Abbildung 2: Die 2 Arten von Traceability

Die Hauptdifferenzen, mit denen diese Arten zu kämpfen haben, sind die Informationen und Probleme, die sie verarbeiten und unterstützen müssen. *Pre-T* umfasst das iterative Erarbeiten der Anforderungsspezifikation, vor allem wie und in welchen Schritten

Anforderungen erarbeitet wurden. [Pohl96] Verschiedenste Erhebungstechniken, Entscheidungen oder Fragestellungen können festgehalten werden.
Eine Anforderung setzt sich aus verschiedenstenen (oft widersprüchlichen) Quellen zusammen. Daher ist das Wichtigste, die Fähigkeit, Anforderungen von und zurück zu ihrer Ursprungsaussage, in der Anforderungsspezifikation festzulegen.
Nach [Gote94] sollte besonders diese Phase des *Traceability* stärker beachtet und verbessert werden als *Post-T*. Dies könnte hauptsächlich daran liegen, dass Probleme, die in der Anfangsphase entstehen, sich bis zum Schluss durchziehen und somit später schwieriger zu lösen sind.
Post-T hängt von der Fähgikeit ab, Anforderungen von und zurück zu einer Anforderungsspezifikation zurückzuverfolgen, bis hin zu einer Reihe von Produkten,
in denen sie sich verteilt haben. Wenn Änderungen eintreten, müssen diese durch die Kette ihrer Verteilung neu propagiert werden. [Gote94]
In dieser Phase hängen Entwurfsentscheidungen von Anforderungen ab. Da sich Anforderungen stets ändern können, sollte man diesen wichtigen Punkt hinsichtlich des Systems und der Tests gut abschätzen. [Brci07]

2.6. Voraussetzungen

Um eine gewisse Qualität garantieren zu können, sollte man einige Voraussetzungen beachten.
Für *Pre-T* ist es wichtig, die Wünsche des Kunden zu kennen und diese mit ihm zu besprechen. Durch Missverständnisse könnten die Anforderungen falsch verstanden bzw. interpretiert werden.
Die Anforderungsspezifikation, ein vollständiges Softwaremodell, das alle Funktionen abdeckt, und der dazugehörige Programmcode, sind unverzichtbar für die *Post-T*.
Folgende Punkte sind ebenfalls zu beachten:

- Festhalten der Beiträge der Mitarbeiter
- Festhalten von Alternativen und Begründungen
- Bestimmen einer festen Struktur, wie *Traces* festgelegt werden sollen

Besonders der letzte Punkt sollte hervorgehoben werden, da man ohne feste Strukturen leicht den Zusammenhang der Informationen übersieht und dies zu Unklarheiten führen

kann. Die bereits gesammelte Information müsste so neu überarbeitet werden und die bisherige Arbeit wäre somit wertlos. [Minz04]

2.7. Einfluss auf Software-Qualitätsmerkmale

Der Begriff der Qualität ist nach DIN55350-11 definiert, als die Gesamtheit der Eigenschaften und Merkmale eines Produktes oder einer Tätigkeit, die sich auf die Eignung zur Erfüllung gegenüber Erfordernissen bezieht.
Software-Qualität ist somit kein exakter Begriff und wird von Qualitätsmerkmalen beschrieben. Teilweise ist die Beurteilung subjektiv und hängt somit von der Perspektive des Betrachters ab.
Inwieweit sich *Traceability* auf Qualitätsmerkmale auswirkt, wird nach der Auflistung von [Tech01] analysiert. Die wichtigsten Punkte lauten:

- *Korrektheit:* Mithilfe von *Traceability* lässt sich gewährleisten, dass die genau bestimmten Anforderungen umgesetzt wurden.
- *Robustheit:* Anhand von Modellen und ihrer Funktionalität können diese Bedingungen durchgespielt werden und somit die Fehlerrate reduziert werden.
- *Erweiterbarkeit:* Hier lässt sich mit einem mit dem Programmcode verzahnten Modell ebenfalls Abhilfe verschaffen und Fehler werden ebenso vermieden.
- *Wiederverwendbarkeit: Traceability* wirkt sich hier nicht wirklich positiv aus. Falls eine Wiederverwendung gewünscht wird, müssen Modelle neu entworfen werden, damit die bereits vorher erwähnten Vorteile wieder gelten.
- *Portabilität:* Hier ergeben sich nur indirekt Vorteile. Modelle müssen wieder angepasst entwickelt werden, damit Fehler leichter gefunden und vermieden werden.
- *Benutzerfreundlichkeit: Traceability* stellt klar fest, welche Anforderungen umgesetzt wurden. Verbesserungen können nur erreicht werden, wenn die Anforderungen nach Kriterien der Benutzerfreundlichkeit entwickelt werden. [Minz04][Tech01]

2.8. Umsetzung von Traceability

Wie bereits erwähnt, erfolgt die Umsetzung von *Traceability* über *Traces*. Diese müssen während der Entwicklungsphase vom Entwickler selbst erstellt werden. Dies kann manuell oder automatisch geschehen.

2.8.1. Manuelles Tracing

Manuelles *Tracing* wird mithilfe von Tabellenkalkulationsprogrammen realisiert, indem jede Anforderung und deren Abhängigkeiten zum Code eingetragen wird. Leider ist diese Arbeitsweise sehr zeitaufwändig und fehleranfällig. [Mäde07] Eine Erleichterung und weitere Möglichkeit zur Realisierung wäre das Einsetzen von speziellen Tools, die das Zuweisen per Drag'n'Drop, Listen oder etwaige Lösungen anbieten. Ein solches Werkzeug wäre z.B. das *Trace Dependency Capture Tool*, welches in Kapitel 3.2 näher erläutert wird.

2.8.2. Automatisches Tracing

Um hier korrekte *Traces* zu finden, muss der Entwickler nichts per Hand eingeben oder etwaige Entwicklungsdokumente durchgehen. Es wird ihm eine Liste mit Artefakten (Teile der Entwicklungsdokumente) vorgelegt, welche mit der gerade betrachteten Anforderung zusammenhängen. Es werden jene Artefakte angezeigt, die am wahrscheinlichsten durch *Traces* miteinander verbunden sind. In Abbildung 3 ist die Erzeugung anschaulich dargestellt:
Alle Entwicklungsdokumente und Anfragen, hauptsächlich Anforderungen, werden einem Algorithmus als Input gegeben. Danach erfolgt eine Zerlegung der Entwicklungsdokumente (die hier nicht näher erläutert wird). Die Ausgabe ist die bereits erwähnte Liste mit den Artefakten. Nun kann der Entwickler die Liste systematisch durchgehen und alle Artefakte kennzeichnen, die einen tatsächlichen *Trace* bilden. [Frank08]

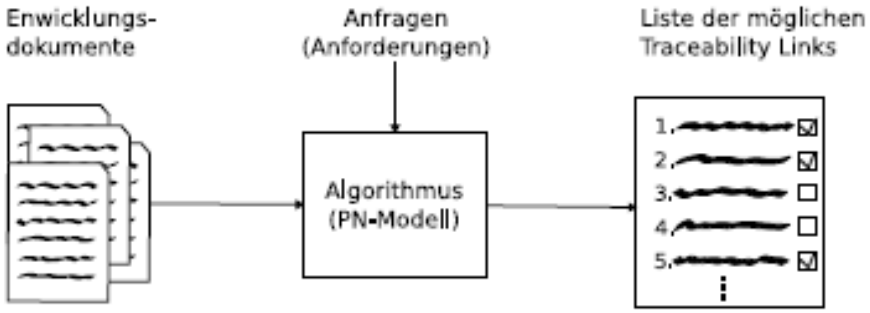

Abbildung 3: Vorgehen bei der automatisierten Verfolgbarkeit

In Tabelle 1 werden die beiden Techniken gegenüberstellt. Dabei ist ersichtlich, dass die automatisierte Methode weniger Erstellungs- und Wartungskosten benötigt. [Tryg97]

Kriterium	Manuell eingefügt	Automatisiert erzeugt
Erstellungskosten	Hoch	Niedrig
Wartungskosten	Hoch	Niedrig
Korrektheit	Zu viele Verknüpfungen	Fehlende Verknüpfungen
Speicherplatz	Zusätzlicher Speicherplatz	Zusätzlicher oder kein Speicherplatz
Flexibilität	Hoch	Hoch

Tabelle 1: Vergleich der Ansätze zur Traceability-Link-Erzeugung

[Frank08] untersuchte die automatisierte Nachverfolgbarkeit mithilfe der Techniken des Information Retrieval. Man muss zusätzlich beachten, dass die Dokumente korrekt und genau in das jeweilige Werkzeug eingegeben werden müssen, da sonst die Ausgabe dementsprechend beeinflusst bzw. verfälscht wird. Somit können automatisierte *Traceability*-Techniken nur in Projekten eingesetzt werden, die ein hohes Mindestmaß an strukturierten und vollständigen Entwicklungsdokumenten vorweisen.

3. Verwendete Software

3.1. GanttProject

Der Quellcode, der für die Testpersonen zur Verfügung gestellt wurde, stammt vom GanttProject, einem Open Source Projektplanungsprogramm. Mithilfe eines Gantt-Diagramms können Projektabläufe realisiert werden. [1]
Weitere Hauptmerkmale sind das Ressourcenmanagement, die Berichterstellung, wie auch Import/Export von MS Project, HTML und PDF Dateien.
Das Programm wurde in Java geschrieben, womit ein Java Runtime Environment von Sun ab Version 1.5 nötig ist, um das Programm ausführen zu können. Eine lokale Installation ist nicht notwendig, da man GanttProject ebenso via Java Web Start online starten kann. [2]

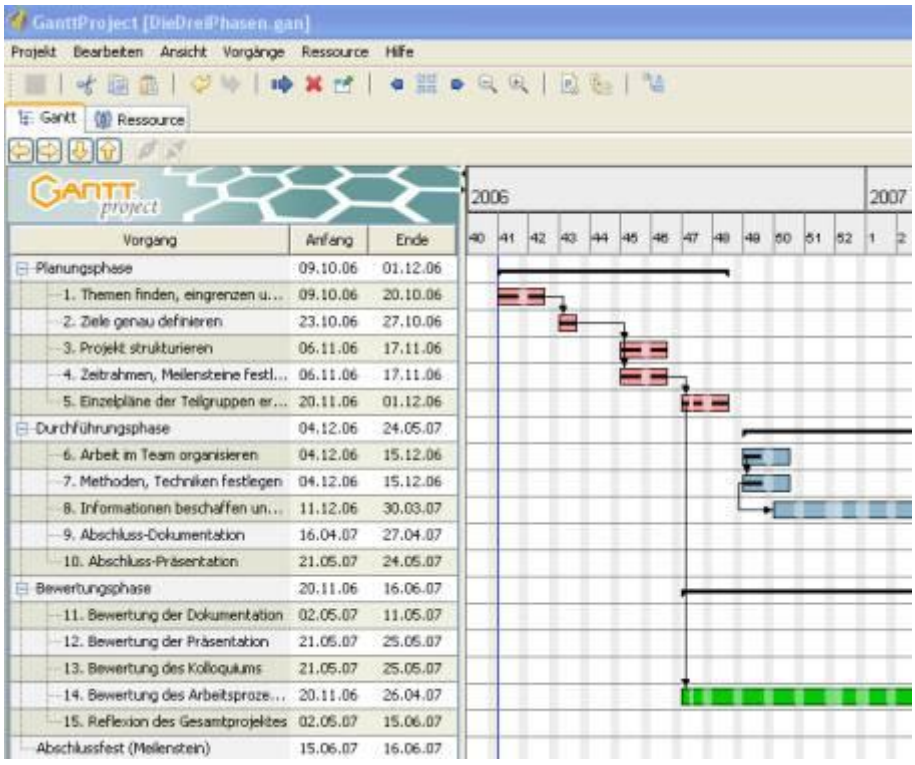

Abbildung 4: Screenshot des GanttProjects

Die Software läuft auf allen gängigen Betriebssystemen (Windows, Mac OS X, Linux) und wer mit Microsoft Project vertraut ist, kann das Programm ohne Blick in die Dokumentation leicht bedienen.

3.2. Trace Dependency Capture Tool

Um den Quellcode betrachten zu können und ihn anhand dessen Anforderungen zuzuweisen, wurde das *Trace Dependency Capture Tool* verwendet. Dies ist ein Tool für das manuelle *Tracing*.

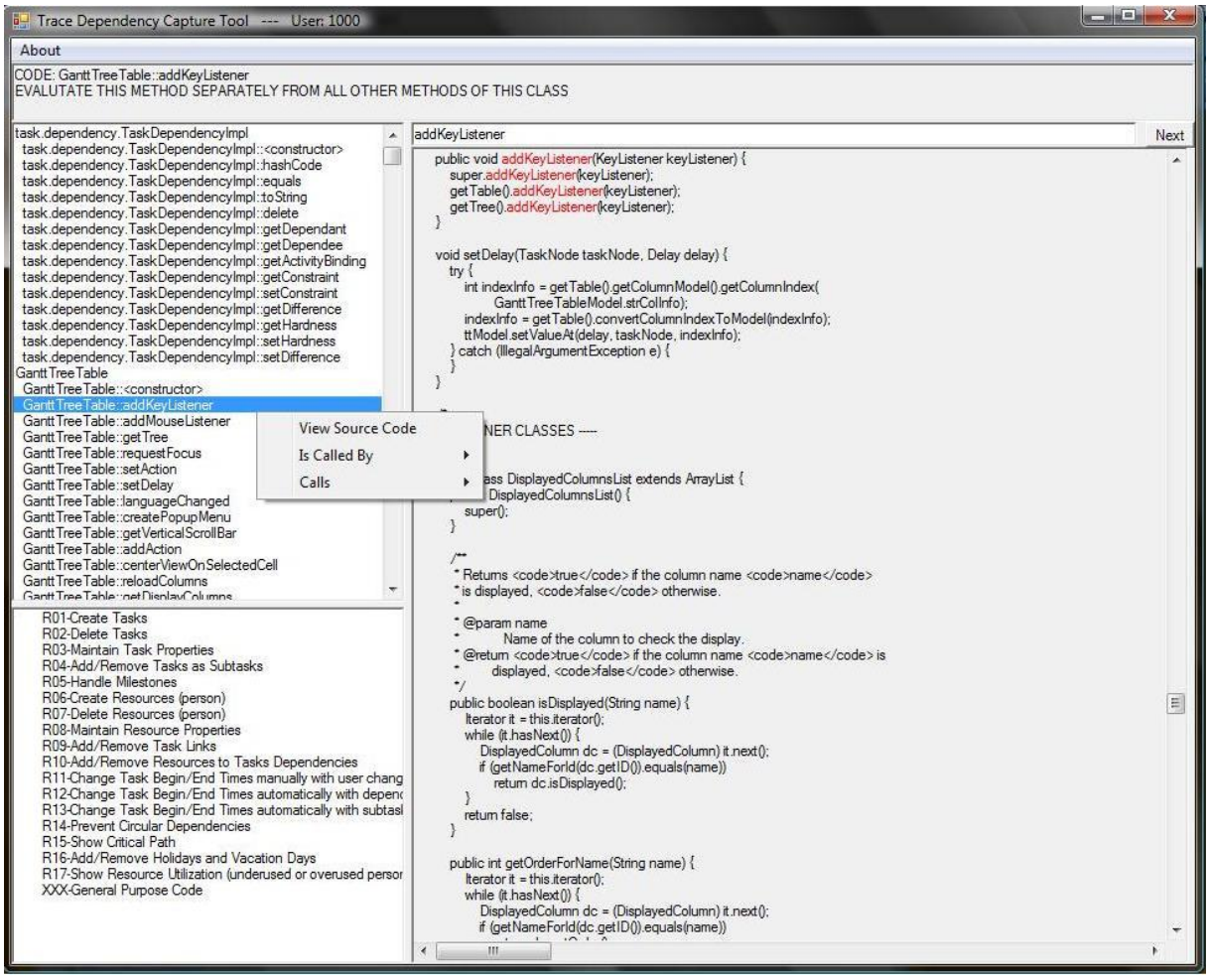

Abbildung 5: Trace Dependency Capture Tool

Wie in der Abbildung ersichtlich, wird die Software in drei Teilbereiche gegliedert: Klassen- bzw. Methodenauflistung, Anforderungen und Source Code.
Links oben wird eine Klasse mit ihren Methoden aufgelistet. Man kann je nach Klasse oder Methode verschiedenste Funktionen darauf anwenden:

- *View Source Code:* der Source Code der Klasse oder Methode wird im Fenster rechts angezeigt.

- *Parent*: Diese Funktion steht nur Klassen zur Verfügung. Hier ist ersichtlich, ob eine Klasse eine Oberklasse besitzt. Zu der Parentklasse kann der Code ebenfalls direkt im Fenster rechts angezeigt werden.

- *Child:* Selbe Funktion wie Parent, jedoch werden statt Oberklassen abgeleitete Klassen angezeigt.

- *Is Called By:* Diese Option steht Klassen und Methoden zur Auswahl. Hier werden Klassen bzw. Methoden dargestellt, die die selektierte Klasse/Methode aufrufen.

- *Calls:* Wie *Is Called By*, nur mit dem Unterschied, dass diesmal Klassen und Methoden angezeigt werden, die von der marktierten Methode aufgerufen werden.

Was jetzt noch zur Erklärung fehlt, sind die Anforderungen. Diese werden links unten angezeigt. Man hat verschiedene Möglichkeiten Anforderungen zu bewerten: *Trace*, *Potential Trace* und *No Trace*.
Kurz gesagt gibt man an, ob ein *Trace* auf die jeweilige Anforderung zutrifft oder nicht. Wenn man sich nicht sicher ist, ob ein *Trace* vorliegt, kann man immer noch *Potential Trace* angeben. Als Hilfe gibt es oberhalb der Source-Code-Anzeige noch zusätzlich eine Suchfunktion.

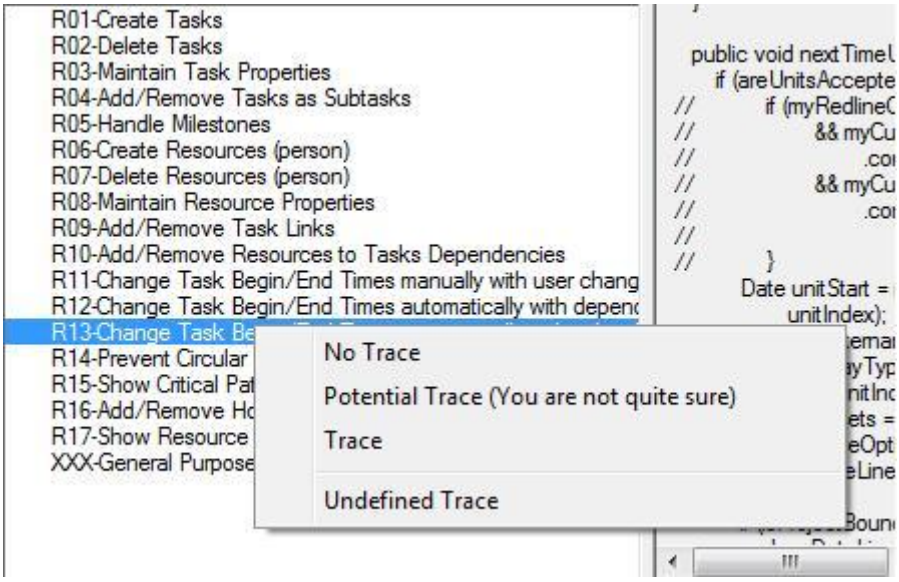

Abbildung 6: Tracezuweisung

4. Beschreibung des Experiments

4.1. Einleitung

Die Grundlage meiner Arbeit und Ergebnisse, beruhen auf dem Experiment von [Egye08]. Dieses wurde an der Johannes-Kepler-Universität Linz mit 39 und an der Technischen Universität Wien mit 88 Master-Level Studenten durchgeführt. Ihre Aufgabe bestand darin, vorgegebene Anforderungen einem bestimmten Source Code zuzuweisen, um *Traces* zu finden. Einige von ihnen untersuchten Zusammenhänge zwischen Anforderungen und Java Klassen (*Class Traces*), während die anderen, statt den Klassen, Java Methoden (*Method Traces*) prüften.

Als Source Code diente das Open Source Tool GanttProject aus Kapitel 3.1. Das System ist sehr groß und besteht aus 41 KLOC Java Code, der auf 516 Klassen und 3689 Methoden aufgeteilt ist. Aufgrund der Größe, Komplexität des Projektes und Unwissenheit der Testpersonen bezüglich des Codes, stellte sich das als nicht gerade simple Aufgabe heraus.

Da die Anzahl der Klassen und Methoden in einer angemessen Zeit von den Testpersonen unmöglich bewältigt werden könnte, wurden aus der großen Anzahl 85 Java Klassen und deren 604 Methoden ausgewählt, da diese am ehesten auf die 17 zugeschnittenen Anforderungen passten. Unter der Auswahl befanden sich verschiedenste Arten von Klassen, wie GUI Code oder auch nur General Purpose Code.

Die Testpersonen in Linz wurden in zwei Gruppen aufgesplittet. Wie bereits erwähnt, fokussierte sich die eine Gruppe auf die Klassen, während die ander nur ein Auge auf die Methoden warf. Für dieses Experiment wurde ihnen das *Trace Dependency Capture Tool* aus Kapitel 3.2 zur Verfügung gestellt und ihre Zeit auf 90 Minuten eingeschränkt. Bei der Zuweisung einer Anforderung ließ man jedoch nur *Trace* oder *No Trace* zu, da *Potential Trace* nicht wirklich aussagekräftig war. Falls man sich also nicht sicher war, sollte man das Feld einfach leer lassen.

Die Ergebnisse dieses Experimentes und mein Beitrag dazu werden in Kapitel 5 noch genauer erläutert.

4.2. Consensus vs. Conflict Data

Wie bereits erwähnt, konnten die Testpersonen mittels Tool zwei Wertungen abgeben: *Trace* und *No Trace*. Zusätzlich muss es einen Bezug zwischen den Daten, bzw. zwischen den Klassen und Methoden geben. Für die Auswertung gibt es eine Übereinstimmung (*consensus*) und einen Konflikt (*conflict*), bei denen sich die Testpersonen widersprachen.
[Egye08] legte fest, wenn mind. drei Testpersonen sich sicher waren, dass ein *Trace* vorliegt, dann liegt ein *consensus* vor. Auch wenn vielleicht mehr als diese drei Personen, sogar der Ansicht waren, dass es *No Trace* ist.

4.3. Class and Method Agreements

Wenn für eine Methode ein *Trace* gefunden wurde, so musste laut [Egye08] auch die dazugehörige Klasse einen *Trace*, in Bezug auf die jeweilige Anforderung, vorweisen. Ob hier *consensus* oder *conflict* vorliegt, kann man ganz einfach herausfinden. Wenn eine Klasse bezüglich einer bestimmten Anforderung als *Trace* definiert wurde und die Methoden der Klasse ebenso, so liegt ein *consensus* vor. Dasselbe gilt auch für den umgekehrten Fall, falls beide *No Trace* sind.
Ein *conflict* liegt lediglich vor, wenn bei der Klasse ein *Trace* bzw. *No Trace* festgelegt wurde und die Methoden das Gegenteil aufweisen.

Class	Methods	
Trace	Trace	Consensus
No Trace	No Trace	
Trace	No Trace	Conflict
No Trace	Trace	

Tabelle 2: Class/Method Agreements

Somit kann eindeutig bestimmt werden, dass ein *Trace* vorliegt, wenn Klassen und Methoden übereinstimmen. Lediglich bei den *conflicts* ist eine genauere Auswertung nötig, ob nun wirklich ein *Trace* existiert oder nicht.

5. Auswertung und Ergebnisse

5.1. Vorbereitung

Um einen Überblick zu bekommen, wie die Testpersonen gearbeitet haben, wurde mir die Aufgabe gestellt, ebenfalls einige Klassen und Methoden mit den Anforderungen zu vergleichen. Der einzige Unterschied war jedoch, dass ich meine Auswahl und die Begründung für diese Entscheidung mitdokumentieren sollte. Ebenso musste ich Informationen angeben, ob ich eine *isCalledBy, Calls …* Funktion angesehen habe und ob diese zu meiner Entscheidung beigetragen hat.
Anfangs waren die Zuweisungen etwas schwieriger bzw. ging alles etwas langsamer voran, da ich mit dem Code nicht sehr vertraut war. Die ausgewählten Klassen arbeitete ich auf Klassen- und Methodenebene durch.
Schlussendlich kann ich sagen, dass sich mit der Zeit gewisse Muster wiederholten und die Zuordnung schneller ging, bzw. dass sich teilweise die Abarbeitung der Klassen- und der Methodenebene nicht so sehr voneinander unterschied. Manche Methoden waren jedoch derart komplex, dass man sich teilweise die ganze Klasse ansehen musste, um zu verstehen, welche Funktionen ausgeführt werden.

5.2. Einteilung der Klassen in Kategorien

Um mehr Aufschluss über die Ergebnisse des Experimentes von [Egye08] geben zu können, bediente ich mich einer Einteilung, die schon viel früher Anwendung fand. [Perr92] definierte Software Architektur wie folgt:

Software Archtiecture = {Elements, Form, Rationale}

Elements stellen Daten dar, *Forms* die Eigenschaften der *Elemente* und *Rationale* dient als Basis für die Architektur, wie Randbedingungen, welche oft von Anforderungen abgeleitet sind.
Wichtig in Bezug auf meine Arbeit war die weitere Unterscheidung der *Elements*, die in *processing elements*, *data elements* und *connecting elements* aufgeteilt waren. *Processing*

elements sind jene Komponenten, die für die Transformation der *data elements* sorgen, während *data elements* logischerweise Daten beinhalten. *Connecting elements* halten die verschiedenen Teile der Architektur zusammen und können jeweils ein *processing* oder *data element* sein.

In Bezug auf das GanttProject war genau die selbe Einteilung nicht adäquat. *Data elements* wurden beibehalten, während *processing* und *connecting elements* zu einer gemeinsamen Kategorie verschmolzen sind. Da das System viele graphische Klassen und Elemente enthält, entstand als dritte Kategorie *GUI*. Für Klassen, die man in keiner dieser Teilbereiche zuordnen konnte, wurde noch zusätzlich eine vierte Kategorie angelegt. Die Verteilung sieht nun wie folgt aus:

- *Data*
- *Control*
- *GUI*
- *General Purpose Code*

Doch was macht eine Klasse zu einer *Data*-Klasse und ab wann grenzt sie an eine andere Klasse? Es gab gewisse Kriterien, die für mich ausschlaggebend waren, wann welche Klasse zu welcher Kategorie gehört.

5.2.1. Data

Unter *Data* verstehe ich alle Rohdaten, die zur Verfügung stehen. Ein Model würde ich dieser Kategorie zum Beispiel zuteilen, da dieses nur als eine Art „Basis“ dient. Beispiele dafür wären in dem GanttProject z.B. das *TreeTableModel* oder eine Resource (HumanResource, ProjectResource).

```
*/class HumanResource extends ProjectResource
...
HumanResource(HumanResourceManager manager)
System.out.println("new HumanResource");
this.name = "";
this.id = -1; // has to be assigned from the managr when re-
source is
```

```
customFields = new Hashtable(manager.getCustomFields());
myManager = manager;
// added
/** Creates a new instance of HumanResource */
HumanResource(String name, int id, HumanResourceManager manag-
er)
System.out.println("new HumanResource");
this.name = name;
this.id = id;
customFields = new Hashtable(manager.getCustomFields());
myManager = manager;
public void delete()
super.delete();
myManager.remove(this);

...
```

Tabelle 3: Auszug resource.HumanResource

Diese Klasse erstellt eine neue *HumanResource* und führt alle Operationen aus, die nötig sind, um eine neue Resource zu erstellen. Später werden diese Daten diversen Projekten im GanttProject zugeteilt.

5.2.2. Control

Eine Klasse wird dieser Kategorie zugeteilt, wenn unter anderem ersichtlich ist, dass etwas berechnet bzw. gesteuert wird. Sehr gut sichtbar ist dies bei der Klasse *action.CalculateCriticalPathAction*. Der Klassenname sagt schon eindeutig: Hier wird der kritische Pfad berechnet. Ein weiteres Beispiel, wäre *task.algorithm.FindPossibleDependeesAlgorithmImpl*.

```
abstract class FindPossibleDependeesAlgorithmImpl implements
FindPossibleDependeesAlgorithm

private TaskContainmentHierarchyFacade myContainmentFacade;
public FindPossibleDependeesAlgorithmImpl()
public Task[] run(Task dependant)
myContainmentFacade = createContainmentFacade();
ArrayList result = new ArrayList();
Task root = myContainmentFacade.getRootTask();
Task[] nestedTasks = myContainmentFacade.getNestedTasks(root);
processTask(nestedTasks, dependant, result);
return (Task[]) result.toArray(new Task[0]);
```

```
protected abstract TaskContainmentHierarchyFacade createCon-
tainmentFacade();
private void processTask(Task[] taskList, Task dependant, Ar-
rayList result)
for (int i = 0; i < taskList.length; i++)
     Task next = taskList[i];
if (!next.equals(dependant))
     Task[] nested = myContainmentFacade.getNestedTasks(next);
// if (nested.length==0)
result.add(next);
//
// else
     processTask(nested, dependant, result);
//
…
```

Tabelle 4: Auszug task.algorithm.FindPossibleDependeesAlgorithmImpl

Der Name sagt hier eigentlich schon aus, dass mögliche Abhängigkeiten gefunden werden sollen. Dazu müssen alle Tasks durchlaufen und berechnet werden, wer von wem abhängig ist.

5.2.3. GUI

Hier scheint es fast noch einfacher zu sein, Zugehörigkeiten zu finden. Klassen mit graphischen Elementen bzw. die diese beinhalten, finden sich in dieser Kategorie wieder. Dies könnte z.B. eine Klasse sein, welche von einem EventObject oder einem anderen graphischen Element erbt. Die folgende Klasse *GanttChartTabContentPanel* enthält sehr viele solche Merkmale. Der Klassenname selbst sagt auch schon aus, dass es ein Panel ist, also ein typisches Charakteristikum, was für die Graphikdarstellung gut ist.

```
GanttChartTabContentPanel

private JSplitPane mySplitPane;
private Component myTaskTree;
private Component myGanttChart;
private final TaskTreeUIFacade myTreeFacade;
private JPanel myTabContentPanel;
private final IGanttProject myProject;
private final UIFacade myWorkbenchFacade;
GanttChartTabContentPanel(IGanttProject project, UIFacade
```

```
workbenchFacade,
TaskTreeUIFacade treeFacade, Component ganttChart)
myProject = project;
myWorkbenchFacade = workbenchFacade;

myTreeFacade = treeFacade;
myTaskTree = treeFacade.getTreeComponent();
myGanttChart = ganttChart;

...

private Component createButtonPanel()
Box buttonBar = Box.createHorizontalBox();
//JToolBar buttonBar = new JToolBar();
//buttonBar.setFloatable(false);
TestGanttRolloverButton unindentButton = new TestGanttRollo-
verButton(
myTreeFacade.getUnindentAction())
public String getText()
return null;

...
```

Tabelle 5: GanttChartTabContentPanel

5.2.4. General Purpose Code

Alles was nicht programmspezifisch ist wird als *General Purpose Code* kategorisiert.

Ein sehr gutes Beispiel hierfür wäre der Gregorianische Kalender, realisiert als Klasse *time.gregorian.GregorianCalendar*.

```
*/class GregorianCalendar extends java.util.GregorianCalendar
/**
* Overrides the original, to solve the october duplicated day
bug
*/
public void add(int field, int value)
if (field == Calendar.DATE)
this.add(Calendar.HOUR, value * 24);
else
super.add(field, value);
public GregorianCalendar()
super();
public GregorianCalendar(int year, int month, int date)
super(year, month, date);
public GregorianCalendar(int year, int month, int date, int
hour, int minute)
```

```
super(year, month, date, hour, minute);
public GregorianCalendar(int year, int month, int date, int
hour,
int minute, int second)
super(year, month, date, hour, minute, second);
public GregorianCalendar(Locale aLocale)
super(aLocale);
public GregorianCalendar(TimeZone zone)
super(zone);
public GregorianCalendar(TimeZone zone, Locale aLocale)
super(zone, aLocale);
```

Tabelle 6: time.gregorian.GregorianCalendar

Der Kalender ist nicht GanttProject-spezifisch. Er kann in jedem anderen Programm verwendet werden.

5.3. Verteilung

Für die Kategorisierung ließ ich, um sicher zu gehen, noch zwei andere Studenten die selbe Unterteilung vornehmen. Es stellte sich heraus, dass wir bei ca. zwei Drittel eine komplette Übereinstimmung hatten, während die anderen Zuordnungen manuell selektiert und noch einmal überprüft wurden. Nur wenige Klassen wurden von jedem anders kategorisiert.

Da manche Grenzen zu anderen Klassen schon zu nahe waren, habe ich auch oft eine zweite Kategorie zugeteilt. Diese wurde jedoch nicht in der Datenauswertung miteinbezogen, da die Analyse der Daten zu komplex geworden wäre. Die Einteilung der Kategorien erfolgte auf die 85 Klassen, die den Testpersonen zur Verfügung gestellt worden sind.

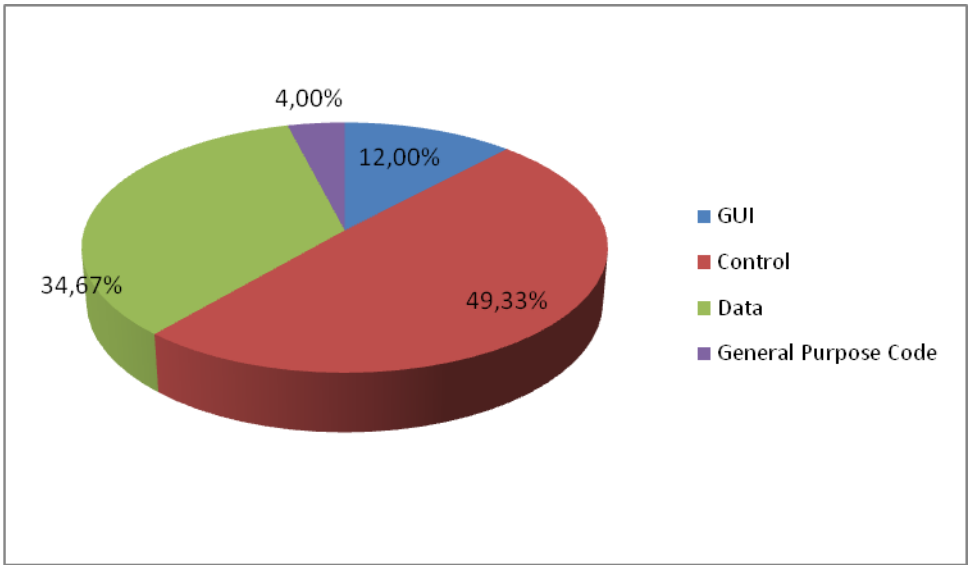

Abbildung 7: Aufteilung der Kategorien

Fast jedes zweite Element ist der Kategorie *Control* zugeordnet, gefolgt von *Data* und GUI. *General Purpose Code* kommt nicht sehr oft vor und ist auch bei den Ergebnissen nicht sehr aufschlussreich gewesen. Der Prozentanteil bei der Aufteilung ist sehr wichtig, da nicht allein die Anzahl der *Traces* oder *Conflicts* ausschlaggebend sind.

Bei der Zuweisung der Kategorien legte ich folgende Regel fest: Wenn Oberklassen gewissen Teilbereichen zugehörig sind, so muss auch die abgeleitete Klasse dieser Gruppierung zugeordnet sein . Diese kann natürlich weiteren Kategorien zugeordnet sein, z.B. aufgrund von Methoden, die hinzukommen. Jedoch bleibt die Hauptkategorie, wie in der Oberklasse, erhalten. Ein Beispiel hierfür wäre zum Beispiel Abbildung 8.

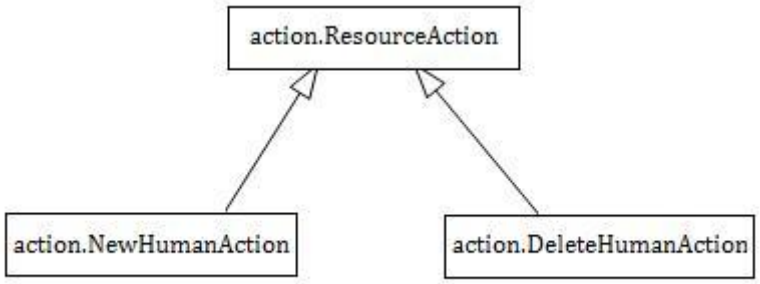

Abbildung 8: Erben der Kategorie

5.4. Conflicts

In [Egye08] wurden die *Conflicts* (in diesem Fall auf Klassenebene) bereits ausgewertet. Meine Aufgabe ist es, festzustellen, inwieweit die Kategorie eine Rolle dabei spielt. Beispielsweise musste ich analysieren ob komplexe *Control* Daten für die Testpersonen schwerer zuzuordnen waren als graphische Elemente der *GUI*-Kategorie usw.

Meine ersten Auswertungen bezogen sich auf die *Class and Method Agreements* aus Kapitel 4.3., welche wie folgt aussieht:

Class	Category	Conflicts	Traces/class	Methods/class
…				
ClazzName: resource.ResourceColumn	Data	**1**	2	1
ClazzName: resource.ResourceEvent	GUI	**2**	4	2
ClazzName: resource.ResourceNode	Data	0	3	3
…				
task.ResourceAssignmentCollectionImpl	Control	**3**	3	4
…				

Tabelle 7: Conflicts mit Kategorien

Wie in der Tabelle 7 ersichtlich, ist der Klassenname mit der zugehörigen Kategorie angegeben. Zusätzlich ist die Anzahl der *Conflicts*, sowie *Traces* je Klasse und Methode, gegeben. Die *Conflicts* ergeben sich aus der Differenz der *Traces* zwischen Klassen und Methoden.

Bei *resource.RecourceColumn* ist es offensichtlich: Ein *Conflict* liegt vor, weil die Klasse zwei *Traces* hat, aber die Methode nur einen. Komplizierter wird der Fall bei *task.ResourceAssignmentCollectionImpl*. Hier liegen drei *Conflicts* vor, obwohl die Klasse drei und die Methode vier *Traces* besitzt. Hier ist es schwierig zu sagen, welche Art von *Conflicts* vorliegen. Es wurde anscheinend ein *Trace* zuwenig in der Klasse gefunden und aus den anderen zwei *Conflicts* lässt sich schließen, dass andere Anforderungen ausgewählt worden sind.

Nach den ersten Auswertungen, bezüglich der *Conflicts* der jeweiligen Kategorie, entschied ich, die Gruppierung *General Purpose Code* aus meiner Wertung auszuschließen. Die Kategorie kommt zu selten vor und es stehen derart wenig Daten zur Verfügung, sodass die Ergebnisse nicht annähernd repräsentativ sind.

Wie bereits in Abbildung 7 ersichtlich war, kommt *Control* mit dem Anteil der Klassen am häufigsten vor, ebenso wie mit der Anzahl der *Conflicts*. Ähnlich tritt dieser Effekt bei *Data*-Kategorie auf, die mit ähnlichen Werten aufscheint. Einzig *GUI* hat ziemlich wenige *Conflicts*. Somit lässt sich sagen, dass *GUI*, grob gesehen, etwas leichter handzuhaben ist, rein von der Betrachtung der Anzahl und des Wertebereiches. Um sicher zu sein, dass *GUI* wirklich die leichtere Kategorie ist, werden zur Auswertung noch andere Maßzahlen herangezogen.

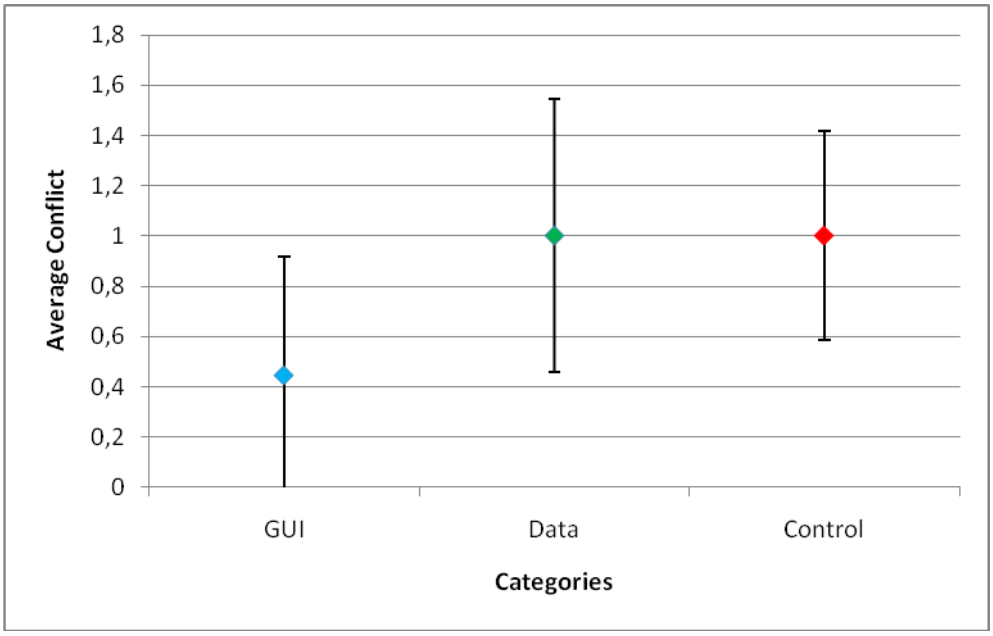

Abbildung 9: Class/Method Conflicts

5.5. Einflussfaktoren

Um die Auswertungen besser analysieren zu können, brauchen wir gewisse Messwerte, die uns erlauben, Aussagen zu treffen. Diese sind in Abbildung 10 ersichtlich:

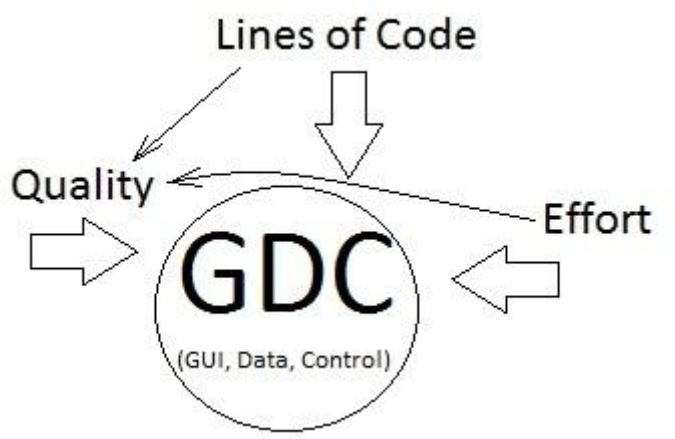

Abbildung 10: Einflüsse auf die Kategorisierung

Unter Qualität versteht man, dass möglichst wenig *Conflicts* auftreten. Je konfliktfreier die Klassen, desto höher die Qualität. Darauf Einfluss haben *Lines of Code*, sowie *Time Effort*. Jedoch ist auch die einzelne Auswertung der Programmzeilen wichtig für die Kategorien, ebenso wie *Time Effort* alleine.
Die Auswertungen in den folgenden Unterpunkten beziehen sich, bezüglich der *Trace*findung bzw. der Auftreten der *Conflicts*, alle auf Klassenebene.

5.5.1. Lines of Code

Um einen *Trace* zuordnen zu können, müssen die Testpersonen den Source Code lesen. Somit hat auch die Anzahl der Zeilen einen Einfluss auf die Entscheidung. In [Egye08] wird zunächst behauptet, dass die Code Komplexität den Aufwand erhöht, was wiederum zu schlechterer Qualität führt. Nach der Einteilung der Klassen anhand ihrem LOC im Vergleich zu ihrem Aufwand, stellte sich heraus, dass zwar der Aufwand wirklich steigt, aber die Qualität deswegen nicht schlechter wird.
In Abbildung 11 ist die Aufteilung der Größe ersichtlich. Die Verteilung ist ziemlich absteigend: *GUI* besitzt durchschnittlich eher Klassen mit mehr LOC, während die Anzahl bei *Data* und *Control* kleiner ist. Interessant ist, dass bei *GUI* ziemlich verschieden große Klassen vorhanden sind, während das Intervall bei den anderen Kategorien eher klein gehalten ist.

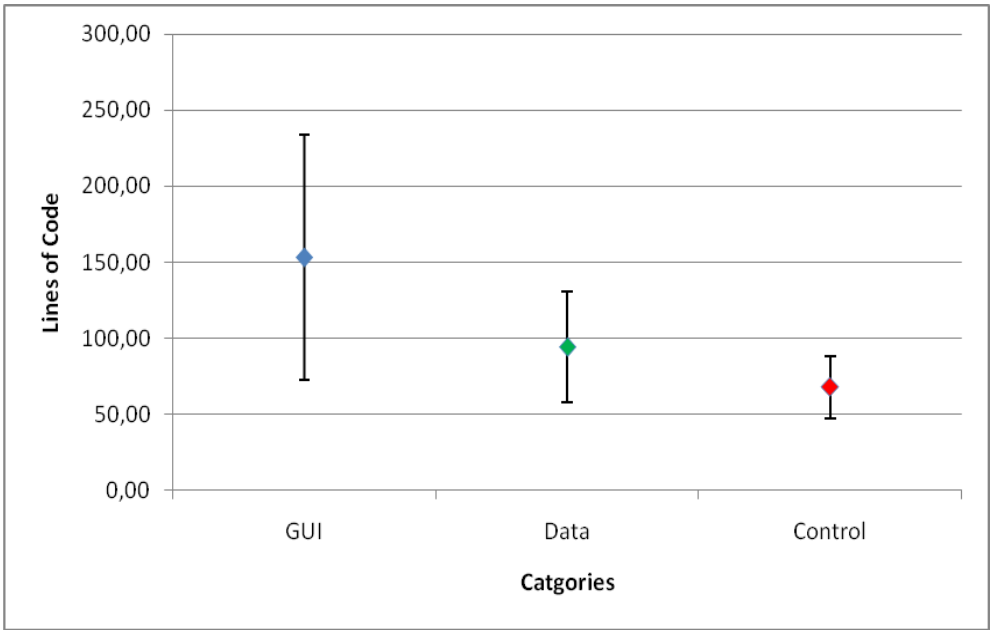

Abbildung 11: LOC der Klassen je Kategorie

Im Bezug zur Qualität wurden die Messwerte der LOC mit den Anzahl der aufgetretenen *Conflicts* verglichen. Bei *GUI* kommen *Conflicts* in teilweise so extrem kleinen bzw. großen Klassen vor, dass sich die Wertabweichung schon im negativen Bereich aufhalten würde. Bei *Data* und *Control* halten sich die Werte in Grenzen, jedoch ist interessant, dass bei *Data* eher mit wenig LOC *Conflicts* auftraten, während dies bei *Control* umgekehrt der Fall ist.

Der zweite wichtige Punkt wäre, bei welcher Kategorie, in Abhängigkeit von LOC, wurden die meisten *Traces* bzw. *No Traces* verzeichnet? Betrachtet man die Werte detaillierter, so stellt sich heraus, dass bei *GUI* die LOC egal sind. Durchschnittlich ist der Wert wieder höher angelagert als bei den anderen, die wieder absteigend folgen, jedoch ist das Intervall wieder immens. *Data* und *Control* haben die Gemeinsamkeit, dass mit mehr LOC eher ein *Trace* gefunden wird, als mit weniger. Vielleicht ist bei den größeren Klassen gleich klar, ob es sich um ein *Trace* handelt oder nicht und die Kleineren mit derart wenigen Informationen dienen, dass es schwierig wird, sagen zu können, ob ein *Trace* zutrifft.

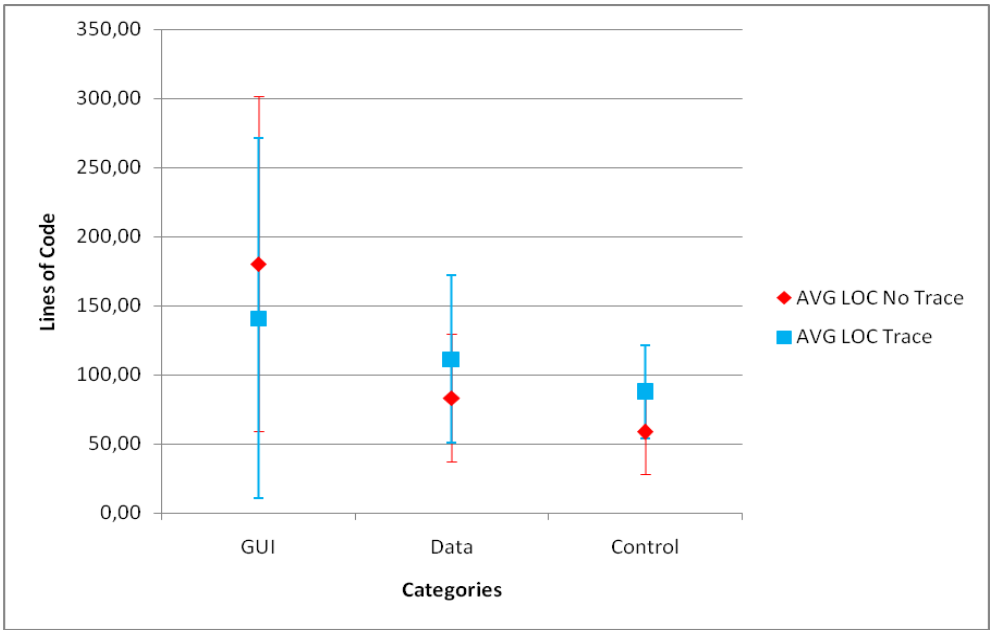

Abbildung 12: LOC in Bezug zur Traceverteilung

Um mehr Auskunft über die LOC zu erhalten, brauchen wir noch einen zweiten Messwert, welcher im nachfolgenden Kapitel genauer erläutert wird.

5.5.2. Time Effort

In [Egye08] wurde eine These aufgestellt, die besagt: „More effort implies better quality“. Es wurde untersucht, ob die schnelleren Testpersonen weniger *Conflicts* erzeugten, als jene, die sich mehr Zeit für ihre Entscheidung ließen. Die Testpersonen wurden in vier Schnelligkeitsgruppen mit jeweils 25% unterteilt. Es stellte sich heraus, dass gerade das schnellste Viertel deutlich weniger *Conflicts* vorweisen konnte, als die langsamen Gruppen. Jedoch wurde angenommen, dass dies nichts mit Intuitivität oder schlechten bzw. guten Studenten zu tun hat, sondern eher damit, dass leichtere *Traces* schneller und klarer zu finden waren. Daraus folgte, dass schwerere *Traces* mehr Zeit benötigten und die zusätzliche Zeit auch nicht half, die Qualität zu verbessern, aufgrund der Schwierigkeit.

Augrund der Daten, die vorliegen, ließ sich schon einmal leicht erkennen, dass die Testpersonen, die mit den Methoden beschäftigt waren, mehr Zeit brauchten, als jene,

die Klassen und Anforderungen analysierten. Jedoch war die Verteilung insgesamt sehr proportional verteilt und wies keine kategoriespezifischen Unterschiede auf.

Vergleicht man nun die Zeit, sieht man, dass sich bei *GUI* ebenso wie der LOC Wert, der *Time Effort* proportional hierzu erstreckt. Nicht nur viel bzw. wenig LOC, auch mit wenig und viel Zeit ist diese Kategorie konfliktanfällig. Es könnte auch sein, dass sich der wenige *Time Effort* auf die kleineren Klassen bezieht und die größeren Klassen mehr Zeit beanspruchen. Die Werte sind sonst, wie vorher schon ziemlich absteigend geordnet und nicht recht signifikant. Man kann jedoch sagen, dass die Zeit kaum eine Rolle spielt und ziemlich überall gleich ist. Somit kann ich „More effort implies better quality“ auch nur widerlegen.

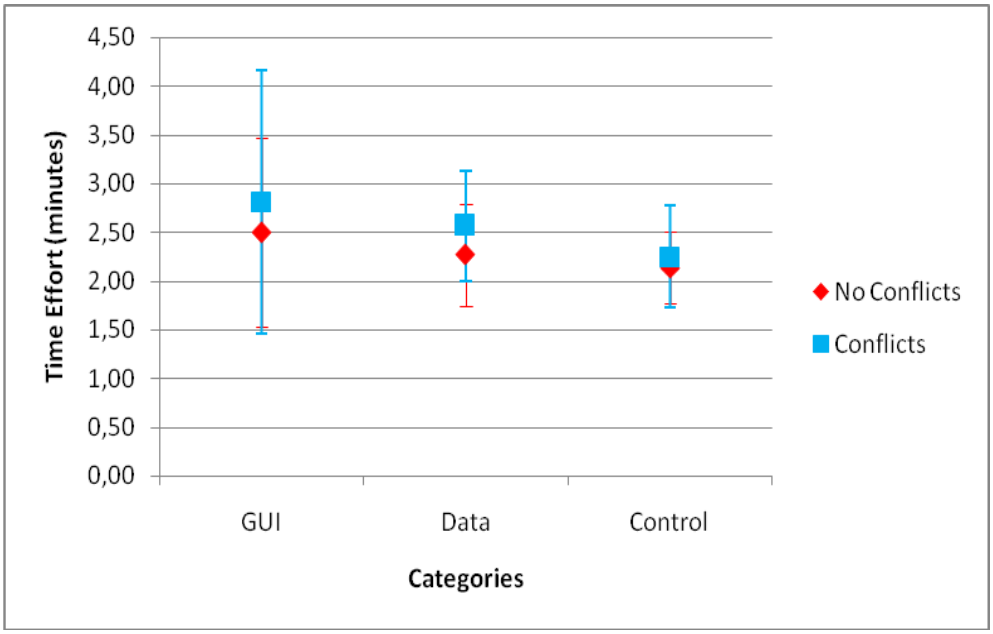

Abbildung 13: Quality unter Time Effort

In Bezug zur *Trace*findung stellt sich heraus, dass *GUI* hier auch durchschnittlich mehr Zeit benötigt als die anderen Kategorien. Bei *Control* sind die Werte fast identisch und somit ist es egal ob ein *Trace* gefunden wurde oder nicht. Bei *Data* war man schneller mit dem Finden von *Traces*, was auch logisch ist: Sobald ein *Trace* gefunden wurde, konnte man sich der nächsten Klasse widmen. Sagen zu können, dass kein *Trace* vorliegt dauert immer länger, da man die ganze Klasse durchgehen muss und somit automatisch mehr Zeit braucht bzw. auch Zeit zum Überlegen. Bei *GUI* ist dies genau der umgekehrte Fall. Man könnte sagen, die Klassen seien so klein, dass man schnell sagen kann, dass

kein *Trace* vorliegt. Doch auf diese Theorie kann man sich nicht verlassn, wenn man bedenkt, dass *GUI* im Durchschnitt sogar die meisten großen Klassen besitzt. Anscheinend ist es hier schwerer sagen zu können, dass ein *Trace* vorliegt und auszuschließen leichter fällt, im Gegensatz zu *Data*.

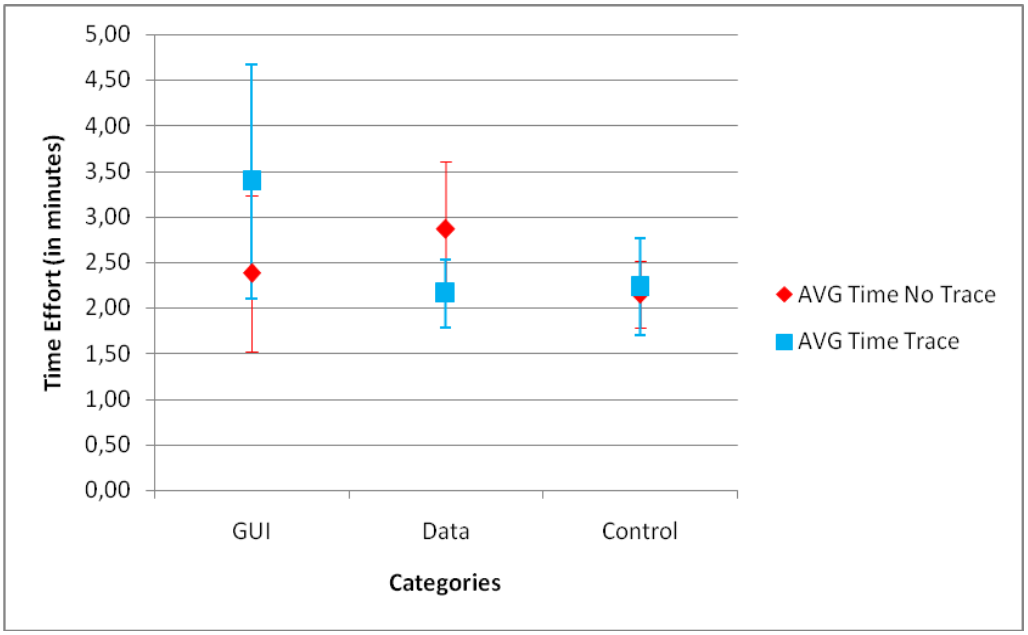

Abbildung 14: Trace/No Trace Verteilung unter Time Effort

5.5.3. Time Effort/LOC

Um noch aussagekräftigere Daten zu bekommen, stellte ich mir die Frage, in welchem Verhältnis die Zeit mit den LOC steht.

Bei der Berechnung ging ich wie bei normalen Software Metriken vor: Wenn man z.B. eine Fehlerdichte herausfinden will, muss man die Fehler durch die (K)LOC dividieren.[3] Dies lässt sich einfach mittels Gleichung darstellen und auf meine Berechnungen anpassen:

$$TD = Tkum/LOC \text{ bzw. } CD = Ckum/LOC$$

Dabei sei *TD* (*CD*): *Trace*- bzw. *Conflict*-Dichte pro Programmzeile

Tkum (*Ckum*): die kumulierte *Trace*- bzw. *Conflict*anzahl

TD(*CD*): statt der Fehlerdichte wird hier die Trace-Dichte bzw. Conflict-Dichte definiert.

Beginnen wir mi der *Conflict*-Dichte, also mit der Dichte der Qualität. Die Grafik bestätigt nur, was in LOC und *Time Effort* schon ersichtlich war: *GUI* ist konfliktanfälliger sind als die anderen Kategorien. Die Werte von *Data* sind etwas geringer, aber sonst ziemlich proportional zu *GUI*. Der einzige Ausreißer ist *Control*, bei dem die Werte genau umgekehrt verteilt sind, da die *No Conflict*-Dichte höher liegt. Daraus lässt sich schlussfolgern, dass mit mehr *Time Effort* pro LOC bei *GUI* und *Data* mehr *Conflicts* entstehen, während bei *Control* jedoch mehr Zeit je LOC bessere Ergebnisse liefern. Alle drei Kategorien haben eine Gemeinsamkeit: Die Werte der *Conflict*-Dichte sind die Spiegelung der LOC Ergebnisse. Durch die Zuhilfenahme der Zeit sind die *Conflict* bzw. *No Conflicts* genau verdreht. Da die Ergebnisse auf unterschiedlich großen Klassen beruhen und somit nicht wirklich vergleichbar sind, müssen die Werte dementsprechend normalisiert werden.

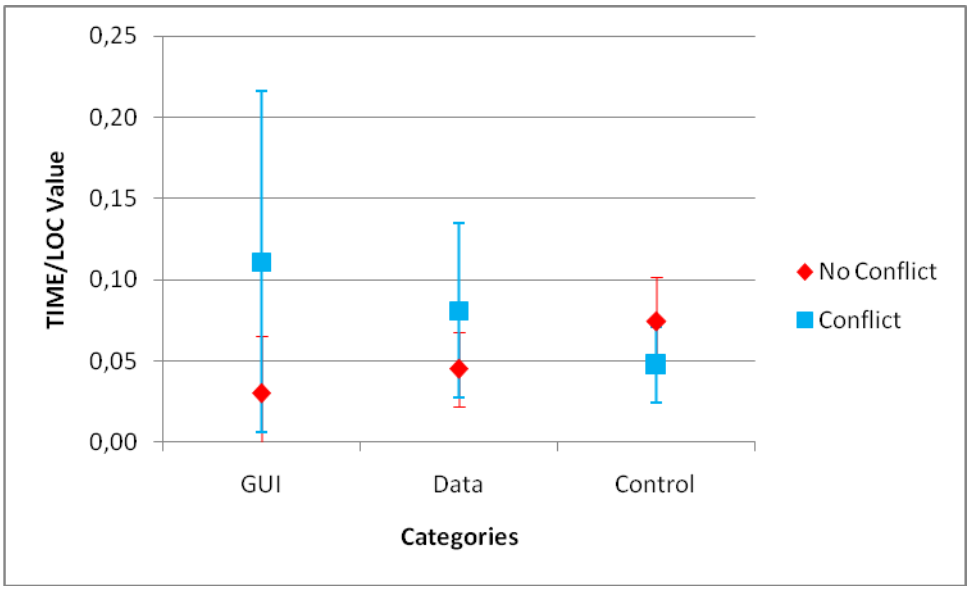

Abbildung 15: TIME/LOC im Verhältnis zu Conflicts

In [Eyge09] wurde die LOC Anzahl in vier Gruppen unterteilt, um eine statistische Signifikanz herauszufinden: 0-30 LOC, 30-100 LOC, 100-300 LOC, > 300 LOC. Aufgrund dessen und der Daten, die vorlagen, wurde ein neuer Wert X berechnet, um die Klassengröße dementsprechend normalisieren zu können. Für die Qualität wurde folgende Formel verwendet:

$$X = 0{,}296 * \ln(LOC) + 0{,}682$$

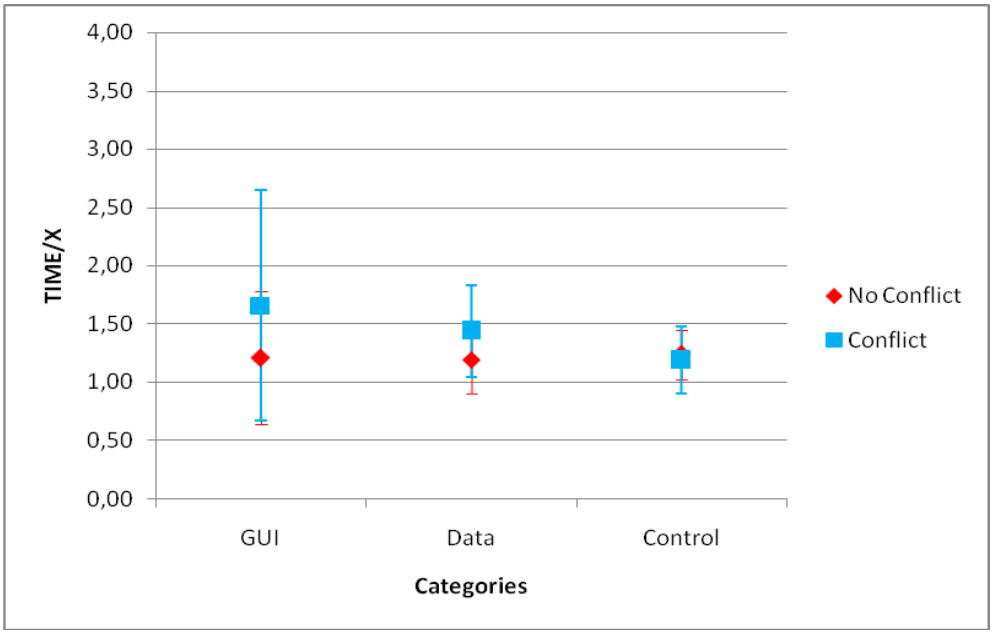

Abbildung 16: Normaliserung in Bezug zur Qualität

Durch die Normalisierung hat sich die Grafik minimal verändert: Alle Klassen haben nun mehr *Conflicts* als *No Conflicts* (vorher war *Control* alleinig mit weniger *Conflicts*), somit ist die Qualität überall mit durchschnittlich mehr etwas Zeit beeinträchtigt. *Control* bleibt zwar am wenigsten konfliktanfällig, jedoch haben sich die Werte der anderen Kategorien auch ziemlich genähert. Die Normalisierung hat die Intervalle etwas verkürzt und die Werte sind aber weiterhin absteigend, beginnend mit *GUI*, angeordnet. Auch mit der Normalisierung haben sich die Daten von *GUI* nicht wirklich verändert bzw. verbessert.

Während bei *GUI* der Durschnittswert beim Finden eines *Trace* höher und beim Nichtfinden niedriger liegt, so ist dies bei *Data* und *Control* genau der umgekehrte Fall. Sehr seltsam ist bei *GUI*, dass der Abstand zwischen den beiden Werten immens ist, während er bei den anderen direkt anschließt.
Man sollte eigentlich beim Finden eines *Traces* immer weniger Zeit brauchen: Sobald man am Anfang der Klasse eines gefunden hat, so kann man die Suche abschließen und zur nächsten Klasse wechseln, ohne sich den restlichen Code anzusehen. Beim Nichtfinden eines *Traces* ist der Benutzer mindestens die ganze Klasse durchgegangen bis er draufkam, dass sowie kein *Trace* vorliegt.

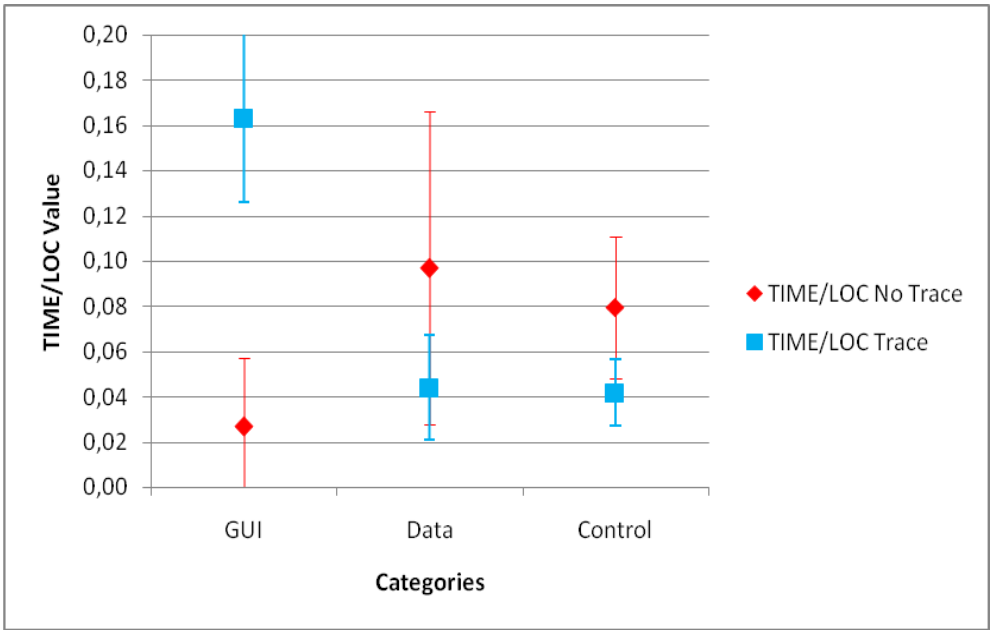

Abbildung 17: TIME/LOC Value im Verhältnis zu Traces

Wie wir wissen, hat *GUI* im Durchschnitt die größten Klassen. Somit sieht man auch in Abbildung 17, wenn ein *Trace* gefunden wurde, dass das seltsamerweise viel mehr Zeit brauchte als das Ausschließen. Anscheinend ist es sehr schwierig sagen zu können, dass ein *Trace* vorliegt.

Die obige erwähnte Theorie trifft somit nur auf *Data* und *Control* zu. Man brauchte also weniger Zeit, da man nicht die ganze Klasse durchlaufen musste. Sobald man aber keines gefunden hatte, musste man die gesamten Klassen durchgehen, was natürlich mehr Zeit verschlang. Bei *Data* tritt dieser Effekt ziemlich stark auf, was man am Intervall erkennen kann. *Control* ist hier etwas niedriger, was daran liegt, dass die Klassen auch im Durchschnitt etwas kleiner waren. Um die zwei Kategorien besser zu unterscheiden, müsste man die Werte auch noch einmal normalisieren. Hier wurde eine ähnliche aber abgewandelte Formale verwendet:

$$X=0{,}272*\ln(LOC)+0{,}141$$

wobei X wieder der invertierte Wert sei.

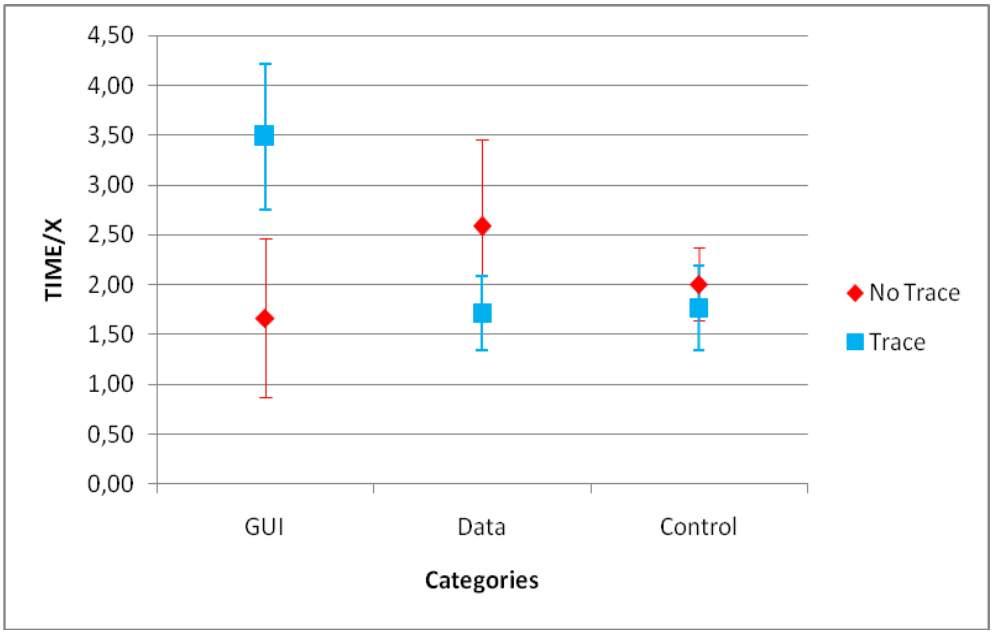

Abbildung 18: Normalisierung in Bezug zur Tracefindung

Die Gesamtwerte (egal ob *Trace* oder *No Trace*) sind ziemlich auf einer Linie, wobei *GUI* den meisten Aufwand beansprucht, dicht gefolgt von *Data* und *Control*. Die Werte sind so dicht beeinander, dass man sich die *Trace/No Trace*-Verteilung genauer ansehen muss.

In Abbildung 18 sieht man die resultierende Grafik, wenn der Aufwand ein *Trace* zu finden normalisiert wird, egal wie groß eine Klasse ist. Verändert hat sich bei *GUI* der Abstand, der geringer geworden ist. *Data* hat sich kaum verändert und *Control* macht nun keinen Unterschied mehr: Hier braucht man zum Finden und zum Ausschließen den selben *Time Effort*. Mit der Normalisierung bestätigt sich nur wieder, dass die *GUI* Kategorie schwerer ist, um ein *Trace* zu finden, während es bei *Data* leichter ist und bei *Control* kaum Unterschiede macht.

5.6. Conflict Agreements

Bei *Conflict Agreements* geht es darum, mit wieviel Prozent Übereinstimmung ein *Trace* bzw. ein *No Trace* bestimmt wurde. Es macht einen Unterschied, ob ein *Trace* zum Beispiel mit 100%, also von allen Testpersonen, als *Trace* identifiziert wurde, oder ob es gerade mit 60% Übereinstimmung gerade noch als *Trace* gelten darf.

Somit war der beste Fall, wenn 100% der Meinung waren, es sei ein *Trace* bzw. *No Trace* und der schlechteste Fall mit genau 50%, wenn genau die Hälfte das Eine oder das Andere meinten. Die Analyse der Daten stellte sich als durchaus komplex heraus, da nicht immer dieselbe Anzahl an Studenten eine Wertung abgaben, da ja die Möglichkeit bestand, das Feld freizulassen. Um Abhilfe zu schaffen, wurde ein Intervall generiert, damit sich die Daten zwischen 50 und 100% befanden.

Bei *Conflict Agreements* muss man beachten, dass es einen Unterschied macht, ob man mit 100% sagt *Trace* oder *No Trace*. Etwas auszuschließen ist immer leichter, als etwas direkt zuweisen zu können.

Die Daten beziehen sich auf jede einzeln zugewiesene Anforderung zu jeder Klasse (85 *Classes* * (*Requirements* + *GPC*)). Nach Betrachtung der Daten wie in Abbildung 19 ersichtlich, war klar, dass man die einzelnen *Trace* bzw. *No Trace* Daten auswerten musste, da so die Ergebnisse fast identisch und nicht recht aussagekräftig waren. Es war ersichtlich, dass die meisten Zustimmungen sich im Bereich der 100% aufhielten.

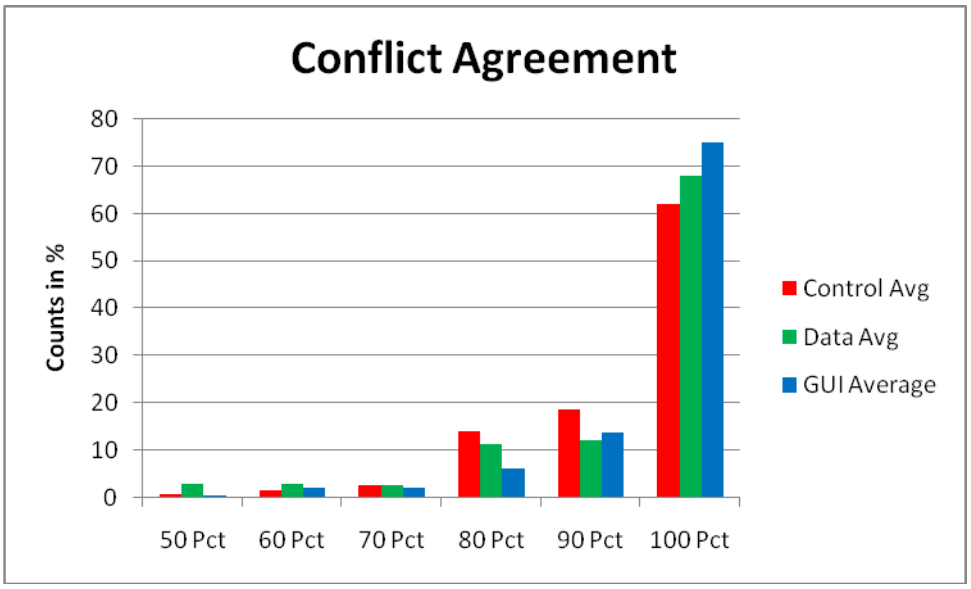

Abbildung 19: Conflict Agreements Total

Die Unterteilung der Daten gab schon etwas mehr Aufschluss. Während vorher die 50%-Marke niedrig gehalten wurde, sieht man, dass diese insgesamt so klein ist, da sie bei *No Trace* eigentlich kaum vorkommt. Wenn es einen 50:50 Conflict gab, dann fast nur in Bezug auf die *Traces*. Wie bereits erwähnt, ist etwas auszuschließen leichter, als etwas zu bestimmen. *GUI* ist die einzige Kategorie, die diesen Bereich noch am niedrigsten hält.

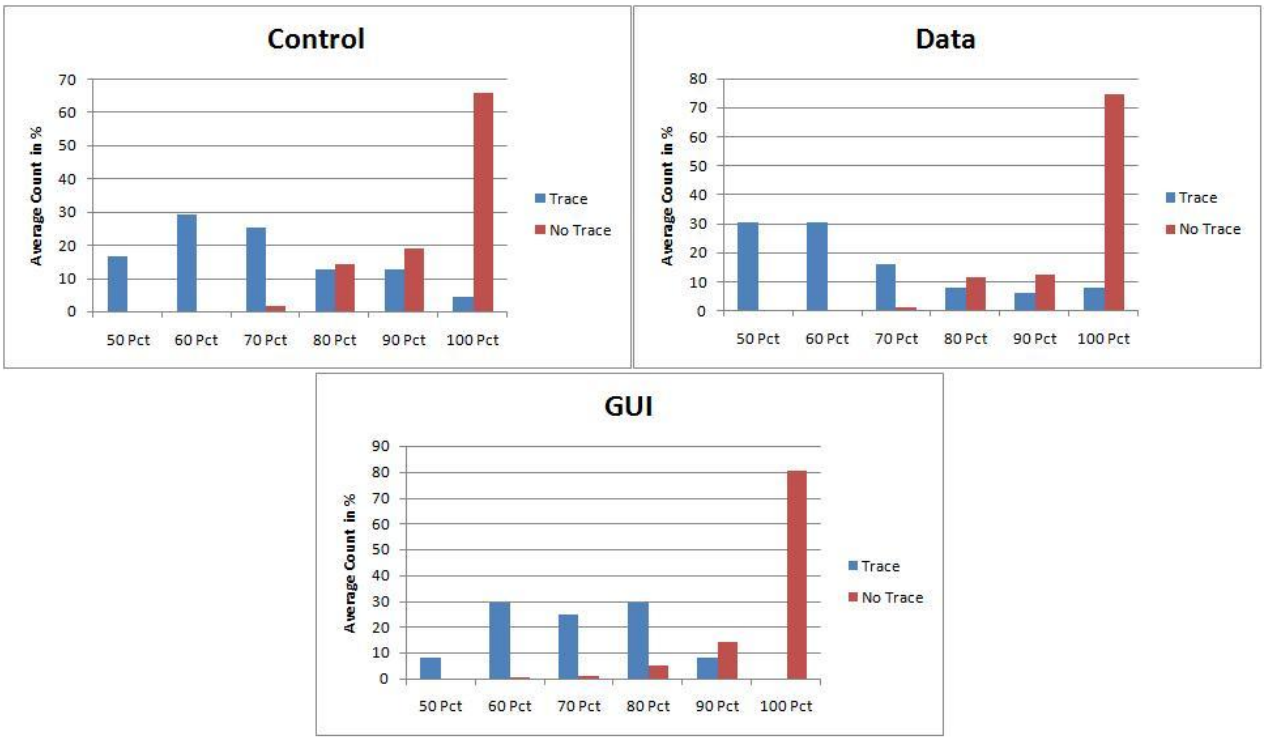

Abbildung 20: Conflict Agreements im Detail

Interessanterweise sind bei *GUI* alle 100%-igen Übereinstimmungen reine *No Trace* Daten. Somit sind zwar die 50:50 *Conflicts* niedriger als bei den anderen, jedoch die totale Übereinstimmung fehlt komplett. Nichtsdesdotrotz, egal welche Kategorie betroffen ist, alle weisen die selben Merkmale auf: 50, 60 und 70% des Übereinstimmungsgrades hält sich hauptsächlich im *Trace*-Bereich auf. Die besseren 80 und 90%, sowie die totale Übereinstimmung sind im *No Trace* Bereich.

Betrachtet man die einzelnen Klassen zusammen, mit ihren ganzen Anforderungen, wird die Theorie nur bestätigt, dass sich *Traces* im unteren Bereich (50 bis 70% *Conflict Agreement*) aufhalten, während *No Traces* weit darüber sind. Dies wies jedoch keine kategoriespezifischen Merkmale auf und war bei den Auswertungen ziemlich gleich verteilt.

5.7. Abschließende Analyse

Bei den Class/Methods *Conflicts* lieferte *GUI* die besten Ergebnisse. Jedoch war diese Wertung unabhängig von Maßzahlen. Während diese erste Auswertung noch gute Ergebnisse hatte, waren die darauffolgenden eher das Gegenteil.
In Bezug auf die Qualität war die Vorkommnisse der Werte ziemlich egal, denn *Conflicts* traten bei kleinen sowie großen Klassen auf, ebenso beim *Time Effort*. Mithilfe der Normalisierung ließ sich feststellen, dass die Qualität in dieser Kategorie, aufgrund der Messwerte, geringer ist, als bei den anderen Kategorien.
Ebenso beim Finden eines *Traces*, war es dieser Gruppierung ziemlich egal, ob viel oder wenig LOC bzw. viel oder wenig *Time Effort* vorlag. Bei Betrachtung beider Messewerte und der Zuhilfenahme der Normalisierung stellte sich heraus, dass es schwer ist in dieser Kategorie ein *Trace* zu finden, bzw. dass es einfach mehr Zeit benötigt.

Bei *Data* wurde die Qualität bei mittelgroßen LOC und mittlerer Zeit beeinträchtigt. Auch im Verhältnis der Messwerte zueinander sind die Werte ähnlich wie *GUI*, jedoch etwas besser.
Bei der *Trace*/*No Trace*-Verteilung sieht man zunächst, dass *Traces* bei größeren Klassen und mit weniger *Time Effort* eher gefunden werden. Ein *Trace* zu finden ist hier kaum ein Aufwand, während das Ausschließen eine Schwierigkeit darstellt.

Die Qualität spielt bei *Control* kaum eine Rolle. Zuerst sah es so aus, dass kleine Klassen und weniger *Time Effort* die Qualität stärkten. Jedoch nach Betrachtung der Verhältnisse zueinander spielte weder das eine noch das andere eine Rolle. Ebenso bei der *Trace*findung. Nach Betrachtung aller Werte und mit Einfluss der Normalisierung, spielten der *Time Effort* und der LOC Wert gar keine Rolle und somit konnten hier kaum signifikante Daten gewonnen werden.

Die *Conflict Agreements* bestärken nur, was in dem Experiment von [Egye08] festgestellt wurde: Dass durch die *No Trace* Votes die hohen Werte, hauptsächlich entstanden sind. Kategoriespezfisch findet sich hier nicht wirklich etwas Aussagekräftiges.

Traces zu finden stellt sich schwieriger heraus, als sagen zu können, dass *No Trace* vorliegt. Darum sind die 100%igen Übereinstimmungen fast immer *No Trace*, egal welcher Kategorie die Klasse angehört.

Während es anfangs für *GUI* am besten aussah, so würde ich dieser Kategorie den meisten Qualitätsverlust zuschreiben, aufgrund der verteilten Aufkommen von *Conflicts*. Wegen der *Trace*-Dichte und dem dazugehörigen Intervall, stellt es sich auch als Schwierigkeit heraus, sagen zu können, wann es nun wirklich leichter sein könnte ein *Trace* zu finden. *Data* und *Control* würde ich in im Verhältnis zur Qualität gleichsetzen, da ziemlich ähnliche Werte vorliegen. Bei der *Trace*findung gebe ich *Data* die bessere Wertung, da hier weniger *Effort* benötigt wird.
Qualitätsverlust geschieht bei jeder einzelnen Kategorie, bei einigen mehr bei anderen weniger. Schlussendlich würde ich *Data* als beste bzw. leichteste Unterteilung bestimmen, gefolgt von *Control*. *GUI* bildet das Schlusslicht mit eher weniger guten Ergebnissen.

6. Zusammenfassung

Traceability ist sicher gewinnbringend in langlebigen Software Systemen. Anforderungen sind so leichter mit dem Code zu verknüpfen und nachzuvollziehen. Die Einführung eines automatischen *Tracing* Tools wäre essentiell, um den immensen Aufwand den *Trace Recovery* bereitet, gering zu halten. Jedoch ist auch manuelles *Tracing* nötig, das, wie man sieht, sehr lange braucht, dafür genauer, aber auch vom Menschen abhängig und nicht unfehlbar ist. *Conflicts* sind in beiden Fällen unvermeidbar.

Egal wie komplex der Code ist, oder wie viele LOC eine Klasse vorweist, oder wie viel Zeit investiert wurde, den Code zu verstehen: *Conflicts* wird es immer geben.

Die Einteilung der Kategorien sollte Abhilfe schaffen, um die *Conflicts* im manuellen Bereich ausfindig zu machen, um somit vielleicht gewisse Punkte einhalten zu können, wie zum Beispiel mehr oder weniger Zeit zu investieren. Somit könnten diese *Conflicts* vielleicht kaum mehr vorkommen oder wenigstens die Anzahl verringert werden.

In der Zukunft wird sicher noch einige Entwicklung auf diesem Gebiet notwendig sein, damit die Vorteile, die *Traceability* bietet, flächendeckend bei der Softwareentwicklung, -wartung und -bearbeitung angewendet werden können.

Literaturverzeichnis

[Boga04] Bogaczyk, Matthias: *Traceability - Verfolgbarkeit sich ändernder Anforderungen im Softwareentwicklungsprozess*, Universität Duisburg-Essen, Wintersemester 2004/05

[Brci07] Brcina, Robert: *Arbeiten zur Verfolgbarkeit und Aspekte des Verfolgbarkeitsprozesses, Softwaretechnik-Trends*, Band 27 Heft 1, ISSN 0720-8928, Februar 2007

[Egye08] Egyed, Alexander; Graf, Florian; Grünbacher, Paul: *Recovering Trace Links between Requirements and Code: Understanding the Human in the Loop*, Institute of Systems Engineering and Automation, Johannes Kepler University Linz, Austria, 2008

[Egye09] Egyed, Alexander; Graf, Florian; Grünbacher, Paul: *Trace Recovery without a-priori System Knowledge: Two Exploratory Experiments*, Institute of Systems Engineering and Automation, Johannes Kepler University Linz, Austria, 2009

[Frank08] Franke, Stefan*: Traceability: Automatisierte Nachverfolgbarkeitvon Anforderungen*, 17.Februar 2008

[Gote94] Gotel O.; Finkelstein A.: *An Analysis of the Requirements Traceability,* London, 1994

[Knet02] Knethen, Antje; Paech, Barbara: *A Survey on Tracing Approaches in Practice and Research*, Fraunhofer IESE, Jänner 2002

[Mäde07] Mäder, Patrick; Philippow, Illka; Riebisch, Matthias: *Customizing Traceability Links for the Unified Process*, Technical University of Ilmenau, Germany, 2007

[Minz04] Minzenmay, Thomas: Qualitätsverbesserung durch Traceability, Paderborn, 3. Mai 2004

[Perr92] Perry, Dewayne E.; Wolf, Alexander L., *Foundations for the Study of Software Architecture*, Software Engineering Notes vol 17 no 40, Page 40, October 1992

[Pohl96] Pohl, Klaus: PRO-ART: *Enabling Requirements Pre-Traceability*, Aachen, 1996

[Tech01] Technologieberatung und Systementwicklung Hamburg: *Kriterien für Qualität von Software*, 2001, http://www.tse.de/papiere/ergonomie/Softwarekriterien.html, zugegriffen am 8.März 2009

[Tryg97] Tryggeseth, E.; Nytro, O.: *Dynamic Traceability Links Supported by a System Architecture Description,* International Conference on Software Maintenance, Oktober, 1997

[Wier95] Wieringa, Roel: *An Introduction to Requirements Traceability,* Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, 1. November 1995

[1] http://lehrerfortbildung-bw.de/kompetenzen/projektkompetenz/pm_software/ganttproject/, zugegriffen am 4. September 2009

[2] http://www.wiwi-treff.de/home/index.php?mainkatid=1&ukatid=13&sid=52&artikelid=2257&pagenr=0, zugegriffen am 4. September 2009

[3] http://www.computerwoche.de/heftarchiv/1991/36/1141579/, zugegriffen am 10. September 2009

Abbildungsverzeichnis

Tabellenverzeichnis