Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3515

Automatisierte Verarbeitung von Crash Dumps

Joerg Ploedereder

Studiengang:	Softwaretechnik
Prüfer/in:	PD Dr. rer. nat. Michael Schanz
Betreuer/in:	Dipl.-Inf. Joachim Ernst Vollrath
Beginn am:	9. Juli 2013
Beendet am:	8. Januar 2014
CR-Nummer:	D.2.5, H.1.2

Kurzfassung

Fehler in der Software kennt jeder, der mit Computern arbeitet, zur Genüge. Manche Fehler sind lediglich ein Ärgernis für den Nutzer, andere machen die Nutzung der Software unmöglich. Diese zweite Fehlersorte äußert sich darin, dass die Software abstürzt bzw. nicht mehr weiter nutzbar ist. Um solche Fehler bei einer bereits ausgelieferten Software zu beheben, muss der Hersteller vom Auftreten des Fehlers informiert werden. Hierzu kann sich der Hersteller der Nutzung von Crash Reports bedienen. Crash Reports sind Fehlerberichte, die im Fehlerfall an den Hersteller der Software gesandt werden und Daten enthalten, die Informationen zum Absturzzustand liefern.

Abhängig von der Anzahl der beim Hersteller eingehenden Fehlerberichte und der zu Verwaltung deren zugeteilten Entwickler, wird es nach einiger Zeit mühsam und vor allem zeitaufwändig, die Übersicht über bekannte Fehler zu behalten und die neu eingehenden manuell zu verarbeiten.
Ziel dieser Diplomarbeit ist es, ein System zu entwickeln, das die Verarbeitung von automatisch erstellten Fehlerberichten von manueller Arbeit in einen halb-automatischen Vorgang überführt.

Der Sinn liegt hierbei in der Effizienzsteigerung und Arbeitseinsparung für die Entwickler. Zur Zeit muss jeder eingehende Fehlerbericht noch von Hand manuell überprüft werden, der Fehler bzw. die fehlererzeugende Methode gefunden und danach einem bekannten Fehler im Bug-Tracking-System zugeordnet oder darin als neu klassifiziert werden. Ebenso ist es möglich, dass vergleichbare Fehler nicht als ähnlich vom Menschen klassifiziert werden, da ihnen das Wissen um den anderen Fehler fehlt und ihre subjektive Suchabfrage im Bug-Tracking-System kein Ergebnis liefert.

Die Arbeit stützt sich dabei auf die bekannten Theorien und Umsetzungen für Fehlerzuordnung und soll ferner den genauen Grad der Automatisierung feststellen, der für ein System geeignet ist, das sowohl auf händisch erstellten Fehlermeldungen, als auch auf automatisch generierten Fehlerberichten basiert. Die für die Zuordnung bekannter Fehler verschiedenen bekannten Techniken sollen auf ihre Nützlichkeit hin evaluiert werden. Das nützlichste Verfahren soll dann in die Implementierung des Gesamtsystems einfließen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Fehler liegen in der Natur des Menschen und da Software etwas von Menschen geschaffenes ist, ist es unvermeidbar, dass sie Fehler enthält. Es bleibt nur zu sehen, zu welchem Punkt des Software-Lebenszykluses diese Fehler entdeckt werden.
Diese Diplomarbeit geht nicht von Entwurfsfehlern aus, sondern von Fehlern, die durch fehlerhaften Code oder fehlerhafte Anwendung entstehen. Die zeitlichen Betrachtungspunkte im Software-Lebenszyklus sind also eingeschränkt.

Während der Entwicklung stehen dem Entwickler meist die Debug-Ausgabe der Entwicklungsumgebung, die ihm auftretende Fehler aufzeigt, oder andere Möglichkeiten, um im Falle eines auftretenden Fehlers die Fehlerquelle einzuschränken, zur Verfügung. In der Testphase kann abhängig von den Testverfahren, ähnlich wie in der Entwicklungsphase, der Fehler in seiner Quelle schnell eingeschränkt werden. Hier sind es aber auch nicht mehr die Debug-Ausgaben des Programms, die dem Entwickler Auskunft über das Versagen geben, sondern die Tests bzw. Fehlerberichte, die das Programm im Testlauf schon erzeugt.

Bei ausgelieferten Produkten ist es nicht mehr so einfach, die Fehler zu bestimmen und zu kategorisieren. Hier muss man sich des Systems der Fehlerberichterstattung behelfen, wenn man feststellen will, warum das Programm unvorhergesehen den Dienst quittiert hat.
Im Falle eines Absturzes wird eine Fehlerbericht-Datei, oder auch Crash Dump File, erstellt, die den laufenden Prozess mit all seinen zugehörigen Stacks zum Zeitpunkt des Absturzes nach bester Möglichkeit je Thread aufzeichnet. Anschließend wird diese Dump File via automatischem Sendeverfahren zur Auswertung an den Software-Hersteller geschickt.
Dieses Verfahrens bedienen sich die großen Software-Firmen schon lange. Microsoft nutzt sein hauseigenes Windows Error Reporting (kurz WER) System und stellt dies auch anderen Software-Herstellern zur Verfügung [MSW]. IBM nutzt sein eigenes System. Und Mozilla nutzt sein Bugzilla System [Bug] - um nur einige Systeme zur Fehlerermittlung zu nennen.
Während Microsoft und IBM schon immer ein automatisches System zur Berichterstattung verfolgt haben, beruhte die Fehlermeldung bei Mozilla anfangs noch darauf, dass der Nutzer manuell ein Bug-Tracking-System aufrief und seinen Fehler mit Schilderung des eigenen vermeintlichen Vorgehens bis zum Auftreten des Fehlers hin meldete. Davon ist Mozilla jedoch schon länger abgekommen und lässt sich die Fehlermeldungen nun automatisch erstellen und auf Nutzerwunsch auch mit Kommentar in ihr Bug-Tracking-System übertragen.
Sei es nun durch manuelle oder automatische Fehlerberichterstattung, immer mehr Firmen nutzen diese Berichte, um ihre Qualität zu verbessern.

1.1 Problemstellung

Das Problem, welches sich nun stellt, ist die korrekte Zuordnung des im Fehlerbericht enthaltenen Fehlers zu einem Eintrag im Bug-Tracking-System. Je nach Status des Eintrags gibt es bereits eine mögliche Lösung für den Fehler oder der Fehlerbericht dient als Referenz für den Entwickler, der diesen Fehler noch beheben soll.

Ein komplett menschliches Zuordnungsverfahren hat in erster Linie das Problem der Mannbindung im Sinne des zeitlichen Aufwandes.
Die eingehenden Fehlerberichte zu sichten und zu analysieren benötigt viel Zeit. Ist der Fehler gefunden, beginnt in der Regel das Suchen im Bug-Tracking-System nach einem korrespondierenden Fehler. Diese Suche kostet weitere Zeit, da sich der bearbeitende Entwickler nicht immer erinnern kann, ob und wo der Fehler schon hinterlegt ist.
Meist wird mit der Verteilung dieser analytischen Aufgabe auf mehrere Entwickler das Ziel verfolgt, den zeitlichen Aufwand pro Kopf zu reduzieren. Leider erhöht sich damit aber auch die Suchzeit und vor allem die Chance einen dem Bug-Tracking-System bekannten Fehlerbericht nicht zu finden, da die subjektiven Suchmethoden und die Unwissenheit über die Arbeit der Anderen die Suche erschweren.

Und so kommt der zweite Aspekt ins Spiel, nämlich die Möglichkeit bei der Zuordnung Fehler zu machen.
Egal, ob eine oder mehrere Personen zuständig sind, allein durch die Vergesslichkeit des Einzelnen besteht die Gefahr, einen schon bestehenden Fehlereintrag nicht zu finden. Mit zunehmender Personenzahl nimmt sie überproportional zu, da die Einträge der anderen meist nur durch das Ergebnis der Suchanfrage für den Einzelnen ersichtlich sind, selbst wenn gewisse Standards definiert wurden.
Diese Gefahr, einen Eintrag nicht zu finden, zieht, verglichen mit dem Zeitaufwand, auch noch ein ganz anderes Problem in Form von Kostensteigerung mit sich, nämlich die Fehlpriorisierung von Fehlern an Hand der Anzahl ihrer Meldungen. Diese Fehlpriorisierung kann das Erkennen und zeitige Beheben wirklich kritischer Fehler verhindern. Um diesem vorzubeugen, ist es essenziell, dass die Fehler so korrekt wie möglich kategorisiert werden und somit klar ersichtlich wird, wie oft welcher Fehler gemeldet wurde.

1.2 Aufgabenstellung

Um sowohl den Zeitaufwand zu verringern, als auch die Zuordnungsgenauigkeit zu optimieren, soll im Rahmen dieser Diplomarbeit ein System entwickelt werden, welches die eingehenden Fehlerberichte verarbeitet, mit der Datenbank der bekannten Fehlerberichte abgleicht und eine Zuordnung oder Neueintragung vornimmt.
Da sich in den zu Grunde liegenden Forschungsarbeiten aber auch abzeichnet, dass voll automatisierte Verfahren nicht immer die gewünschte Treffergenauigkeit haben, soll eine menschliche Zuordnungskontrolle stattfinden.

1.3 Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen und verwandte Arbeiten: Hier werden die Grundlagen und verwandte Arbeiten dieser Arbeit beschrieben.

Kapitel 3 – Aufbau des Systems: Hier wird ein Überblick über die Bestandteile des Systems und ihre einzelnen Funktionsweisen gegeben.

Kapitel 4 – Ergebnisse: Hier werden die Ergebnisse aufgezeigt, die ein vollautomatisches System und das geplante System geliefert haben, und in Bezug zueinander und zu dem komplett manuell basiertem Verfahren aufgezeigt.

Kapitel 5 – Zusammenfassung und Ausblick fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

2 Grundlagen und verwandte Arbeiten

Ein Fehler in der Software, welcher zu einem kritischen Abbruch führt, also ein tatsächlicher Mangel in der Software, wird heute immer häufiger in der Erstellung eines Dump Files enden.

Im Falle der Software der Auftraggeberfirma für diese Diplomarbeit basieren die Fehler meist auf Speicherschutzverletzungen. Diese Speicherschutzverletzungen werden durch Zeigerverweise auf ungültige Speicheradressen, sowie Zugriffe auf geschützte oder nicht erreichbare Ressourcen hervorgerufen.
Manche dieser Fehler klassifizieren sich auch als sogenannte Heisenbugs [Hei], also Bugs, die ihr Verhalten verändern oder gar ganz verschwinden, wenn man versucht sie nachzuvollziehen. Ihr Name nimmt Bezug auf Werner Heisenberg, einen Physiker, der die Effekte der Quantenmechanik und die Tatsache erforschte, dass schon die Beobachtung eines Systems das System verändert. Andere Fehler sind jedoch wiederholbar und konkret in ihrem Auftreten.
Diese gilt es zu identifizieren und zu beheben, doch dafür muss man erst einmal von ihrer Existenz wissen bzw. erfahren.

Die Entwicklung zur Nutzung von Dump Files geht daraus hervor, dass in den meisten Fällen die Auskünfte des Nutzers beim Versagen der Software gegenüber dem Service Dienstleister eher unvollständig und unzureichend sind, um den Fehler genau festzustellen. Dies ist keine Böswilligkeit, sondern zum Einen die ungenaue Erinnerung des Nutzers an seine Aktionen, die zum Absturz führten, zum Anderen eben die Tatsache, dass die meisten Aktionen nicht nur einen Funktionsaufruf zur Folge haben, sondern eine ganze Abfolge. Somit ist erkennbar, dass alleine die vom Nutzer gegebene Auskunft zwar einen Fehler melden kann, aber die Ursachen dennoch aufwendig gesucht werden müssen.

Vorab sollen hier die wichtigsten Begriffe geklärt werden, die im Umgang mit Dump Files eine Rolle spielen:

Dump File Ein Dump File ist die Abbildung eines Auszugs des Speichers bezüglich eines Prozesses zum Zeitpunkt des Absturzes und listet die aktiven Vorgänge ihrem Thread zugeordnet auf.
Es wird automatisch erstellt und kann je nach Programmierung des Absturzprotokolls automatisch an den Software-Hersteller übermittelt werden oder aber auch auf dem Rechner verbleiben, bis der Nutzer sich entschließt, die Datei selbst an den Software-Hersteller zu übermitteln.

Es muss bei der Akquise der Dump Files zwischen zwei verschiedenen Systemen unterschieden werden: „in-process Error Handling“ und „out-of-process Error Handling“ [pro].
Der Vorgang des „out-of-process Error Handlings“ basiert darin, dass das Betriebssystem den laufenden Prozess überwacht, das sogenannte Watchdog-Prinzip kommt hier zur Anwendung. Dieser Watchdog, zu deutsch Wachhund, bildet einen eigenen Prozess, der alle anderen oder auch nur eine bestimmte Menge an Prozessen von außen überwacht. Endet einer der überwachten Prozesse unerwartet, „schlägt“ der Watchdog an und sammelt automatisch die Dumpdaten ein, die beim Versagen des Prozesses automatisch entstehen und schickt diese hernach an eine systemabhängige Sammelstelle. Nach diesem Prinzip fungiert das Windows Error Reporting System [MSW]. Der Software-Hersteller muss sich nun nur noch einen Zugang zu dieser Sammelstelle zulegen, um die eingehenden Dump Files zu erhalten.
Der Vorgang des „in-process Error Handlings“ wird im abstürzenden Prozess selber ausgelöst und abgehandelt. Dieses Prinzip ermöglicht es dem Software-Hersteller seine eigene Sammelstelle für die Dump File Sendungen anzugeben und auch den Inhalt des Dump Files zu reduzieren bzw. den kompletten E-Mail-Inhalt nach seinen Vorstellungen aufzubauen. So kann er zum Beispiel auch den Nutzer nach weiteren Informationen fragen. Der große Nachteil am „in-process Error Handling“ liegt darin, dass es eben aus dem abstürzenden Prozess heraus vollzogen wird. Man kann nie wissen, in welchem Zustand der Prozess tatsächlich beim Zeitpunkt des Absturzes ist. Es kann sein, dass der Absturz geregelt von statten geht, wodurch ein Dump File vom Prozess geschrieben werden kann. Möglich ist auch, dass der Zustand so kritisch ist, dass der Prozess nicht mehr in der Lage ist, ein verwertbares Dump File zu erstellen. Nicht verwertbare Dump Files können durch fehlerhaften Speicher oder durch vorzeitiges vollständige Beenden des Prozesses entstehen. Fraglich ist auch, ob der Prozess den Sendevorgang noch erfolgreich beenden konnte.

Bei der Entscheidung für ein Fehlerberichterstattungsverfahren muss man abwägen, ob man sich auf die Sammelverfahren anderer Hersteller verlassen will und dafür immer Daten erhalten will, oder lieber ein genau festgelegtes Schema an Daten, die dafür aber nicht immer brauchbar sind, erhalten will bzw. auch das Risiko hat nie welche zu erhalten. Das Prinzip, auf dem die Fehlerberichterstattung der forschungsauftraggebenden Firma basiert, ist das „in-process Error Handling“-Prinzip.
Für das automatische Auswertungssystem, welches hier im Rahmen der Diplomarbeit entwickelt wurde, ist es allerdings, bis auf eine mögliche Erkennung von korrumpierten Dump Files, von Anfang an irrelevant, welches Verfahren gewählt wurde.

Um ein Dump File letztlich auszulesen, ist ein Debugger nötig, sowie die Symbole oder andere Dateien, die anstelle der Symbole noch weiteren Aufschluss über Debug- und Projektzustandsinformationen geben, damit der Debugger die korrekten Referenzen setzen kann. Die Dateien, die weiteren Aufschluss über Debug- und Projektzustandsinformationen geben können, sind abhängig von der Entwicklungsumgebung. Die für die Software, deren Fehlerberichte im Rahmen dieser Diplomarbeit verarbeitet werden sollen, verwendete Entwicklungsumgebung ist Microsoft Visual Studio. Ihre erzeugten Dateien für diesen Zweck sind .pdb-Dateien [MSV].

Prozess Ein Prozess bezeichnet das Ausführen eines einzelnen Programms. Jeder Prozess führt eine Reihe von Threads aus, in denen es zu Funktionsaufrufen kommt und die Funktionalität der Software umgesetzt wird. Einfach gesagt, bedingt der Aufruf eines beliebigen ausführbaren Programms den Start eines klar identifizierbaren Prozesses, der mit Beendigung des Programms auch beendet wird.

Thread Threads dienen der Aufteilung von Funktionalitäten des Prozesses. Ein Thread wird für eine Ausführungsreihenfolge ins Leben gerufen und endet, wenn die Reihenfolge durchlaufen ist, mit der Abgabe des Ergebnisses an den Prozess.
Im Zeitalter der mehrkernigen CPUs spielt Threading eine wichtige Rolle, da Threads es ermöglichen, dass Vorgänge des Prozesses auf die verschiedenen Kerne verteilt werden können und so echte Parallelität erreicht werden kann.
Jeder Thread bedient sich auch bei seiner Verarbeitung verschiedener Register und unabhängiger Call Stacks. Somit kann immer genau nachvollzogen werden, welcher Thread sich wo in seiner Verarbeitung befindet, was bei der Fehlerdiagnose sehr wichtig ist. Es gibt den Main-Thread, also der Hauptstrang an Funktionalität, der direkt aus dem Prozessaufruf hervor geht. Der Main-Thread kann zusätzlich zur Durchführung seiner eigenen Operationen andere Stränge hervorrufen und von ihnen ihre Ergebnisse erhalten. Diese vom Main-Thread erzeugten Nebenstränge werden in der Fachsprache als Threads bezeichnet.

Call Stack Call Stack, zu deutsch auch Kellerspeicher, ist das Konzept eines Speichersystems, das nach dem Last-In-First-Out Prinzip fungiert. Im Rahmen eines Prozesses wird dem Prozess und jedem Thread von der Laufzeitumgebung sein eigener Stack zur Verfügung gestellt.
Vom Thread aufgerufene Funktionen werden in den Kellerspeicher in der Reihenfolge ihres Aufrufens geschrieben. Ruft eine Funktion eine weitere Funktion auf, wird diese Funktion über die sie aufrufende Funktion „gestapelt“. Erst wenn die Funktion abgehandelt ist, verlässt sie den Kellerspeicher und gibt den Zugriff auf die Funktion darunter wieder frei.
Kommt es zu einem kritischen Versagen der Software, so werden alle mit dem Prozess und dessen Threads verbundene Stacks in ein Dump File gespeichert und der Absturz verursachende Stack wird als Call Stack darin abgebildet.
Ein Call Stack im Rahmen eines Dump Files bezieht sich also auf den Kellerspeicher des Absturz verursachenden Threads, mit der Stapelung der Funktionsaufrufe bzw. Frames zum Zeitpunkt des Absturzes. Da der Debugger durch den Call Stack lesen muss, listet er als Erstes die zuletzt aufgerufene Funktion auf und zum Schluss die Funktion, die als Initialfunktion in dem Thread einging.

Frame Ein Frame bezeichnet im Rahmen dieser Arbeit die Wissensrepräsentation bezüglich eines einzelnen Funktionsaufrufes, der in einem Call Stack aufgeführt wird und durch einen Debugger entschlüsselt wurde.
Nebst dem Funktionsaufruf enthält die Frame-Ausgabe eines simplen Debugging-Vorgangs die Zeilennummer bzw. den Offset bezüglich des Einsprungpunktes der aktuellen Funktion im Maschinencode. Ein Frame kann auch noch mehr Daten enthalten, wie später in dieser Arbeit gezeigt wird. Diese Daten erhält man allerdings nur bei einem aufwendigeren Debugging-Vorgang.

Auf Grund der Lesereihenfolge beim Auslesen eines Call Stacks ist der Frame mit dem Index 0 die Repräsentation des Wissens der zuletzt aufgerufenen Funktion. Je höher der Index des Frames ist, desto weiter liegt er in Richtung Beginn des Threads.

Debugger Ein Debugger ist ein Software-Werkzeug, das Entwickler benutzen, um Fehler zu finden und zu diagnostizieren. Häufig wird er als Bestandteil einer Entwicklungsumgebung benutzt, um Fehler zu beheben oder um das Entwickelte zu testen. Es gibt allerdings auch alleinstehende Debugger, die universeller einsetzbar sind, da sie nicht wie entwicklungsumgebungsabhängige Debugger an vorgegebene Entwicklungssprachen gebunden sind. Bei der automatischen Fehlerberichtserkennung werden diese zur Diagnose von Fehlern benutzt.

Symbole Symbole oder auch Debugsymbole sind notwendig, damit beim Vorgang des Debuggings korrekte Zuordnungen bezüglich der Funktionsaufrufe im Call Stack vorgenommen werden können.
Da Software beim Compilieren ihre menschliche Lesbarkeit und meist auch ihre klare Struktur verliert, können anhand eines Dump Files ohne Symbole nur die Register der Maschine ausgelesen werden. Die Daten aus den Registern sind für Menschen ein nur schwer verständliches Ergebnis. Es ist allerdings möglich beim Compilieren Referenzpunkte und zusätzliche Informationen setzen zu lassen, welche als Symbole bezeichnet werden.
Anhand dieser Symbole ist es möglich beim Debugging-Vorgang Referenzen auf die aufgerufenen Stellen im für den Menschen lesbaren Code zu setzen. Somit erhält der Mensch wieder für ihn verständliche Auskünfte und Verweise auf seinen Code.

.pdb-Dateien .pdb-Dateien, auch Programmdatenbankdateien, sind letztlich nichts anderes als Symboldateien. Sie enthalten sowohl Debug- wie auch Projektzustandsinformationen, die eine genaue Verknüpfung des auszulesenden Dump Files mit dem dem Entwickler vorliegenden Source Code ermöglichen und dadurch auch Sprünge an exakt die Stelle im Code, die gerade vom Frame aufgerufen ist.
Im Regelfall wird beim Debuggen durch einen Entwickler auf die Symboldateien für Funktionalitäten von Dritten zugegriffen, während der Debugger für die vom Entwickler eigenen Prozessanteile die Programmdatenbankdateien zu Rate zieht. Das .pdb-Format ist dabei eine Microsoft eigene Entwicklung für ihre Visual Studio .NET Entwicklungsumgebung [pdb].

Dump Files sind in den meisten Fällen die einzige Möglichkeit für Entwickler, ihre ausgelieferte Software gezielt auf auftretende Fehler hin zu untersuchen. Sollte der Nutzer also dem Versand des erstellten Dump Files, der soeben kritisch beendeten Software, an den Hersteller zustimmen, erhält der Entwickler die Möglichkeit Einblick in den Fehler zu nehmen, ohne sich dabei auf die Beschreibung des Nutzers zum Fehlerhergang verlassen zu müssen.

Diese Dump Files müssen dann natürlich auch analysiert werden. Dazu stehen verschieden aufwendige Werkzeuge zur Verfügung, angefangen beim Debugger der Entwicklungsumgebung, die für die Softwareentwicklung genutzt worden waren, bis hin zu beispielsweise Windows höchst eigenem Windows Debugger [Win]. Jedes dieser Tools hat seine Vorzüge.
Die Nutzung der Entwicklungsumgebung lässt direkte Sprünge zu den Stellen im Source

Code zu, die in den Frames aufgeführt werden, was die direkte Fehlerbehebung durchaus beschleunigen kann. Allerdings hat die Analyse in der Entwicklungsumgebung zur Fehlerfeststellung bzw. zur Fehlerortung bereits einen recht hohen Zeitaufwand bis alle nötigen Dateien geladen sind. Daher ist die Nutzung der Entwicklungsumgebung alleine zur Feststellung des Fehlers auf Grund des Zeitaufwandes nicht sehr geeignet.

Windows Debugger Der Windows Debugger ist für die einfache Fehlerortung deutlich schneller in seiner Verarbeitung des Dump Files. In seiner Grundeinstellung erhält man Auskunft über den Thread, der zum Absturz führte, in Form einer Auflistung der Frames des verursachenden Call Stacks. Der Debugger ist aber in seiner Darstellung noch erweiterbar, sodass alle Threads gelistet werden können und bietet weitere nützliche Einstellungen zur Erweiterung der Fehlerortung. Dabei ist der Windows Debugger in seiner Verarbeitungsgeschwindigkeit meist deutlich schneller, weil er auf aufwendige grafische Oberflächen verzichtet, da es sich um eine Konsolenanwendung handelt. Ferner lässt sich der Windows Debugger durch Parameterübergabe konfigurieren und steuern, wodurch er zu einem idealen Werkzeug zur Automatisierung des Fehlerberichtsauslesens wird.
Ob der Windows Debugger nun direkt vom Entwickler benutzt wird und ihm so den Inhalt eines Dump Files in der Console ausgibt oder aber automatisiert Dump Files ausliest und entsprechende Dateien zum Einsehen anlegt, ist letztlich für die einfache Fehlersuche irrelevant. Abbildung 2.1 zeigt eine Beispielausgabe der Console durch den Windows Debugger. Mit diesem Wissen, welches der Windows Debugger bereitstellt, kann sich der Entwickler nun daran machen, den Fehler zu beheben.
Das in dieser Arbeit zu entwickelnde System wird sich auf die automatisierte Nutzung des Windows Debuggers stützen.

Die Abbildung 2.1 zeigt die allgemeine Fehlerinformation, eine „Access violation“, in den oberen Zeilen und darunter die Ausgabe des Stacks. In der Ausgabe des Stacks aufgeführt befindet sich in jeder Zeile ein Funktionsaufruf. Gefolgt wird dieser Aufruf von einer Zeilenangabe in eckigen Klammern, sofern diese Funktion vom Debugger auf eine vorhandene .pdb-Datei abgebildet werden konnte. Abschließend zeigt die Zeile den Offset-Wert an. Der Offset wird auf die aus einer Symbol-Datei bekannte Anfangsposition der in dieser Symbol-Datei exportierten Funktion aufgerechnet und zeigt damit die Position im Maschinencode zur aufgerufenen Funktion auf.
Ist eine Zeilenangabe gegeben, so ist der Offset Wert unwichtig, da nur die Zeilenangabe tatsächlich vom Menschen direkt auf den Code übertragen werden kann, um die Stelle zu finden, an dem sich der Funktionsaufruf befindet. Der Offset hingegen gibt nur den Abstand zur Anfangsadresse der Funktion an. Für die Nutzung des Offsets bezüglich der genauen Code Position bzw. Zeile, die aufgerufen wird, bräuchte man einen Decompiler, um den Prozess aufzuschlüsseln und Einsicht zu erhalten. Allerdings erhält man den Offset auch ohne Zeilenangabe, wenn sich die Daten allein aus einer Symboldatei ableiten und keine .pdb-Datei zur Referenz zur Verfügung steht.
Die oberste Zeile ist der letzte Funktionsaufruf, der in dem Thread vor dem Absturz getätigt wurde.

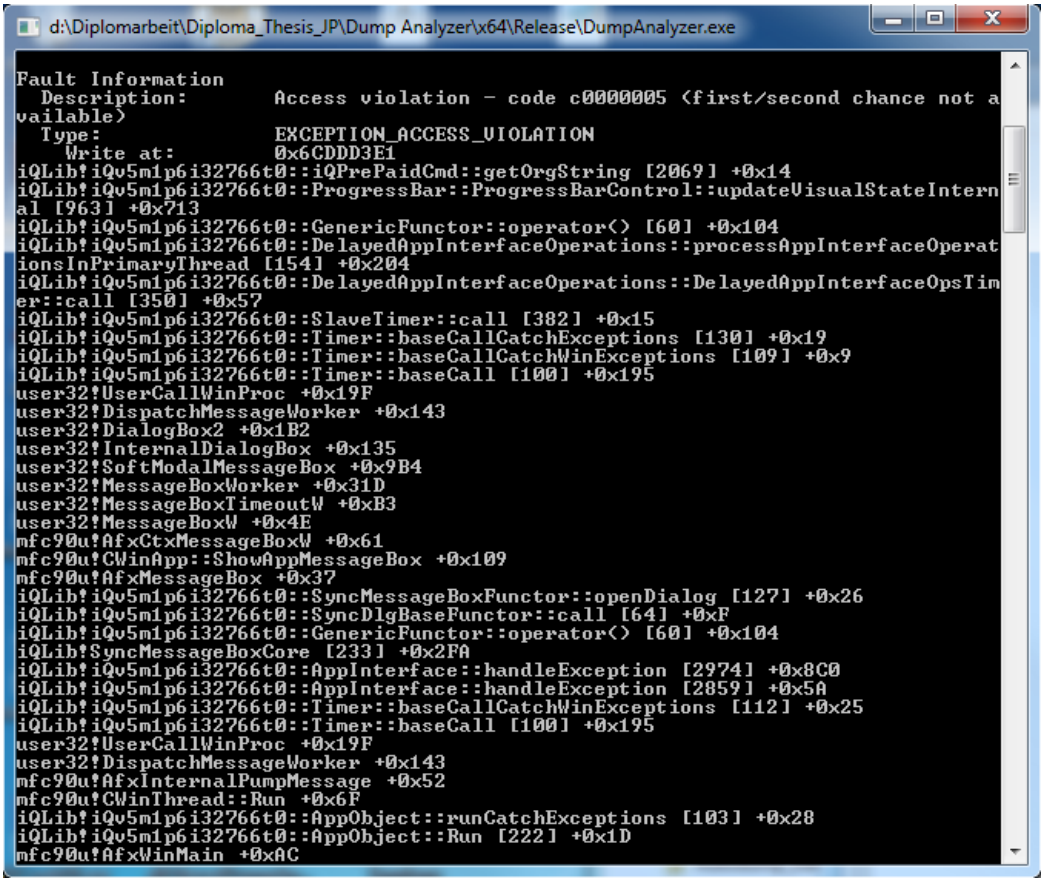


Abbildung 2.1: Einfache Windows Debugger Ausgabe in der Console

Die momentan gängige Vorgehensweise des Entwickler-Teams, die zu dieser Diplomarbeit geführt hat, besteht darin, dass automatisch erstellte Absturzmeldungen via E-Mail durch den Nutzer an eine vorgesehene Adresse gesendet werden. Eine solche Absturzmeldung besteht aus Daten, die der Nutzer zum Teil vorab bei der Installation der Software bei sich festgelegt hat, wie seine Identifikation und die gewählte Software selbst. Des weiteren kann der Nutzer direkt vor dem Absenden des Fehlerberichtes ergänzende Daten angeben und auch sein Vorgehen beschreiben. Diese Daten sind alle im E-Mail-Text enthalten. Ferner ist der E-Mail noch das zugehörige Dump File als Anhang beigefügt.
Auf diese Adresse haben zwei Entwickler verwaltenden Zugriff und sie müssen die eingehenden E-Mails einzeln bearbeiten, indem sie die möglichen vom Nutzer erweiterten Informationen bezüglich des Absturzhergangs aus dem E-Mail-Text entnehmen und das angehängte Dump File abspeichern. Schließlich kennzeichnen sie die E-Mail mittels der Kategorisierungsoption des E-Mail-Programms als „Bearbeitet“, damit der Kollege um die Bearbeitung durch den anderen weiß. Diese Kategorisierung ist auch dem Entwickler beim Überprüfen der E-Mails dienlich, um festzustellen, wie viele Fehlerberichte eine Zuordnung erwarten.
Hiernach wird das Dump File mit der Entwicklungsumgebung Visual Studio geladen und

mit den revisionsabhängigen .pdb-Dateien verknüpft, bevor die Debugging-Funktion von Visual Studio eingesetzt wird. Nachdem Visual Studio das Dump File verarbeitet hat, kann der Fehler nun von dem bearbeitenden Entwickler an Hand des Call Stacks und der Aufführung des Codes dazu analysiert werden.

Ist der Fehler ermittelt, ob früher durch Nutzermeldung oder nun durch den Eingang eines Dump Files, muss es eine Möglichkeit geben, diesen Fehler festzuhalten und das Entwicklerteam über dessen Existenz oder das wiederholte Aufkommen dieses Fehlers zu informieren. Hierzu gibt es verschiedenste Bug-Tracking-Systeme, die sich mehr oder minder nach Anforderungen der Entwickler oder der Firma für diese Aufgabe eignen.

Bug-Tracking-System Ein Bug-Tracking-System ist dadurch gekennzeichnet, dass es den Entwicklern die Möglichkeit gibt, auftretende Fehler in einer Datenbank abzulegen und jederzeit auch wieder suchen und einsehen zu lassen. Während manche Systeme nur das Verwalten der Fehlerberichte selbst als Aufgabe haben, gibt es auch Systeme, die den Produktionsverlauf unterstützen, in dem sie die Fehlerbehebung an Entwickler zuweisbar machen, oder automatisch eine Fehlerpriorität zur Behebung anhand vorab definierter Richtwerte festlegen.

In dieser Diplomarbeit wird das „Mantis Bug Tracking“ [Man] System verwendet.
Dieses System lässt eine Dokumentation der einzelnen Fehler zu, dabei kann nebst Fehlerbeschreibung und Entwicklerzuweisung auch die Schwere des Fehlers und sein Behandlungsstatus für jeden Fehlereintrag vermerkt werden. Weitere Werte sind auch einstellbar, spielen aber für die Diplomarbeit meist keine Rolle. Zusätzlich zu den beschreibenden Eigenschaften des Mantis-Eintrages für einen gemeldeten Fehler ist es möglich Dateien an jeden Fehlereintrag anzuhängen, um dem Entwickler eine Hilfestellung bei der Fehlererkennung zu geben.

In dieser Datenbank suchen die beiden Entwickler, nach der Analyse des Fehlers aus der von ihnen bearbeiteten E-Mail, nach einem korrespondierenden Fehlereintrag. Wird ein Eintrag gefunden, so laden die Entwickler das analysierte Dump File in diesem Eintrag hoch und ändern wenn nötig noch einige Einträge, wie Behandlungsstatus oder die Schwere des Fehlers. Finden sie keinen korrespondierenden Eintrag, so wird ein neuer Eintrag angelegt und mit der Datei und weiteren Werten versehen.
Um eine schnellere und vor allem erfolgreichere Suche zu garantieren, haben sich die Entwickler mit der Zeit angewöhnt, bei einem neuen Eintrag in die Datenbank den Call Stack in ein beschreibendes Feld zu kopieren. Die Suche nach einem korrespondierenden Eintrag basiert in den meisten Fällen inzwischen darauf, dass der Entwickler eine Funktion aus dem eben untersuchten Call Stack heraussucht und diese als Suchkriterium angibt. Da dieses Verfahren erst im Laufe der Verwendung des Bug-Tracking-Systems entwickelt wurde, gibt es noch einige ältere Einträge, deren Call Stack nicht suchbar ist. Davor mussten die Entwickler sich mit der Suche nach Schlüsselwörtern begnügen, die nach subjektivem Empfinden mehr oder weniger passend zur Beschreibung des Fehlers waren.
Alle anderen Entwickler haben auch die Möglichkeit Dump Files hochzuladen, wenn ihnen

im Rahmen ihrer Entwicklungen ein Absturz unterkommt, der nicht direkt Ursache ihrer Veränderungen war. Ihre Eintragssuche basiert auf demselben Verfahren, wie bei ihren beiden Kollegen, die mit der Fehlerberichtsverwaltung betraut sind.
Jedes Mal, wenn der Fehler vermeintlich erneut gemeldet wird, wird dem entsprechenden Fehlereintrag in Mantis die neu eingegangene Dump File angehängt. Im Wesentlichen bedeutet dies: Je mehr Dateien an einem Fehlereintrag in Mantis anhängen, desto höher sollte seine Behebung priorisiert werden.

Im Fall des Forschungsauftraggebers ist das Anlegen und Verwalten dieser Datenbank vor Entstehen dieser Diplomarbeit alleine durch den Menschen geschehen und dies hat einige Nachteile:
Hoher Aufwand an Zeit und Arbeit, die Subjektivität des einzelnen Entwicklers bei der Zuordnung des Fehlers zu einem bestimmten Fehlerbericht, das Wissen der Entwickler um den Inhalt und der Existenz jedes Fehlereintrags und die Verfälschung der Priorität zur Fehlerbehebung auf Grund von Fehlzuweisungen.

Betrachtet man die benötigte Zeit und Arbeit, die für die Zuständigen entsteht, ist festzustellen, dass der Aufwand für die Analyse der Fehler zuzüglich ihrer Behebung zu viel Arbeitszeit von den Entwicklern beansprucht.
Weiter fällt an, dass bei mehreren zuständigen Personen ihre Subjektivität bei der Zuordnung des Fehlers zu einem bestimmten Fehlereintrag unterschiedlich ausfällt. Während der eine Zuständige einen neuen Eintrag anlegt, würde sein Kollege den Fehler noch zu einem bestehenden hinzufügen, da ihm die Fehler ähnlich genug erscheinen.
Zusätzlich gibt es das Problem des Wissens um den Inhalt und der Existenz jedes Fehlereintrags. Selbst mit Hilfe von Bug-Tracking-Systemen oder anderen Datenbank-Verwaltungsprogrammen kommt es vor, dass der Nutzer einen gesuchten Fehlereintrag nicht findet, weil er nicht um ihn weiß oder eben etwas anderes darunter versteht als seine Kollegen. Daraus folgend legt er einen neuen Eintrag an oder ordnet den Fehler möglicherweise einfach nur einem unpassenderen Eintrag zu.
Wird ein neuer Eintrag angelegt, wozu eigentlich schon ein Eintrag im Bug-Tracking.-System besteht, so kommt es zu einer Verfälschung der Prioritäten, wenn man diese von der Anzahl der angehängten Dump Files pro Fehlereintrag abhängig macht.
Eine Fehlzuordnung sorgt weiterhin dafür, dass der zuständige Entwickler für die Behebung des Fehlers, der die Fehlzuordnung erhalten hat, verwirrt und in die Irre geführt wird. Entschließt er sich dann, die nicht wirklich zugehörigen Dump Files zu ignorieren, findet ein Fehler keine Behandlung. Behebt er ihn trotzdem, verfälscht sich die Statistik bezüglich der Anzahl an behobenen Fehlern, da er zwei Fehler behoben hat, aber nur einen Fehlereintrag als behoben einstufen kann.

Um diese Nachteile so weit wie möglich zu minimieren, befasst sich die Wissenschaft schon lange mit der Problematik der automatisierten Verarbeitung in der Fehlerberichterstattung.

Nach Brodie et al. [MB05] sind meist die Hälfte aller Fehlermeldungen, manchmal auch bis zu 90 Prozent wiederkehrende Auftritte von bereits bekannten Problemen. Im schlechtesten Fall sind es sogar wieder auftretende Fehler, die als behoben galten. Brodie et al. beschreibt weiterhin, dass in etwa ein Drittel des Arbeitspensums des Service Teams für die Suche

und mögliche Zuordnung an Hand von Fehlersymptomen zu bestehenden Fehlern benötigt wird.

Gerade das Vergleichen von Fehler erzeugenden Vorgängen ist etwas, das sich durchaus für Automatisierung eignet. Die Dump Files direkt auf binärer Ebene zu vergleichen, macht allerdings keinen Sinn, da dieser Vergleich nur so lange ein korrektes Ähnlichkeitsmaß erzeugen würde, wie man nur Dateien derselben Softwarerevision vergleicht. Unterscheidet sich die Revision, hat sich der Code der zugrunde liegenden Software verändert. Dadurch würde ein binärer Vergleich Abweichungen aufzeigen, wo an sich dieselbe Funktion aufgerufen wird, sich jedoch bedingt durch Änderungen die Zeilennummer unterscheidet. Es muss also ein anderes Format aus den Dump Files abgeleitet werden, das über Revisionsänderungen hinweg vergleichbar ist.

Call Stack als String In diesem Ansatz wird ein String basiertes Verfahren verfolgt, wie es von Brodie et al. [MB05] intendiert ist und auch bei Lerch et al. [JL13] und Bartz et al. [YDN12] Verwendung findet.

Eine String-Darstellung ist die Umwandlung von binären Daten in eine mehr oder minder für den Menschen lesbare Darstellung. Auf jeden Fall aber garantiert sie, dass der Mensch Muster erkennen kann. Der Mensch könnte - ohne Rücksicht auf den Zeitaufwand - auch zwei beliebige Dateien vergleichen, indem er eine String-Darstellung beider erstellen lässt und dann die Position und Art der einzelnen aufgeführten Symbole in beiden Dateien vergleicht. Da Dump Files im Debugging-Vorgang eine vom Menschen lesbare Darstellung ergeben, bietet sich die Verwendung dieser Darstellung für einen Vergleich an.

Basiert man die Dump File Analyse auf eine String-Darstellung, entspricht diese ohne weitere Formatierung der Ausgabe des Windows Debuggers in der Konsole, siehe Abbildung 2.1. Man hat pro Zeile einen Funktionsaufruf, seine mögliche Zeilennummer (sofern es eine .pdb zum Auflösen der Daten gab) in eckigen Klammern und schließlich den in hexadezimaler Darstellung aufgeführten Offset zum Funktionsanfang.

Abhängig davon wie und wo der kritische Abbruch im Prozessverlauf stattgefunden hat, enthält das angezeigte Dump File unterschiedlich viele Stacks und je Stack unterschiedlich viele für die Erstellung des Dump Files zuständige Funktionsaufrufe, die nicht zur allgemeinen Fehlerfeststellung beitragen. Durch ihre unterschiedliche Anzahl und Verteilung auf die Threads würde ein Vergleich zweier Dateien mit ihrer Berücksichtigung ein verfälschtes Ergebnis ergeben. Daher ist es nach Brodie [MB05] auch empfohlen diese Frames aus dem Call Stack für den Vergleich zu ignorieren.
Die Abbildung 2.2 zeigt ein Beispiel für einen solchen Call Stack mit samt der Einteilungen der Frametypen bezüglich ihrer Relevanz.

Nach den Dump File Erstellungsfunktionen und den auf die Erstellung der Fehlermeldung bezogenen Funktionen folgt die Funktion, in welcher der Fehler kritische Ausmaße angenommen hat. Dies ist aber nicht immer die Fehler verursachende Funktion. Diese kann weiter unten im Call Stack liegen.
Mit zunehmender Entfernung vom obersten Frame nimmt allerdings die Relevanz zum

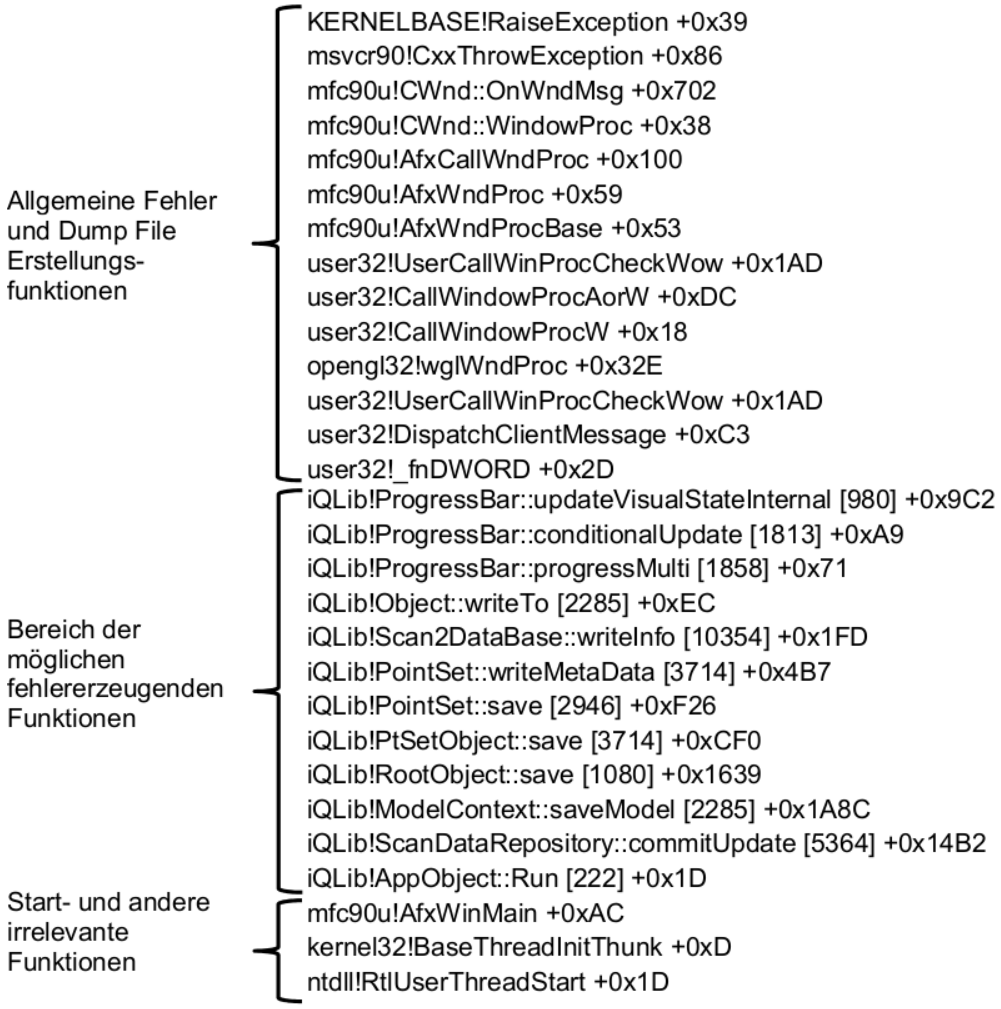


Abbildung 2.2: Beispiel eines Call Stacks

Fehlerbeitrag statistisch immer weiter ab. Krohn-Hansen [KH12] stellt in seiner Thesis sogar heraus, dass die meisten Abstürze und Fehlererscheinungen ihre Ursache in den obersten 10 relevanten Frames haben.

Je weiter man nach unten im Call Stack vordringt, desto häufiger stößt man auf Einstiegsroutinen und Startfunktionen, die seltenst Relevanz für die Fehler weit oben im Call Stack zeigen. Ist es allerdings nicht auszuschließen, dass solche Routinen oder Prozesse Fehler verursachen, werden sie mit betrachtet. Ansonsten kann man sie als irrelevant abtun.

Ein weiteres vergleichsverzerrendes Problem ist Code Rekursion. Da es selten genau vorgegebene unveränderliche Rekursionstiefen sind, muss Rekursion in den Vergleichsstrings behoben werden und auf eine einzige Aufführung für den Vergleich reduziert werden. Unterschiedliche Rekursionstiefen können auftreten, wenn man dieselbe Operation mehrfach durchlaufen lassen will und der Nutzer dabei die Anzahl bestimmen kann, oder es auch

den Operationsanforderungen nach eine bestimmte Anzahl an Durchläufen geben muss. Bei Grafikprogrammen beispielsweise kann es schnell zu einer unterschiedlichen Anzahl an Rekursionen kommen, wenn der Nutzer die Anzahl an Filterdurchläufen, um beispielsweise den Schärfegrad zu erhöhen, definiert, um ein bestimmtes Ergebnis zu erhalten. Hier würde die Funktion des Filteralgorithmus nun in bestimmter Anzahl nacheinander aufgerufen werden, um der Anweisung gerecht zu werden.

Nach der Bereinigung der Dump File Erstellungsfunktionen und der Rekursion können nun die Call Stacks verglichen werden, um ein Ähnlichkeitsmaß festzulegen und sie daran zu kategorisieren. Das Prinzip und der Algorithmus der längsten gemeinsamen Teilsequenz entnommen aus „Algorithmen - Eine Einführung“ von Cormen et al. [THC07] dient dabei als Basis zum Berechnungsverfahren für die Ähnlichkeit zweier Call Stacks. Dieser Algorithmus verfolgt den Ansatz zwei Dateien oder Strings zu vergleichen und dabei die gemeinsamen Sequenzen zu ermitteln und ihre Längen festzustellen. Sind alle Sequenzen gefunden, wird die längste dieser Sequenzen ermittelt.
Dieser Algorithmus gibt einem allerdings nur ein allgemeines Ähnlichkeitsmaß, welches keinerlei Rücksicht auf den Abstand der ähnlichen Frames zum obersten Frame nimmt, noch zum Versatz in diesem Abstand zueinander. Dies muss aber dringend berücksichtigt werden und wird daher eingehend im Kapitel zum Algorithmus beschrieben.

Von den verschiedenen Kategorisierungsverfahren, um Einträge zu erstellen oder zu ergänzen, war für diese Arbeit das ReBucket Verfahren nach Dang et al. [YDN12] am einflussreichsten.

Das ReBucket Verfahren basiert auf dem Systementwurf, dass man Fehlerberichte, die zum gleichen Fehler führen, zu einem Fehlereintrag zusammenfasst. Das zugewiesene Eintragen verfolgen zwar auch die meisten anderen der veröffentlichten Verfahren, doch anders als die Verfahren, die sich an der Ähnlichkeit der Auswirkungen des Fehlers oder den Fehlersymptomen orientieren, geht es auch beim ReBucket Verfahren um den String Vergleich von Call Stacks. Die Berechnung der Call Stack Ähnlichkeiten basieren Dang et al. hierbei auf einer modifizierten Variante des vorhin erwähnten „Längste gemeinsame Teilsequenz“-Algorithmus aus „Algorithmen - Eine Einführung“ von Cormen et al [THC07].
Ein Fehlereintrag in einem Bug-Tracking-System wird von Dang et al. als Bucket bezeichnet, egal ob es sich um einen Eintrag mit einer oder mehreren Dump Files handelt. Entsprechend der Anzahl an Dump Files in einem solchen Bucket, die die eingegangene Fehlerberichtsmenge wiedergibt, wird bei Dang et al. die Behandlungspriorität festgelegt.
Überschreitet die Ähnlichkeit der beiden Strings bei einem Vergleich einen vordefinierten Mindestwert, werden die Dump Files der verglichenen Call Stacks zusammen in einen Bucket geworfen.
Wird ein Call Stack mit einem bestehenden Bucket, der mehrere Dump Files beinhaltet, verglichen, entspricht der Ähnlichkeitswert für Dang et al. dem geringsten Ähnlichkeitswert, der beim Vergleich mit allen Dateien in dem Bucket erhalten wurde.
Das ReBucket Verfahren erzielt bei seinen Tests laut Veröffentlichung bezüglich der Übereinstimmung mit dem erhofften Ergebnis einen durchschnittlichen F-Wert von 0.876. Der F-Wert beschreibt hierbei, wie weit die Ergebnisse des Systems mit den tatsächlich zu erwartenden

Ergebnissen übereinstimmen. In diesem Fall also, wie weit die automatische Zuordnung Ergebnisse liefert, die mit einer absolut korrekten Zuordnung - die bei dieser Problemstellung nur mit erheblichem Aufwand erhalten werden kann - übereinstimmt.

Der Vorteil dieser Einträge bzw. Buckets, wie sie auch im weiteren Dokument bezeichnet werden, ist, dass aus ihnen nebst einer Prioritätenbildung auch noch ein besseres Verständnis für die Fehlerursache entstehen kann. Da es sich bei der ähnlichkeitsbasierten Zuweisung nicht um eine absolute Ähnlichkeit handeln soll, gibt es zulässige Unterschiede in den einzelnen Call Stacks zueinander.
Diese Unterschiede können solche Ausmaße annehmen, dass es zwar derselbe Fehler ist, aber der Weg diesen Fehler zu erreichen sich komplett unterscheidet. Auch dies kann trotzdem einen ausreichenden Ähnlichkeitswert erzeugen und somit dem Bucket beigefügt werden. Durch die nun unterschiedlichen Dump Files zu einem Fehler, die aber tatsächlich denselben Fehler erreichen, kann sich nun jeder Entwickler ein genaueres Bild vom Fehler machen. Ebenso verhindert diese mögliche Vielfalt an Fehlermeldungen eine Fehlerkorrektur auf einen einzigen Auslöser hin, da mehrere Aspekte bekannt sind.

Allerdings ist es auch negativ für die Entwicklung, wenn zwei verschiedene Fehler in einem Bucket landen. Zwei verschiedene Fehler führen, wie schon erwähnt, meist dazu, dass ein Fehler gar nicht behoben wird oder, wenn er behoben wird, es keinem bekannt wird, außer dem behebenden Entwickler. Sollte der Fehler je als Einzelfehler eingetragen und damit zur Behebung bekanntgegeben werden, wird unnötiger Zeitaufwand bei der Suche nach dem Fehler produziert, da die Fehlerursache bereits behoben wurde, ohne damals als solches bekannt zu werden. Krohn-Hansen hat in seiner Evaluation von Crash Analysen aufgezeigt, dass es daher sinnvoller ist, den Fehlerbericht in mehrere Buckets zu stecken oder sie zu verknüpfen, als mehrere Fehler konkret in einem Bucket zu haben.

Es folgen abschließend zu den Grundlagen ein paar visuelle Beispiele bezüglich der Zuordnung von Call Stacks zum selben Bucket. Diese Arbeit basiert zwar nicht auf der Graphentheorie bezüglich der Vergleiche in sich, doch ist es die einfachste Möglichkeit Ähnlichkeiten visuell zu veranschaulichen.

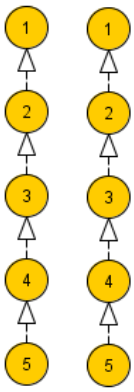
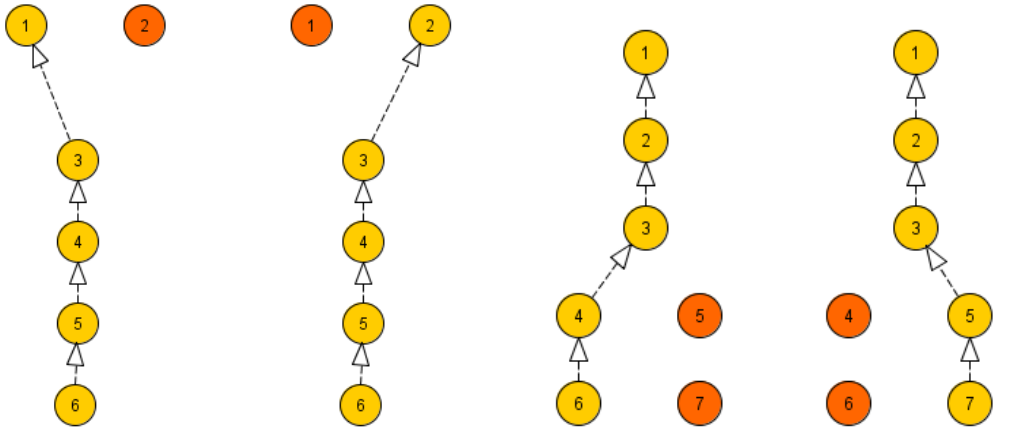


Abbildung 2.3: Call Stack Graphen absoluter Ähnlichkeit

Abbildung 2.3 zeigt eine Darstellung in Graphenform von zwei Call Stacks, die identisch sind.
Die in Abbildung 2.4a und 2.4b abgebildeten Graphen zeigen jeweils zwei Graphen die sich minimal unterscheiden. Per Definition kann es bei den verglichenen Call Stacks aber um Call Stacks desselben Fehlers handeln, da der Fehler in den Teilen liegen kann, die sich ähneln. Abbildung 2.4a zeigt hierbei Call Stacks, die sich in der zuletzt aufgerufenen Funktion unterscheiden. In Abbildung 2.4b liegt der Unterschied im Ursprung des Threads. Das Definieren dieser beiden Beispiele als selben Fehler begründet sich darin, dass der Fehler entweder aus einer Folge von Funktionen aufgerufen wird, die für mehrere Vorgänge gleich ist (wie im ersten Fall dargestellt), oder eine Funktionsfolge ist, die von mehreren unterschiedlichen Hergängen ausgelöst werden kann.
Es obliegt dem Entwickler zu klassifizieren, ab wann er statt einer Zuweisung zum selben Eintrag lieber das Anlegen zweier Einträge in Betracht zieht und diese verknüpft, um das Gesamtbild für die Fehlerbehebung trotzdem zu bieten.



(a) Call Stack Graphen unterschiedlicher Endfunktion

(b) Call Stack Graphen unterschiedlichen Ursprungs

Abbildung 2.4: Call Stack Graphen mit Ähnlichkeit

Es gibt natürlich auch den Fall des unterschiedlichen Fehlers, der aber einige gleiche Elemente zwischen den beiden Call Stacks im Vergleich findet. Auch hier obliegt es dem Entwickler zu spezifizieren, ab welchem Grad der Unterschiedlichkeit keine Zuordnung zum selben Eintrag mehr stattfinden kann. Abbildung 2.5 zeigt ein einfaches Beispiel für einen solchen Fall, in dem eine Ähnlichkeit vorhanden ist, aber sowohl der Weg zur Ähnlichkeit, als auch der Weg davon weiter im Thread dafür sprechen, dass es sich nicht um denselben Fehler handelt.
Auszuschließen ist es jedoch nicht immer, wie auch in diesem Beispiel sichtbar, dass der Fehlerverursacher in Knoten Nummer 3 liegt und es tatsächlich ein gemeinsamer Fehler wäre.
Dies sind eben die Momente, in denen ein automatisiertes System weniger Nutzen hat als der Mensch, der die Fehlerberichte auswertet.

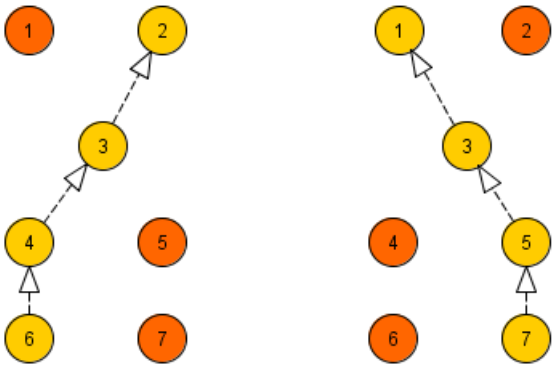


Abbildung 2.5: Call Stack Graphen unterschiedlicher Fehler

3 Aufbau des Systems

3.1 Das System auf einen Blick

Das für diese Diplomarbeit zu entwickelnde System soll in der Lage sein, per Nachrichtendienst eingehende Fehlermeldungen zu registrieren. Die Nachrichten sollen verarbeitet werden und zusammen mit ihren anhängenden Dump Files in ein Format gewandelt werden, das weiter verarbeitbar ist. Dieses Datenformat soll dann als Basis für den Vergleich zweier Call Stacks in String-Darstellung dienen. Anhand dieses Vergleiches wird ein Ähnlichkeitswert des Fehlerberichtes zu einem bestehenden Bucket oder die Neuerstellung eines Buckets errechnet. Bis hierhin gleicht das System im Entwurf seiner Funktionalität grundlegend all den anderen Systemen der aufgeführten Basiswerke [YDN12, MB05, JL13].

Nach der Ähnlichkeitsberechnung ändert sich die Funktionalität des Systems im Vergleich zum vollautomatischen Verfahren. Anstatt die Zuweisung des Fehlerberichts automatisch zu einem bestehenden Bucket vorzunehmen oder einen Neuen anzulegen, wird ein menschlicher Eingriff bezüglich der Zuordnung erwartet. Um eine Entscheidungsbasis für den Menschen zu bieten, werden ihm die größten Ähnlichkeitswerte bezüglich der einzelnen Buckets aufgelistet.

Der Entscheidung durch den Menschen folgend wird der Fehlerbericht dann wieder weitestgehend automatisch verarbeitet werden und zur Überprüfung und möglichen Ergänzung durch den Entwickler im gewählten Bug-Tracking-System aufgezeigt.
Es wird hier absichtlich vom gewählten Bug-Tracking-System gesprochen und nicht von Mantis, da das zu entwickelnde System bis auf eine Schnittstelle keinerlei Verbindung und Notwendigkeit zur Nutzung eines bestimmten Bug-Tracking-Systems haben soll.

Das Gesamtsystem teilt sich in drei Hauptmodule auf. Im Ersten wird sich um den Empfang und die Erstellung einer vergleichbaren Darstellung des Call Stacks gekümmert. Im Zweiten Modul geht es um den Vergleich - das algorithmische Kernstück. Und im dritten Modul geht es schließlich um die gesteuerte Entscheidungsfindung und Verarbeitung.

Eine Übersicht bietet Abbildung 3.1.

3.2 Fehlerbericht-Verarbeitung

Die Fehlerberichte erreichen die zuständige Verarbeitungsstelle via E-Mail. Diese E-Mails werden beim Nutzer bei einem kritischen Fehler automatisch erstellt und bieten dem Nutzer

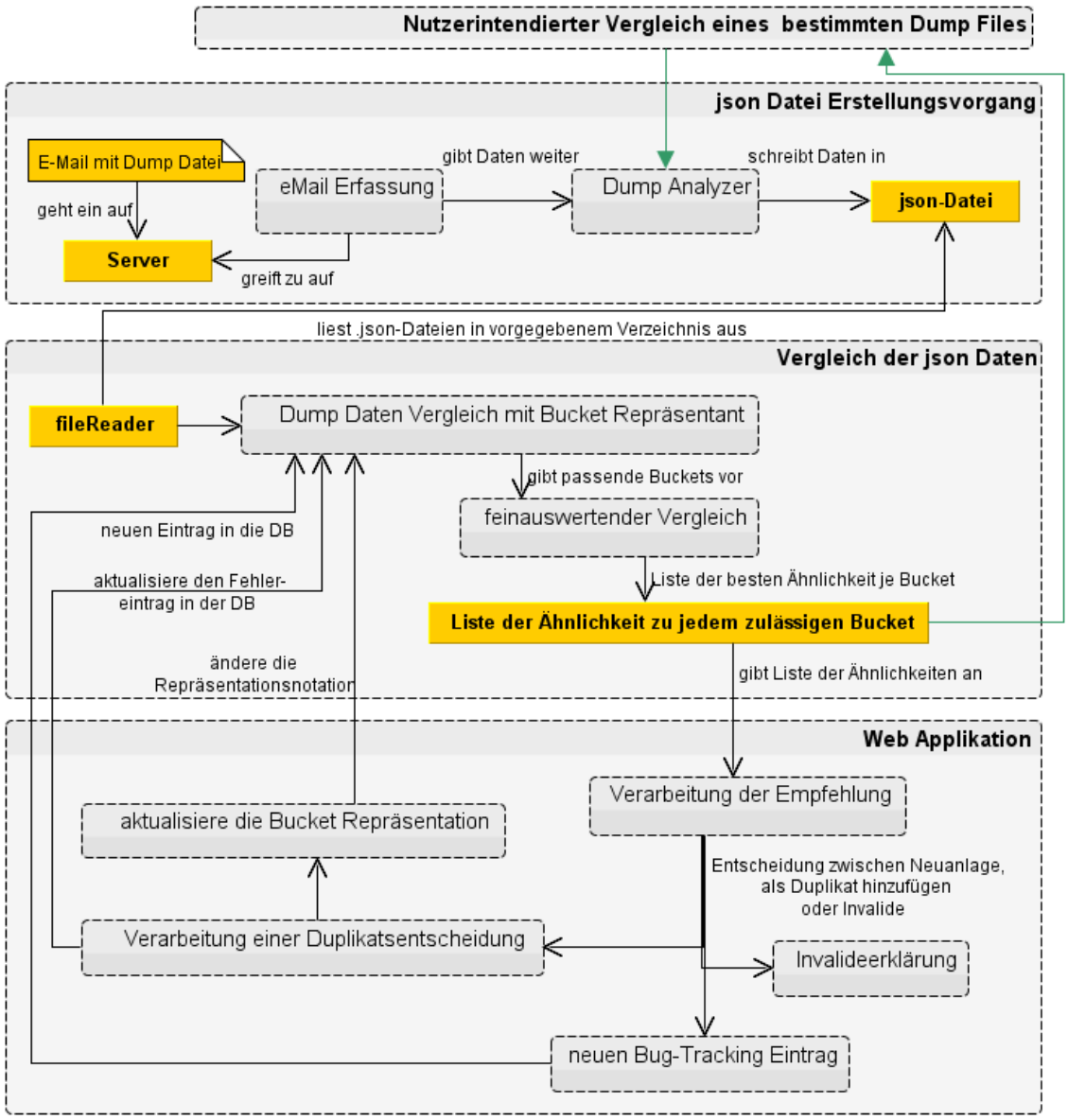


Abbildung 3.1: Grober Aufbau des Gesamtsystems aufgeteilt in seine Hauptfunktionalitäten nach dem Workflow

die Möglichkeit weiteres Feedback abzugeben. Die E-Mail wird mit einem Dump File als Anhang versehen und versendet.
Dieses Wissen nur als E-Mail zu lagern, würde bedeuten, dass jeder Entwickler Zugang zu dieser E-Mail benötigt. Daher werden letztlich alle enthaltenen Daten in das Bug-Tracking-System geschrieben. Somit muss bei der Verarbeitung der E-Mail nicht nur auf den Anhang Rücksicht genommen werden, sondern auch auf den Nachrichteninhalt. Hierzu benötigt man aber das geeignete Medium, um die Daten auszulesen und weiterzugeben.

Abbildung 3.2 zeigt wie die Module der E-Mail-Verarbeitung und des Dump-File-Auslesens zueinander und zu den ihnen voran gehenden Schritten, wie auch den nachfolgenden Modulen stehen.

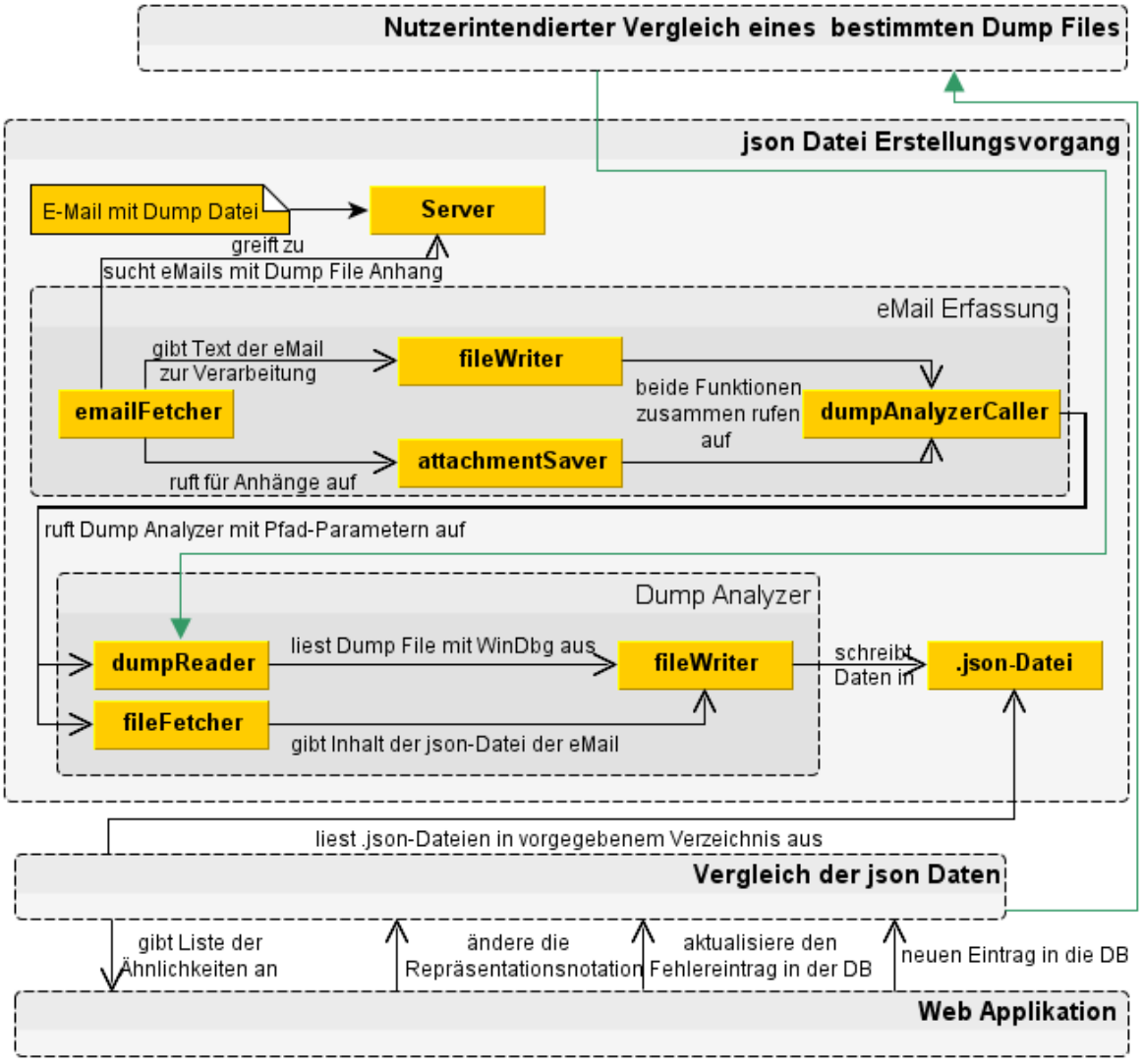


Abbildung 3.2: Ablauf und Zugriffsabhängigkeiten beim Workflow der E-Mail- und Dump-File-Verarbeitung

3.2.1 Verarbeitung der E-Mails

Die Wahl für ein Datenformat fiel auf das JSON Format [jso], da dieses Format Übersichtlichkeit über die enthaltenen Daten ermöglicht und die Möglichkeit besitzt, gezielt ausgelesen

zu werden. Außerdem ist es für den Menschen besser lesbar als ein XML Format beim selben Grad an Maschinenlesbarkeit.

Das Erfassungssystem für eingehende Fehlerberichte scannt in regelmäßigen Abständen die eingegangenen E-Mails. Wurde unter den eingegangenen E-Mails eine E-Mail nach dem Fehlerberichtformat gefunden, wird geprüft, ob diese E-Mail einen gültigen Anhang vom Typ des Dump Files hat. Ist dieser Typ angehängt, beginnt die automatische Verarbeitung der E-Mail.

Zuerst wird eine Basisdatei erstellt, deren Dateinamen sich aus dem Dateinamen des Anhangs, dem Absender und dem Eingangszeitpunkt der E-Mail zusammensetzt. In der Basisdatei werden vorerst der Eingangszeitpunkt der E-Mail, Softwaretyp, Versionsnummer und Releasetyp, als auch der Name des Senders und seine E-Mail-Adresse niedergeschrieben. Den Softwaretyp, die Versionsnummer und den Releasetyp erhält das System hierbei aus dem Dateinamen des angehängten Dump Files. Der Dateinamen setzt sich nach Firmenkonvention aus dem Typ der Software, die den Fehler erlitten hat, gefolgt von der Versionsnummer mit Revisionsnummer und dem Releasetyp, der zu erkennen gibt, ob es sich um die 64-bit Version oder die 32-bit Version handelt, zusammen. Weitere Daten, wie vom Nutzer mögliche angegebene Vorgangsbeschreibungen oder Kontaktdaten, werden aus dem E-Mail-Text herausgelesen, sofern der Nutzer diese beim Versand der Fehlermeldung angegeben hat.
Der Eintrag der Reportversion hat den Sinn für den Fall einer Weiterentwicklung des Aufbaus der .json-Datei eine Möglichkeit zu haben, zu erkennen, ob die zwei .json-Dateien überhaupt verglichen werden dürfen, oder ob eine der Dateien erst einmal auf einen gültigen Standard bezüglich der Darstellung ihres Inhaltes gebracht werden muss.

Ist die E-Mail ausgelesen und der Anhang gespeichert, wird die E-Mail zur Kenntnis der beiden zuständigen Entwickler als „In automatischer Bearbeitung“ gekennzeichnet. Hiernach wird ein Programm zur Verarbeitung des gespeicherten Dump Files mit dem Windows Debugger aufgerufen.

3.2.2 Verarbeitung der Dump Files

Um die Dump Files vergleichbar zu machen, muss die in Kapitel 2 bereits erwähnte String-Darstellung des Call Stacks erreicht werden. Hierbei bedient sich das System des Windows Debuggers und erzeugt damit die fehlenden Stack-Daten für die bereits bestehende .json-Datei, welche namentlich mit ihrem Dump File korrespondiert, nur von Typ .json ist und nicht .dmp.
Die Algorithmik, den Debugger zu automatisieren, wurde nach Beispielen von Josh Poley [Polo8b],[Polo8a] sowie an Hand von Internet-Beispielen von den Blogs „Yesterday is HISTORY, Tomorrow is MYSTERY“ [daio8] und „Win32Easy“ [Win11] entwickelt. Der Debugger ist zu einem eingebetteten Werkzeug geworden, dem Parameter übergeben werden und dessen Ausgaben bezüglich dieser Parameter weiter verarbeitet werden.

Da es mehrere Versionen und Revisionen von den für das System relevante Programmen gibt, wird aus dem standardisierten Dateinamen die Versions- und Revisionsnummer ausgelesen

und dementsprechend die für den Debugging-Vorgang benötigten Symbole und .pdb-Dateien geladen.

Am Ende ergibt sich ein Stack mit der Aufführung von Frames, bestehend aus den Feldern „module“, „class“, „function“, „line“ und „classpath“, wie in der Abbildung 3.3 im Ganzen zu sehen ist.
„module“ gibt das entsprechende Modul an, in dem der Frame aktiv ist. „class“ gibt die Klasse innerhalb des Moduls an, gefolgt vom Funktionsname im Feld „function“, die für den Frame aufgerufen ist.
Der Inhalt des Felds „line“ kann zweierlei Daten enthalten. Ist der Funktionsaufruf beim Debugging-Vorgang durch eine .pdb-Datei auflösbar, steht in diesem Feld die tatsächliche Zeilenzahl und der zusätzlich gelieferte Offset-Wert wird verworfen. Gibt es keine .pdb-Datei, die den Funktionsaufruf abbilden kann, erhält man nur den Offset-Wert.
Da sich der Offset innerhalb einer Revision für die Funktion nicht ändert, kann er wie die tatsächliche Zeilenzahl bezüglich des Vergleichs behandelt werden. Es darf der Offset-Wert zwar nie mit einem Zeilenwert verglichen werden, doch kommt dies nicht vor, da es nicht möglich ist, dass der Funktionsaufruf bei der einen Datei Zeilennummern erhält, während man bei der anderen nur den Offset erhält, da man entweder die .pdb-Dateien für die Module für jede Revision hat, oder für keine. Erst wenn man eigenständig im Code suchen würde, nützt einem die Offset-Angabe nichts mehr, da sie selten auch nur annähernd ihrem Wert nach eine Zeile in der Nähe der aufgerufenen Zeile Code abbildet. Doch gibt das System für die Programme, die durch die Dump Files analysiert und verbessert werden sollen, die genauen Zeilendaten an, da ihre Symbole mittels Referenzierung ihrer .pdb-Dateien komplett auflösbar sind. Die Verwendung von Offset-Werten betrifft also im Grund nur Funktionen Dritter, die im Prozessverlauf aufgerufen werden.

Um den Entwicklern die Arbeit noch weiter zu erleichtern, wird zusätzlich zu den bisher genannten Angaben auch noch der Klassenpfad bezüglich der eigens hergestellten Softwarefunktionen ausgelesen und im Feld „classpath“ gespeichert. So kann der Entwickler mit einem Blick den Dateipfad zur jeweiligen Source Datei, aus dem der momentane Frame zustande kommt, sehen. Dies beschleunigt bei der Fehlerbehebung das Suchen der Stelle ungemein und hat noch weitere Vorteile, auf die zu einem späteren Zeitpunkt eingegangen wird.

Die Verarbeitung eines Dump Files ist aus Nutzungsgründen auch ohne eine vorangehende E-Mail Verarbeitung möglich. Es wird einfach eine .json-Datei erstellt, die sich der Jetztzeit bedient und alle E-Mail bezogene Daten mit der Kennzeichnung „Autodump Analyser“ versieht. Somit können auch lokal von den Entwicklern Dateien für den Vergleich erstellt werden.

```
{
“ crashreport“:{
                “ reportversion“: “ 1.0“,
                “ version“: “ 5.2.100.34318“,
                “ software“: “ BASIC“,
                “ release“: “ x64“,
                “ header“:{
                        “ entrystamp“: “ 2013-09-17 14:03:14“,
                        “ sender“: “ Max Musterman“
                        “ email“: “ Max@Musterman.de“
                },
                “ data“: {
                        “ description“: “ Tried to change a tabbed view to a normal view.“,
                        “ company“:  “”,
                        “ address“:  “”
                },
“ stack“:       [
                {
                “ frame“: “ 0“,
                        “ module“: “”,
                        “ class“: “”,
                        “ function“: “ x%08I64X“,
                        “ line“: “ 0“,
                        “ classpath“: “”
                },{
                “ frame“: “ 1“,
                        “ module“: “ iQLib“,
                        “ class“: “ iQPrePaidCmd“,
                        “ function“: “ getOrgString“,
                        “ line“: “ 2069“,
                        “ classpath“: “ common/iqlib/iqlicense.cpp“
                },{
                ⋮
                },{
                “ frame“: “ 23“,
                        “ module“: “ ntdll“,
                        “ class“: “”,
                        “ function“: “ _RtlUserThreadStart“,
                        “ line“: “ 27“,
                        “ classpath“: “”
                }
                ]
                }
}
```


Abbildung 3.3: Beispiel einer automatisch erstellten .json-Datei

3.3 Dump Stack Vergleich

Bislang hat das System nichts Anderes getan, als die Grundlagen zu seiner Funktionalität gelegt. Nun jedoch kommt das Kernstück des Systems. Mit den bestehenden .json-Dateien, die anhand ihres Dateinamens jeweils eindeutig einem bestimmten Dump File zuzuordnen sind, kann der Vergleich vorgenommen werden. Abbildung 3.4 zeigt grob den Aufbau des für den Vergleich zuständigen Moduls.

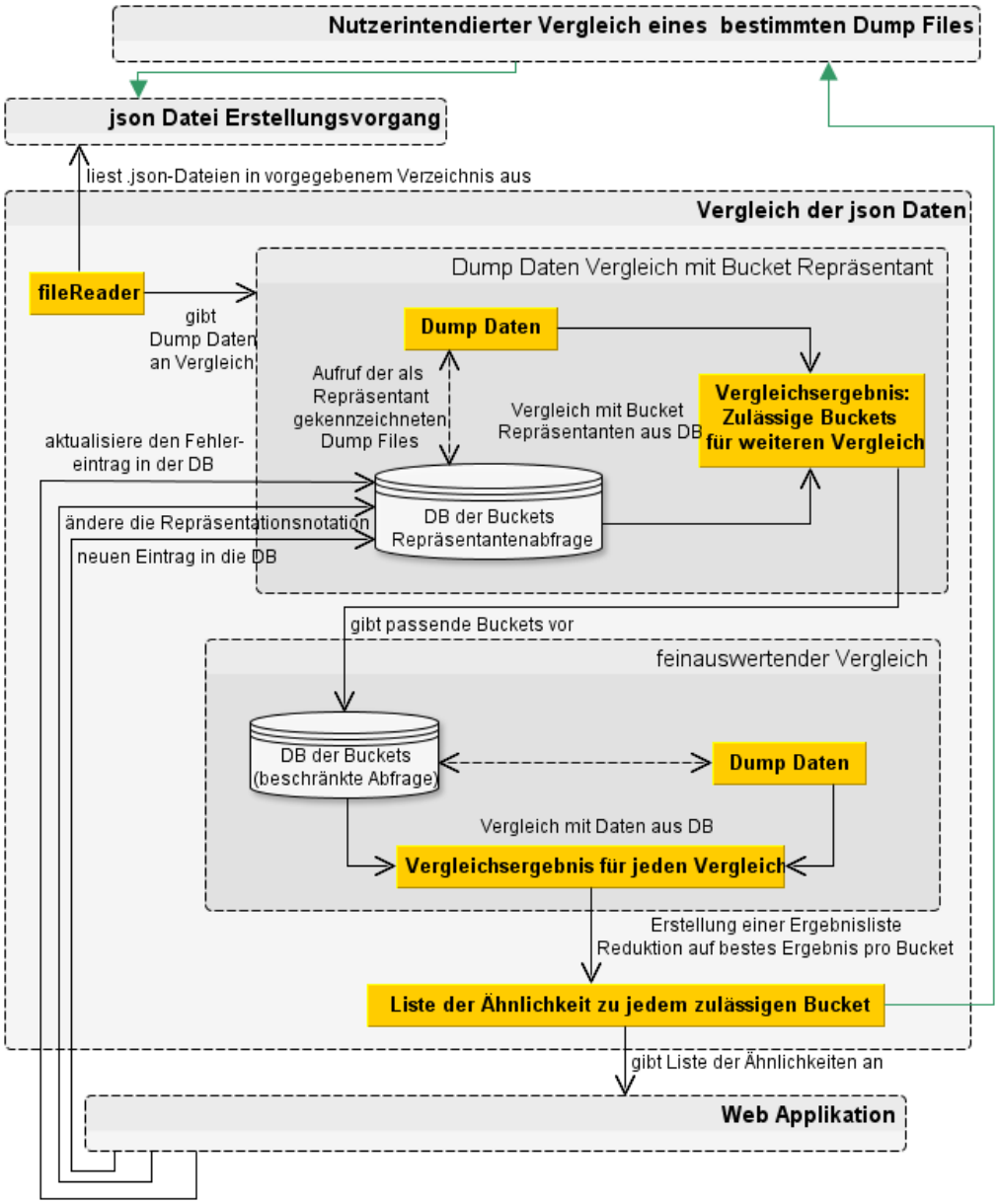


Abbildung 3.4: Ablauf des Vergleichsvorgangs

3.3.1 Der Grundalgorithmus zum Vergleichsberechnen

Berechnungsalgorithmus

Als Basis für die Ähnlichkeitsberechnung braucht es als Erstes eine Ermittlung der längsten gemeinsamen Sequenz. Eine Sequenz ist im Allgemeinen eine Aneinanderreihung von Gliedern. Im Fall dieser Diplomarbeit ist unter einem Glied ein Funktionsaufruf zusammen mit einer Zeilennummer oder einem Offset-Wert zu verstehen. Kurz eine Glied entspricht einem Frame.
Seien S_1 und S_2 die jeweiligen Call Stack Strings zweier Call Stacks. Ein String ist aufgebaut aus k Frames f: $S_{\mathrm{i}} = \{f_{\mathrm{i},1}, f_{\mathrm{i},2}, f_{\mathrm{i},3}, ..., f_{\mathrm{i,k}}\}$. Nun ist eine gemeinsame Teilsequenz somit $S_{\mathrm{gi}} = \{f_{1,\mathrm{j};\ 2,\mathrm{l}}, f_{1,\mathrm{j+1};\ 2,\mathrm{l+1}}, f_{1,\mathrm{j+2};\ 2,\mathrm{l+2}}, ..., f_{1,\mathrm{j+n};\ 2,\mathrm{l+n}}\}$. Um die längste gemeinsame Teilsequenz zu bestimmen, müssen nun die Längen aller gemeinsamen Teilsequenzen $S_{\mathrm{g}} = \{S_{\mathrm{g}1}, S_{\mathrm{g}2}, ..., S_{\mathrm{gh}}\}$ miteinander verglichen werden. Es ergeben sich also bis zu 2^{k} Teilsequenzen.

Da der Vergleich eine Vergleichsmatrix erzeugt, kann die Lösung auch rekursiv bestimmt werden. Dazu wird das Sequenzproblem in Teilprobleme unterteilt.

Wenn $f_{1,\mathrm{n}} = f_{2,\mathrm{m}}$ gilt, muss eine längste gemeinsame Sequenz von $S_{1,\mathrm{n\text{-}1}}$ und $S_{2,\mathrm{m\text{-}1}}$ gefunden werden. Da $f_{1,\mathrm{n}} = f_{2,\mathrm{m}}$ kann man das einfach an diese längste gemeinsame Sequenz anhängen und erhält die tatsächlich längste gemeinsame Sequenz.

Sind $f_{1,\mathrm{n}} \neq f_{2,\mathrm{m}}$ müssen zwei Teilprobleme gelöst werden: Zum Einen die längste gemeinsame Sequenz von $S_{1,\mathrm{n\text{-}1}}$ und $S_{2,\mathrm{m}}$ und zum Anderen von $S_{1,\mathrm{n}}$ und $S_{2,\mathrm{m\text{-}1}}$. Die längere Sequenz ist eine längste gemeinsame Sequenz.

Erkennbar hierin ist, dass die Teilprobleme eine Überlappung haben und rekursiv lösbar sind. Hierzu wird die Länge $g[i,j]$ als die Länge einer Sequenz der Strings S_{i} und S_{j} definiert. Entfällt $i = 0$ oder $j = 0$, so ist auch die Länge der Sequenz null. Diese daraus resultierende längste gemeinsame Sequenz ist folglich auch null.

$$g[i,j] = \begin{cases} 0 & \text{wenn } i = 0 \text{ oder } j = 0, \\ g[i-1, j-1] + 1 & \text{wenn } i,j > 0 \text{ und } f_{1,i} = f_{2,j}, \\ max(g[i, j-1], g[i-1, j]) & \text{wenn } i,j > 0 \text{ und } f_{1,i} \neq f_{2,j}. \end{cases} \tag{3.1}$$

Die Gleichung 3.1 legt eindeutig fest, welches Problem in Abhängigkeit vom Verhältnis von $f_{1,i}$ und $f_{2,j}$ zueinander zu betrachten ist. Gilt $f_{1,i} = f_{2,j}$ ist das Teilproblem in $S_{1,\mathrm{n\text{-}1}}$ und $S_{2,\mathrm{m\text{-}1}}$ zu betrachten. Ist aber $f_{1,i} \neq f_{2,j}$, sind die zwei Teilprobleme in $S_{1,\mathrm{n\text{-}1}}$ mit $S_{2,\mathrm{m}}$ und $S_{1,\mathrm{n}}$ mit $S_{2,\mathrm{m\text{-}1}}$ zu betrachten und der Maximalwert zu wählen. Der Aufwand liegt hier bei $\Theta(mn)$.

Die unter 3.1 aufgeführten Formeln lassen sich für den vorgesehenen Zweck auch noch reduzierter darstellen, da der Vergleichsfall, dass $i = 0$ oder $j = 0$ bei den gegebenen Daten nicht eintreten wird.

$$g[i,j] = max \begin{cases} g[i-1,j-1]+1 & \text{wenn } f_{1,i} = f_{2,j}, \\ g[i,j-1] & \text{wenn } f_{1,i} \neq f_{2,j}, \\ g[i-1,j] & \text{wenn } f_{1,i} \neq f_{2,j}. \end{cases} \tag{3.2}$$

Um die Länge der längsten gemeinsamen Sequenz zu berechnen, werden die Werte $g[i,j]$ in einer zeilenweise berechneten Matrix $G[0..n,0..m]$ gespeichert. Diese Matrix kann man nun verwenden, um entweder die längsten gemeinsamen Sequenzen visuell aufzuzeigen oder eben auch die längste Sequenz zu berechnen. Dies alles basiert noch alleine auf Kapitel „Längste gemeinsame Sequenz“ aus dem Buch „Algorithmen - Eine Einführung“ [THC07].

Nun interessiert aber nicht nur die Länge der gemeinsamen Sequenzen, sondern vor allem ihre Position im Call Stack, denn je entfernter sie vom oberen Ende des Call Stacks ist, desto irrelevanter wird die Sequenz. Sequenzen, die kurze Unterbrechungen erleiden, sind auch von Interesse, da diese Unterbrechungen nur zum Teil und meist abhängig von ihrer Position im Call Stack Fehlerrelevanz vorbringen können. Der oben geschilderte Algorithmus jedoch beachtet diesen Punkt nicht.

Um diese Punkte jedoch zu berücksichtigen, bezieht sich diese Arbeit auf die Veröffentlichung von Dang et al. [YDN12] und führt die Gewichtungsparameter c und o ein.
c ist der Abstandskoeffizient zum obersten Frame und o der Versatzkoeffizient zwischen den beiden verglichenen Call Stack Strings. Ihre Werte können manuell gesetzt werden oder durch lernbasierte Verfahren bestimmt werden.
Für die hier verwendeten Werte wurden 485 Dateien, deren allgemeiner Ähnlichkeitsgrad zueinander bereits bekannt war, miteinander verglichen und den Parametern ein Intervall von $[0..2]$ mit 0.1 Schrittweite zur Verfügung gestellt.
Es musste hierbei festgestellt werden, dass für c das Intervall auf $[0.7..2]$ beschränkt werden sollte, da sonst Unterschiede im oberen Bereich des Call Stacks durch ein hohes Ähnlichkeitsmaß weiter unten im Call Stack kaschiert werden konnten.

$$cost(i,j) = \begin{cases} e^{-c*min(i,j)}e^{-o*abs(i-j)} & \text{falls } f_{1,i} = f_{2,j}\ , \\ 0 & \text{sonst.} \end{cases} \tag{3.3}$$

Der Faktor *cost* resultiert aus dem Abstand vom obersten Frame des Call Stacks und der Verschiebung zwischen den beiden, dieselben Daten beinhaltenden, Frames in den

verglichenen Call Stacks, wie in Gleichung 3.3 gezeigt. Mit dem Faktor *cost* verändert sich nun die Gleichung 3.2 zum Folgenden hin:

$$G[i,j] = max \begin{cases} G[i-1,j-1] + cost & \text{wenn } f_{1,i} = f_{2,j}, \\ G[i,j-1] & \text{wenn } f_{1,i} \neq f_{2,j}, \\ G[i-1,j] & \text{wenn } f_{1,i} \neq f_{2,j}. \end{cases} \tag{3.4}$$

Diese Ähnlichkeitsmatrix G für die beiden verglichenen Call Stack Strings setzt sich aus den Vergleichen zwischen dem obersten bis zum i-ten Frame des Strings S_1 und dem obersten bis zum j-ten Frame des Strings S_2 zusammen. Der oberste Frame hat hierbei den Index 0. Nach Definition der Ähnlichkeitsmatrix lässt sich die Ähnlichkeitsberechnung für S_1 und S_2 nun auf einen einzigen Wert $G[m,n]$ reduzieren. Hierbei ist m die Länge von S_1 und n von S_2.

$$sim(S_1, S_2) = \frac{G_{m,n}}{\sum_{j=0}^{l} e^{-cj}} \tag{3.5}$$

In der Gleichung 3.5 ist l der Längenwert der kürzeren der beiden Strings S_1 und S_2 und bildet die Obergrenze der Summe im Divisor.

Mit Hilfe dieses Algorithmus kann die Ähnlichkeit zweier Call Stack Strings recht einfach ermittelt werden, um so die Basis für die Entscheidung zum Zusammenführen der beiden Dateien in einem Bucket zu setzen.

Call Stack Bereinigung

In Kapitel 3.2.2 wurde die Datenstruktur eines Frames aufgezeigt. Es soll nun aber nicht nur ein Frame mit einem anderen verglichen werden, sondern zwei ganze Stacks von Frames. Der oben aufgeführte Algorithmus ist in der Lage zwei Stacks zu vergleichen und ihre Ähnlichkeit zueinander zu berechnen. Das Problem hierbei ist, dass diese Stacks alles beinhalten, was das Dump File an Frame-Daten im Debugging-Vorgang ausgegeben hat. Diese Stacks enthalten durchaus noch Rekursionen und teilweise auch Dump File erstellende, wie auch Error Handling Funktionen.
Rekursionen haben den Nachteil, dass sie meist in nicht vorher festgelegter Anzahl vorkommen, und sich daher von Fehlerbericht zu Fehlerbericht in ihrer Anzahl unterscheiden, obwohl derselbe Fehler gemeldet wird. Würde man nun die Call Stacks dieser Fehlerberichte vergleichen, würde die Bestimmung der längsten Sequenz allein auf Grund der unterschiedlichen Rekursionstiefe zum Schluss kommen, dass die Strings über den gesamten Call Stack nicht identisch sind. Somit würde das System eine Zuordnung möglicherweise verweigern, obwohl es sich bei den beiden Berichten um denselben Fehler handelt.
Bezüglich der Funktionen, die zur Erstellung des Dump Files und des Error Handling

dienlich sind, ist es abhängig vom Thread, wie viele Funktionen diesbezüglich im Stack aufgeführt werden. Diese Willkürlichkeit führt dazu, dass es sich bei einem Sequenzvergleich ergeben kann, dass ungleiche Glieder und Verschiebungen in den Positionen im Stack bezüglich der gleichen Glieder existieren. Auch hier würde das System, ähnlich wie bei der unterschiedlichen Rekursionstiefe, zwei gleiche Fehler als nur teilweise übereinstimmend gewichten.

Um den korrekten Wert zu erhalten, müssen die Call Stacks vor dem Vergleich gefiltert werden. Der Filteralgorithmus läuft über den kompletten in der .json-Datei enthaltenen Call Stack und bereinigt den Stack um seine Rekursionen, seine Dump File erstellenden Funktionen und die Error Handling Funktionen.

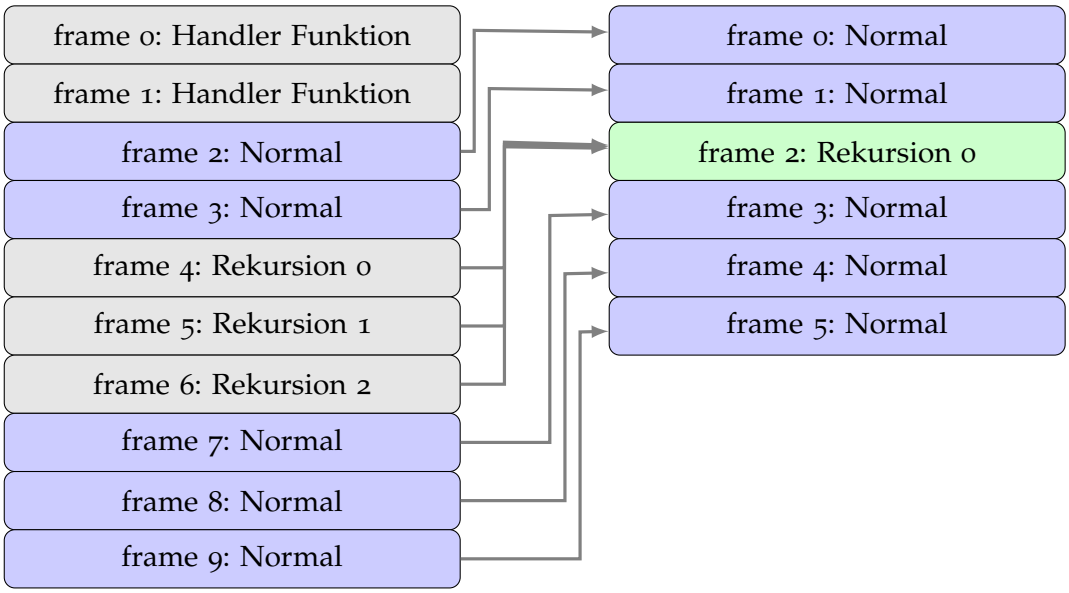


Abbildung 3.5: Links das mögliche Schema eines vollständigen Call Stacks. Rechts der bereinigte Call Stack für den Vergleich.

Abbildung 3.5 zeigt schematisch, wie der Call Stack nach der Filterung für den Vergleich bereit steht.

Die Filteralgorithmik beschäftigt sich zuerst mit der Entfernung der Dump File erstellenden und der Error Handling Funktionen, in der Abbildung als „Handler Funktion“ bezeichnet. Diese sind in ihrem Aufrufstring bekannt und werden bei Vorkommen aus der Vergleichsmenge entfernt, während alle Frames entsprechend des entfernten Frames in ihrem Index nachrücken.
Auf den so bereinigten Stack wird nun der Rekursionsfilter angesetzt. Rekursion wird daran erkannt, dass wiederholt derselbe Inhalt in nacheinander stehenden Frames vorkommt. Algorithmisch wird es umgesetzt, indem man beim obersten Frame 0 anfängt und dessen Inhalt mit dem direkt darauf folgenden Frame 1 vergleicht. Sind beide nicht identisch, besteht keine Rekursion. Nun wird Frame 1 mit Frame 2 verglichen. So schreitet man durch den gesamten Call Stack.

Würde man beispielsweise beim Vergleich von Frame 2 mit Frame 3 auf identische Frames stoßen, wird Frame 3 gelöscht und Frame 2 mit Frame 4 verglichen. Ergeben Frame 2 und Frame 4 keine Identität, rücken Frame 4 und alle darauf folgenden Frames auf, sodass der Index wieder fortlaufend ist. Ergibt sich jedoch eine Identität, wird auch Frame 4 gelöscht und der Vergleich mit Frame 5 fortgesetzt, und das so lange, bis keine Identität mehr mit Frame 2 gefunden wird. Dann rücken alle verbleibenden Frames nach Frame 2 im Index auf und die Rekursionssuche fährt mit dem neuen Frame 3 fort. Rekursionsvorkommen werden somit auf genau einen Frame reduziert.
Der so gefilterte Call Stack steht nun zum Vergleich zur Verfügung.

Sequenz Vergleichskriterien

Für den Vergleich sind folgende Daten wichtig:

- **Frame Nummer**: Sie gibt dem Algorithmus zur Ähnlichkeitsbestimmung die Position des Frames im Stack an.
- **module**: Der Modulname auf den sich der Frame bezieht.
- **class**: Der in der Datei enthaltene Klassenname, die angewandt wird
- **function**: Der Funktionsname in der Klasse selbst
- **line**: Die erreichte Zeilennummer in der Funktion zum Zeitpunkt des kritischen Abbruchs

Beim Vergleich wird überprüft, ob die Werte in den Feldern „module“, „class“, „function“ und „line“ übereinstimmen. Hierbei ist die Überprüfung hierarchisch geschachtelt, um unnötige Überprüfungen zu vermeiden.
Stimmt alles überein, werden die Frame Nummern zur Verrechnung freigegeben.

Da sich allerdings durch Fehlerbehebungen und neue Funktionen die Zeilennummern über Revisionen und Versionen hinweg ändern können, ist die Zeilennummer nur bei gleichen Versions- und Revisionsnummern als absolut vergleichbar anzusehen. Bei unterschiedlichen Versionen und Revisionen besteht die Möglichkeit die Zeilennummer komplett zu ignorieren oder aber einen Toleranzwert einzufügen.
Diese Arbeit basiert auf der Wahl eines Toleranzwertes.
Der Toleranzwert wurde ermittelt durch Vergleichen des Codezuwachses über die Versionen hinweg. Als Ergebnis kam ein medialer Wert von 10 Prozent heraus, der nun zur Berechnung der Abweichtoleranz bei der Zeilennummer bei Versions- und Revisionsunterschieden dient.

Die 10 Prozent werden vom Größeren der beiden Zeilenwerte berechnet und dann überprüft, ob der andere Zeilenwert innerhalb eines 10 prozentigen Intervalls zum ersten Wert liegt. Ist dies der Fall, sieht die Berechnung vor, diesen Frame als Ähnlich zu kategorisieren. Es ist durchaus möglich, dass es sich nicht um einen ähnlichen Frame handelt und so ein fehlerhafter Ähnlichkeitswert ermittelt wird. Doch ist der Fehler geringer, als die dadurch korrekt als Ähnlich erkannten Call Stacks, die ohne dieses Intervalls nicht so hoch als

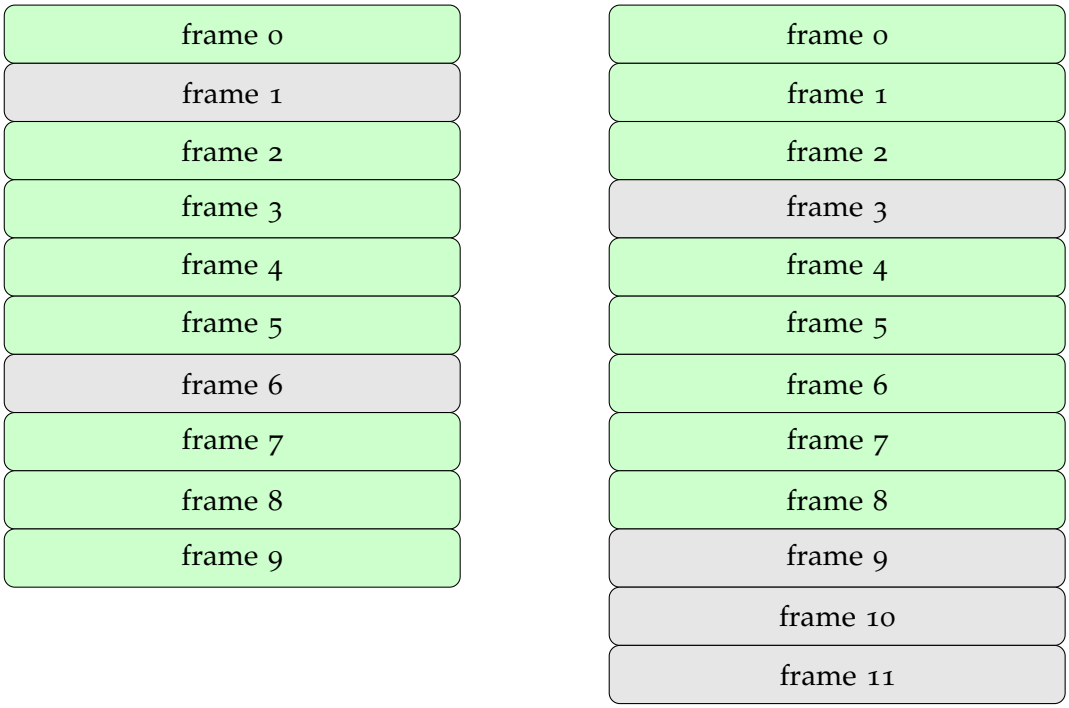


Abbildung 3.6: Beispiel für den Vergleich zweier Call Stacks. Grüne Frames sind identisch, graue Frames haben keinen korrespondierenden Frame

ähnlich kategorisiert worden wären. Diese höhere Kategorisierung entscheidet nachweislich in bestimmten Fällen über die Zuordnungsempfehlung und ändert sie von einer falschen Zuordnung in eine korrekte.

3.3.2 Bucket Vorbereitungen für die bestehende Datenbank

Da bereits eine Fehlerberichtsdatenbank existierte, mussten die existenten Buckets ermittelt werden. Hierzu ist anders als Dang et al. [YDN12] bei diesem System nicht jedes File als Repräsentant für seinen Bucket gedacht.
Dies begründet sich in einer Leistungssteigerung des Vergleichhergangs durch die Reduktion der Vergleichsmenge unter Berücksichtigung, dass für bereits bestehende Buckets zwischen ihren beinhalteten Dateien ein hoher Ähnlichkeitswert bestehen soll.

Das Mengengerüst der Test-Datenbank zeigt 228 verschiedene Buckets mit insgesamt 425 darin verarbeiteten Dump Files. Die Menge an Dateien, die durch das System im Rahmen dieser Diplomarbeit verarbeitet werden soll, beträgt 194. Würde der Ansatz von Dang et al. verfolgt werden, würde im schlimmsten Fall, bei einem Vergleichsdauer-Median von $0,0069$ Sekunden pro Datei, eine Laufzeit zwischen 3 Sekunden für die erste Datei und $4,32$ Sekunden für die letzte Datei der zuzuteilenden Dateien anfallen, da jede eingehende Datei mit allen Dateien im Bug-Tracking-System abgeglichen werden muss. Dies wäre in einem automatisierten System vorerst nicht von Belang, da dieser Verzug nicht weiter auffallen

würde, als dass die Behandlung eines einzelnen eingegangenen Fehlerberichts mit der Zeit länger dauern würde.

Unser System hat jedoch die menschliche Kontrollkomponente und ist so aufgebaut, dass sie die eingehenden Dateien für die Entscheidungen vorverarbeitet und der Entwickler den Zeitpunkt der Entscheidung bestimmt. Somit muss das System je nach Entscheidung des Entwicklers alle noch ausstehenden Empfehlungen aktualisieren. Da die menschliche Komponente aber selten geduldig ist, muss das System so nahe an die Echtzeit geführt werden wie irgendmöglich. Deshalb hat jeder Bucket einen Repräsentanten, der als erstes mit neuen Call Stacks verglichen wird.
Um diesen Repräsentanten zu bestimmen, wurde in jedem Bucket eine Ähnlichkeitsbestimmung vorgenommen und die Datei, welche den höchsten Übereinstimmungsgrad mit allen anderen Dateien erzielt, als Repräsentant gewählt.

Nun werden bei einem Vergleich zuerst alle Bucket-Repräsentanten zu Rate gezogen. Überschreitet das resultierende Ähnlichkeitsmaß für den Vergleich mit einem Bucket-Repräsentanten einen vorher definierten Grenzwert, werden die Dateien im Bucket mit dem Call Stack verglichen und ihr höchster Wert als Ähnlichkeitswert zu dem Bucket genommen.

Buckets, die durch die Entscheidung des Entwicklers neu angelegt werden, erhalten automatisch die Datei, über die der Bucket angelegt wird, als ihren Repräsentanten.

3.3.3 Änderungsmöglichkeit des Bucket Repräsentanten

Da der initiale Bucket-Repräsentant auch die Vergleichsgrundlage und erste Hürde für einen Vergleich mit den restlichen Dateien des Buckets bildet, ist es ratsam, diesen Repräsentanten so passend wie irgendmöglich zu halten. Durch neuere Revisionen könnte es sonst passieren, dass die geforderte Mindestähnlichkeit, je nach festgelegtem Wert, nicht mehr erreicht werden kann, da sich der ursprünglich gewählte Repräsentant zum Beispiel durch seine Zeilenzahlen einfach zu stark vom momentan gültigen Code für diesen Fehler abhebt.
Um dies zu verhindern ist mit jeder Datei in einem Bucket ein Zähler verbunden, der angibt, wie oft diese Datei für einen Vergleich, der auch anschließend als passend gewählt wurde, von höchster Relevanz war.

Der initiale Repräsentant erhält einen Grundwert von 1, die anderen Dateien einen Wert von 0. Bei einem neu angelegten Bucket wird die Datei, die mit dem Anlegen des Eintrags in das Bug-Tracking-System abgelegt wird, als initialer Repräsentant gewertet.
Beim Wählen einer gegebenen Empfehlung wird der Zähler bei der entsprechenden Datei erhöht und hiernach verglichen, ob der momentan festgelegte Repräsentant immer noch den höchsten Relevanzwert hat.
Gibt es eine andere Datei mit höherem Relevanzwert, werden die Daten dieser Datei im Bug-Tracking-System über die des bisherigen Repräsentanten geschrieben und stehen für neue Vergleiche sofort zur Verfügung. Dadurch soll ein Veralten des Bucket-Repräsentanten verhindert werden.

Das Ändern des Relevanzwertes findet jedoch nur statt, wenn eine vom System empfohlene Zuordnung stattfindet. Eine manuell vom Entwickler eigenständig vorgenommene Zuordnung kann nicht berücksichtigt werden, da hier nicht bekannt ist, welche Datei die nötige Relevanz gezeigt hätte und daher keine Datei korrekterweise eine Wertaufstockung erhalten kann.

3.4 Das Webinterface - Der Ort der Entscheidung

Der Workflow des Webinterfaces lässt sich durch Abbildung 3.7 verdeutlichen. Anders als bisher hat hier der Nutzer Entscheidungskraft.

Die Web-Applikation wird von den Ergebnissen des Vergleichs mit Daten versorgt, hierbei schreibt das Vergleichsmodul für jeden verglichenen Fehlerbericht die erhaltenen sechs besten Einträge in eine Datenbank. Es kann auch vorkommen, dass der Vergleich weniger als sechs Ähnlichkeitswerte ermitteln kann, abhängig davon, wie viele Buckets eine Datei beinhalten, die beim Vergleich einen höheren Wert als den vorgegebenen Grenzwert erzielt. Auch ist es möglich, dass es mehr als sechs Werte gibt, die einen höheren Wert als den vorgegebenen Grenzwert haben. Im Mittel waren es aber auf die Testdaten bezogen sechs ähnliche Buckets, die von Relevanz waren.

Die Einträge dieser Datenbank warten nun auf die menschliche Kontrolle und Weiterverarbeitung. Hier ist die menschliche Kontrollinstanz des halb-automatischen Systems angebracht und hier entscheidet der zuständige Entwickler, ob der Fehlerbericht als „Neu“ oder als ein Duplikat eines bereits vorhandenen Fehlereintrags im Bug-Tracking-System klassifiziert wird. In manchen Fällen kommt es vor, dass der Entwickler den Fehlerbericht als „Invalide“ klassifizieren wird, weil das Dump File sich als korrupt erweist, oder der Absturz keine durch die eigenen Entwickler behebbaren Gründe aufzeigt.

Wird der Eintrag als „Neu“ klassifiziert, so wird ein neuer Eintrag in den Datenbanken des Bug-Tracking-Systems angelegt und zur Kontrolle und Ergänzung dem Entwickler präsentiert.
Wird der Fehlerbericht als Duplikat erkannt, so wird das System den Fehlerbericht an den schon vorhandenen Eintrag im Bug-Tracking-System anhängen. Beim Anhängen werden nebst einfachen Zuordnungen und Aktualisierungen der Daten auch Prüfungen vorgenommen. So wird der Status des Eintrags geprüft, ob er beispielsweise als geschlossen oder repariert definiert ist. Ist einer dieser beiden Status der Fall, wird die Revisionsnummer des neu angefügten Berichts mit der hinterlegten Revisionsnummer zum Schließen des Eintrags verglichen. Sind die Revisonsnummern unterschiedlich, gibt es zwei Möglichkeiten:
Ist die Revisionsnummer des Eintrags neuer als die des Berichts, so ändert sich am Status nichts. Ist aber die Revisionsnummer des Eintrags älter als die des Berichts, so wird der Eintrag wieder geöffnet und der Entwickler auf diese Tatsache hingewiesen, dass er noch Änderungen und Zuweisungen bezüglich des Eintrags vornehmen sollte, bevor er ihn abschließt.

Sowohl bei der Wahl einen neuen Eintrag anzulegen, als auch ein Duplikat zuzuweisen, erstellt das System im Hintergrund automatisch Einträge für den Entwickler im System, um die Arbeitslast zu mindern. Im Fall einer Duplikatszuweisung überprüft das System abschließend den Bucketrepräsentanten auf seine Repräsentativität hin und ersetzt diesen gegebenen Falls durch einen Neuen.

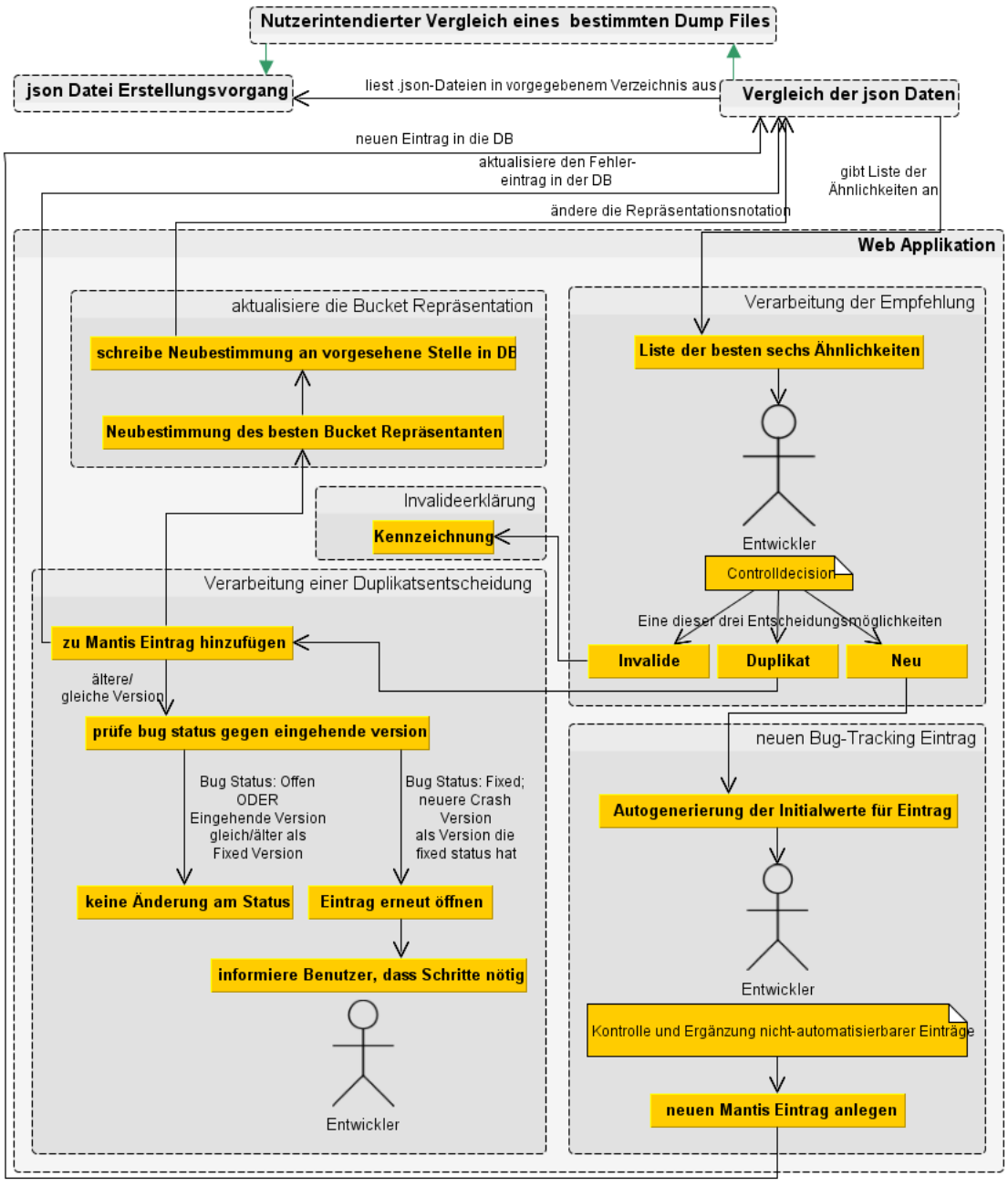


Abbildung 3.7: Workflow des Webinterface

Sollte der Entwickler einen Fehlerbericht als invalide abtun, so wird dieser einer Kennzeichnung unterzogen, welche vollautomatisch und ohne nachträglichen Zutuns des Entwicklers abgeschlossen ist.

Um diese Funktionalität sinnvoll zu unterstützen, ergeben sich die Ansichten des Webinterfaces wie folgt:
Die Startseite des Webinterfaces zeigt eine Auflistung aller noch nicht vom Entwickler verarbeiteten eingegangenen Fehlerberichte, wie in Abbildung 3.8 auf Seite 45 zu sehen ist. Die Liste gibt den Dateinamen an, wie er nach den festgelegten Konventionen aufgebaut wurde. Zusätzlich listet sie den Sender, seine E-Mail-Adresse und das Eingangsdatum auf. Vor allem das Eingangsdatum ist wichtig, damit Fehlerberichte nicht zu lange auf ihre Verarbeitung warten und ist daher auch das Hauptkriterium bei der initialen Sortierung der Liste. Die letzten Felder zeigen dann noch den Softwaretyp, seine Versionsnummer und die Revision. Da sich das Bug-Tracking-System durch internes Zutun der Entwickler verändern kann, wird auch noch der letzte Zeitpunkt, an dem die Datei mit dem Bug-Tracking-System verglichen wurde, angegeben. Dies hat den Sinn, dass der Entwickler auf Basis dieses Wissens den Entschluss fassen kann, das System aufzufordern, alle Dateien nochmals zu vergleichen. In diesem Feld wird auch angezeigt, ob sich die Datei zur Zeit in einem aktualisierenden Vergleichszyklus befindet, der nach Verarbeitung eines anderen Listeneintrags oder nach vorgesehenen Zeitintervallen automatisch ausgelöst wird.
Das Feld „In Use“ gibt an, ob der betreffende Eintrag zur Zeit von einem anderen Entwickler oder auch nur in einem anderen Fenster gerade offen ist, um zu verhindern, dass zeitgleich eine Mehrfachbearbeitung stattfinden kann. Entsprechend des Status dieses Feldes ist der Aufruf des Listeneintrags gesperrt.

An Hand der im Vergleich erstellten Ähnlichkeitswerte zu den einzelnen Buckets wird eine Relevanztabelle zu jedem eingehenden Fehlerbericht erstellt und dem Nutzer des Systems werden maximal die sechs besten dazu passenden Buckets in Form des für jeden Bucket ähnlichsten Call Stacks präsentiert. Vgl. dazu Abbildung 3.9 auf Seite 46.

Ab hier hat der Entwickler die Möglichkeit den Fehlerbericht einem der aufgeführten Vergleichsberichte und dadurch einem vorgegebenen Bucket anzufügen, oder aber den Fehlerbericht als neu zu definieren. Alternativ kann er einen Bucket angeben, der nicht aufgeführt ist und den Fehlerbericht so einem von ihm manuell bestimmten Bucket anhängen. Für den Fall, dass der Call Stack des Fehlerberichtes allerdings keinen von der Firmensoftware verursachten Fehler beinhaltet oder korrumpiert ist, besteht auch noch die Möglichkeit den Fehlerbericht und im gleichen Zug automatisch auch das Dump File als invalide zu klassifizieren.
All diese Optionen sind unterhalb der aufgeführten Call Stacks auswählbar, fehlen in der Abbildung jedoch, um das Wesentliche, nämlich die Aufführung der Call Stacks und der Vergleichswerte in ihrem Eindruck nicht zu mindern. Sie sind ebenfalls in der den Source Code zeigenden Detailansicht vorhanden.

Um eine höhere Genauigkeit und Vereinfachung des Aufwandes zu ermöglichen und damit der Entwickler bei Unsicherheit nicht jedes Mal die Entwicklungsumgebung zum Abgleichen der Frames mit dem Code aufrufen muss, kann der Entwickler weitere Details bezüglich der Source Code Stellen zu den verschiedenen Call Stacks im Bezug auf den neuen Call Stack

anfordern.
Hier kommt das JSON Feld „classpath“ zum Tragen, da anhand dessen die entsprechende Source Code Datei zum Frame gefunden werden kann. Diese Datei wird ausgelesen und anhand der Zeilennummer des Frames eine Umgebung des Source Codes um die Zeilenstelle herum angezeigt. Es erleichtert das Fehlerverständnis ohne den Aufwand eine Entwicklungsumgebung zu starten oder mühsam die Source Code Datei selber zu suchen.
Abbildung 3.10 auf Seite 47 zeigt ein Beispiel für die Darstellung.

Hat der Entwickler nun seine Wahl getroffen und den Fehlerbericht als „Neu“ oder „Invalide“ klassifiziert oder einem bestehenden gelisteten oder ungelisteten Bucket zugeordnet, kommt es zusätzlich zum Eintrag des Fehlerberichts in das Bug-Tracking-System zu den abschließenden Schritten.
Zuerst wird abhängig von der Wahl des Entwicklers die zum Fehlerbericht zugehörige E-Mail gesucht und ihre Kategorisierung von „In automatischer Bearbeitung“ auf entweder „Automatische Bearbeitung abgeschlossen“ oder „Invalider Dump - Automatisch Bearbeitet“ geändert. Ab diesem Zeitpunkt ist es für die Entwickler beim Einsehen der E-Mails ersichtlich, dass die E-Mail behandelt wurde. Wird keine E-Mail gefunden, kann keine Kategorisierung vorgenommen werden, es wird dennoch mit der Bearbeitung fortgefahren.
Direkt im Anschluss an die Kategorisierung der E-Mail wird der Entwickler auf den Eintrag im Bug-Tracking-System weitergeleitet. Hier kann er letzte Einstellungen vornehmen, wie den Eintrag einem Entwickler zur Behebung zuweisen und ergänzende Daten eintragen bzw. ändern. Bei Einträgen, denen durch die Entscheidung des Entwicklers neue Daten hinzugefügt wurden, erhält der Entwickler visuelle Hinweise, falls sich nebst der Datenmenge zusätzliches am Eintrag geändert hat, wie zum Beispiel, dass der Eintrag wieder geöffnet wurde, da die Revisionsnummer des Fehlerberichts neuer ist, als die des Eintrags, als dieser als behoben bzw. geschlossen klassifiziert wurde.
Währenddessen erfolgt auf dem Server parallel die Aktualisierung aller noch zuzuweisenden Berichte bezüglich des neuen oder aktualisierten Eintrags im Bug-Tracking-System. Ergibt sich ein neuer Wert für einen bestehenden Ähnlichkeitseintrag, werden die Einträge neu sortiert. Sollte sich sogar ein neuer Ähnlichkeitswert ergeben, wird er entsprechend seines Wertes in die bestehende Liste eingegliedert. In der Regel ist dieser Vorgang schneller fertig, als der Entwickler Zeit braucht, um den Eintrag im Bug-Tracking-System zu vervollständigen und zuzuweisen. Ist dies einmal nicht der Fall, so wird in der Auflistung der noch zuzuordnenden Fehlerberichte in der Spalte des letzten Vergleichszeitpunktes die Kennzeichnung „Updating“ erscheinen, um dem Entwickler zu verdeutlichen, dass bei diesem Eintrag sich zur Zeit möglicherweise Ähnlichkeitswerte verändern können. Bei den fertigen Einträgen steht die letzte Vergleichszeit.
Ist der Entwickler mit seinen ergänzenden Tätigkeiten bezüglich des momentan präsentierten Eintrags ins Bug-Tracking-System fertig, so kann er wieder zur Liste der zur Kontrolle und Zuweisung offenstehenden Berichte zurückkehren und durch Anwahl eines Listeneintrages den Vorgang von vorne beginnen.

Issue of unhandled issues

Recompare the List with Mantis Database | Take local file for compare

Name	Sender	Sender eMail	Received on	Software	Version	Revision	Latest Compared	In Use
BASIC_5.1.6.32766_x64_Autodump_Analyzern_20131018-181936.json	Autodump Analyzer		2013-10-18 18:19:36	BASIC	5.1.6	32766	2013-10-31 11:53:26	
BASIC_5.1.6.32766_x64_Autodump_Analyzer_20131018-181957.json	Autodump Analyzer		2013-10-18 18:19:57	BASIC	5.1.6	32766	2013-10-31 11:53:44	
BASIC_5.1.3.32608_x64_Autodump_Analyzer_20131018-18208.json	Autodump Analyzer		2013-10-18 18:20:08	BASIC	5.1.3	32608	2013-10-31 11:52:41	
CLOUD_5.1.6.32766_x64_Autodump_Analyzer_20131018-182032.json	Autodump Analyzer		2013-10-18 18:20:32	CLOUD	5.1.6	32766	2013-10-31 11:53:35	
APP_5.1.6.32766_x64_Autodump_Analyzer_20131018-182132.json	Autodump Analyzer		2013-10-18 18:21:32	APP	5.1.6	32766	2013-10-31 11:52:59	

Abbildung 3.8: Hauptansicht des Webtool zur Fehlerberichtsverwaltung

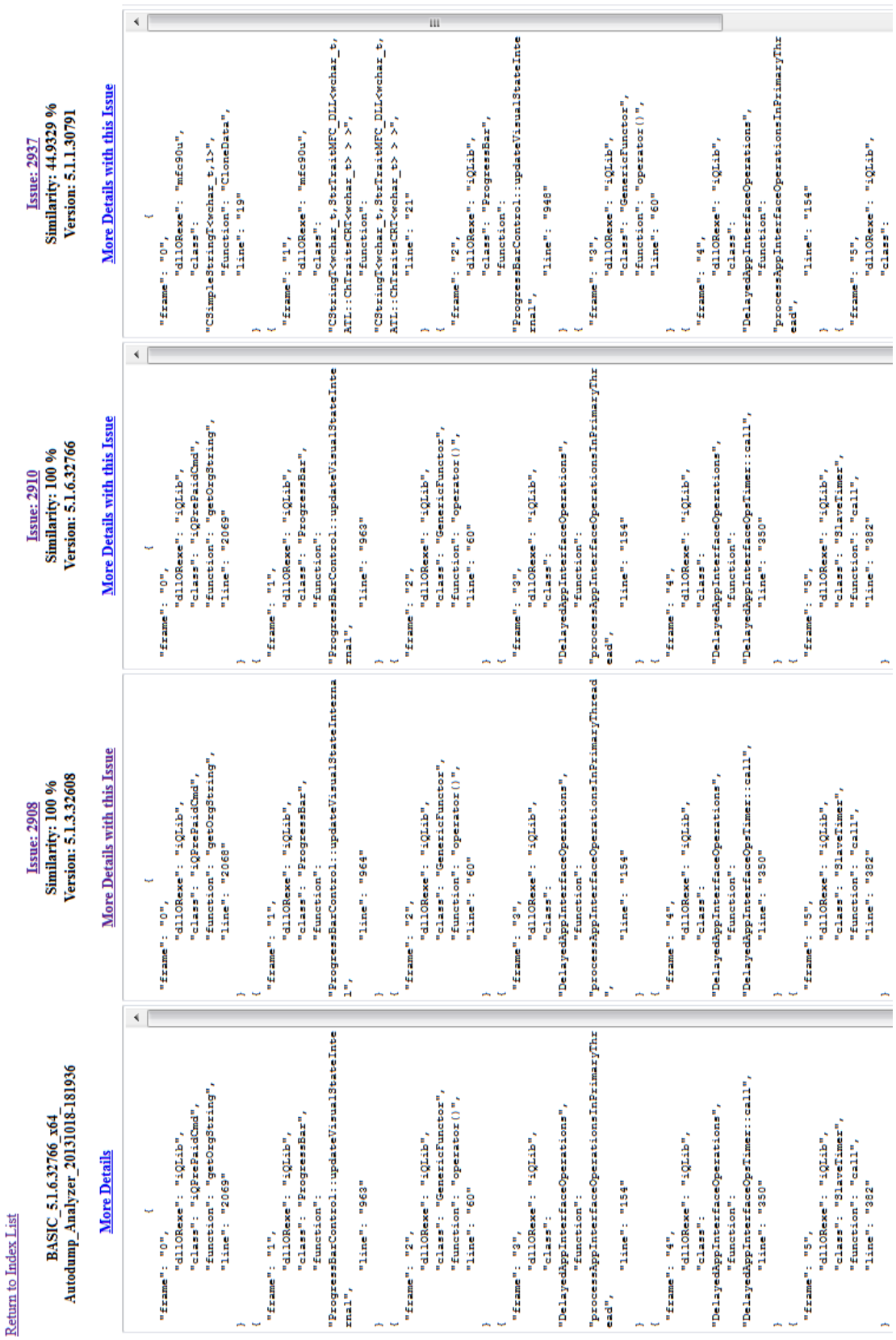


Abbildung 3.9: Allgemeine Detailansicht des Webtool zu einem gewählten Fehlerbericht
46

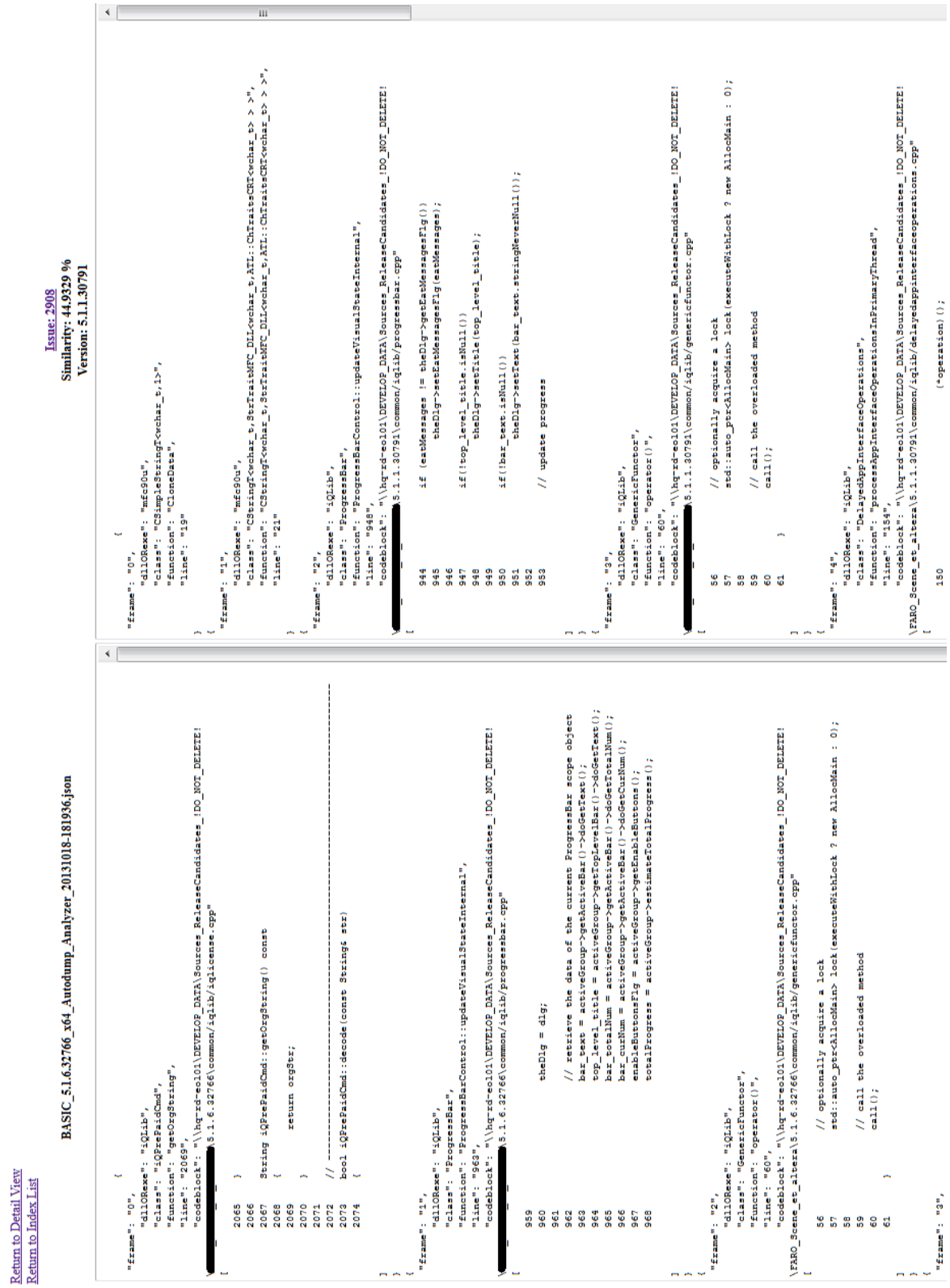


Abbildung 3.10: Detailansicht des Webtool mit Source Code Auszügen zu einem gewählten Fehlerbericht und seiner gewählten Vergleichsdatei

3.4.1 Der Manuelle Vergleich

Das Fehlerberichtsverwaltungssystem soll nicht nur von extern via Automatisierung mit neuen Daten gefüttert werden. Es kommt auch vor, dass Entwickler Bugs bei ihren Tests und Weiterentwicklungen auftun und diese in das Bug-Tracking-System einbringen wollen. Um eine konsolidierte Ordnung aufrecht zu erhalten, mussten die Entwickler bisher die Suchfunktion benutzen. Das automatisierte System ist um den manuellen Vergleich eines Fehlerberichts mit der Fehlerberichtsdatenbank erweitert, welches dem Entwickler die Möglichkeit bietet, ein Dump File direkt einzugeben.
Diese Direkteingabe wird mit allen vorhandenen Buckets abgeglichen und ihre Ähnlichkeit aufgeführt, wie beim automatisierten Teil des Systems.

Folglich wird dem Entwickler die aufwendige Sucharbeit abgenommen und die Chance, eine daraus hervorgehende Fehlzuweisung zu tätigen, reduziert. Private Schnelldiagnosen durch einzelne Entwickler sind bei diesem Teil des Systems auch problemlos möglich und können bei Entwicklungsfehlern helfen.

Bezogen auf den Workflow ergibt sich die Abbildung 3.11.

Diese Präsentation zeigt dem Entwickler allerdings nur die Informationen bezüglich möglicher Ähnlichkeiten zu bestehenden Buckets auf. Sie lässt bislang keinen automatisierten Eintrag in das Bug-Tracking-System zu. Dies ist jedoch kein Übersehen im Design, sondern vorerst Absicht. Das daraus resultierende Problem, dass Einträge von Entwicklern direkt ins System ohne Vergleichsdateien erfolgen und somit nicht in die Vergleiche einbezogen werden können, ist bekannt.
Begründet ist die Absicht keine manuellen Vergleiche ins Bug-Tracking-System zu übernehmen zum Einen, damit dass die Anzahl der von Entwicklern direkt gemeldeten Abstürze des Systems, die ein Dump File erzeugen, gering ist im Vergleich zur Anzahl der von den Nutzern eingehenden Fehlerberichten. Zum Anderen sind die meisten Abstürze, die ein Entwickler direkt erlebt, Abstürze in der Entwicklungsphase, also meist durch Flüchtigkeitsfehler verursacht und noch direkt zu beheben.
In der forschungsauftraggebenden Firma ist es daher üblich, nicht durch momentane Entwicklung bedingt entstehende Dump Files an die zuständige E-Mail-Adresse zu schicken. Dies erhöht zwar den Arbeitsaufwand, da entsprechende E-Mails geschickt werden und die für die Fehlerverarbeitung zuständigen Entwickler zusätzliche Fehlerberichte verarbeiten müssen. Auf Grund der geringen Häufigkeit des Vorkommens solcher Fehlerberichte ist dieser Mehraufwand jedoch kaum merklich und der Aufwand der Implementierung für ein Zuweisungssystem im manuellen Vergleich stünde zur Zeit in keiner Relation zum Nutzen.

Will man diesen Arbeitsaufwand schmälern, empfiehlt es sich den manuellen Vergleich um die Funktionalität zu erweitern, die es ermöglicht, die Vergleichsdaten wie im Hauptsystem anzuwenden. Die dafür nötigen Daten liegen alle vor, es müsste nur eine Erweiterung an den manuellen Vergleich angehängt werden, um die Zuordnungswünsche des Entwicklers umzusetzen. Jeder Entwickler könnte bei ihm entstehende Dump Files in das Bug-Tracking-System vollständig und tauglich für den Vergleich einbinden und nicht die Arbeit an

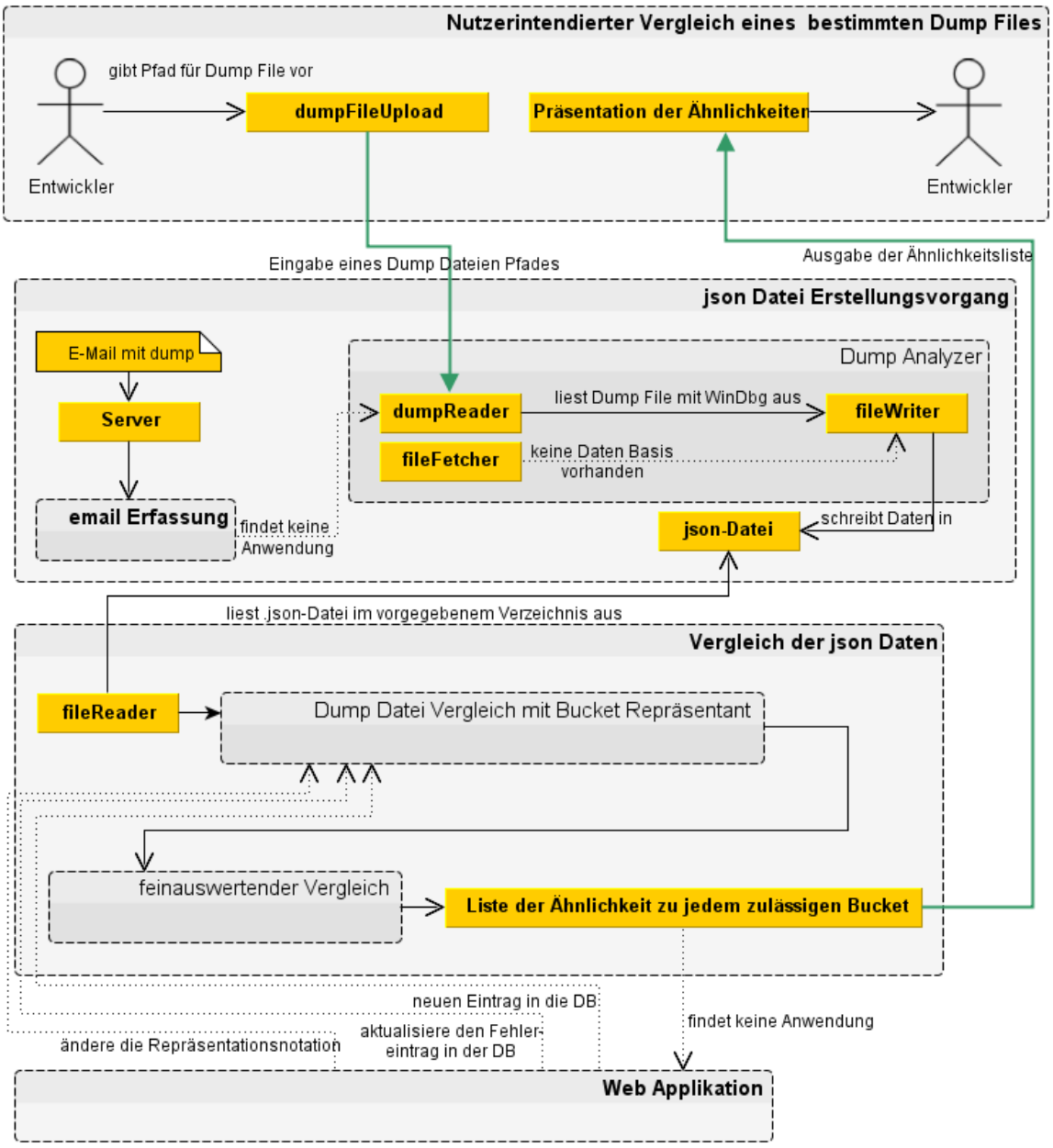


Abbildung 3.11: Workflow des manuellen Vergleichs

die Zuständigen abschieben und weiteren Zeitaufwand durch das Schreiben einer E-Mail erzeugen. Ferner wüsste der Entwickler, welche Funktionalität er genau durch seine Aktionen, die zum Absturz geführt haben, ausgeführt hat. Außerdem könnte der Entwickler sogar testen, ob der Absturz reproduzierbar ist und es sich bei der Absturzursache also um einen wirklichen Bug und nicht um einen Heisenbug handelt.

4 Ergebnisse

Ein zeitlicher Vorteil eines halb-automatischen Systems kann bereits vor dem Erheben der Ergebnisse ausgeschlossen werden, da ein vollautomatisches Verfahren nachweislich schneller ist. Diese Aussage lässt sich auch durch die in dieser Diplomarbeit verwendeten Veröffentlichungen stützen. Stattdessen wird bei der Untersuchung des vollautomatischen Systems der Unterschied zu der von Menschen erstellten Kontrollliste unter die Lupe genommen.

Dasselbe muss natürlich auch bei der Untersuchung bezüglich des Nutzens der menschlichen Kontrollinstanz betrachtet werden. Hier kommt hinzu, dass eine Untersuchung zur Dauer des Vorgangs erhoben wird, um festzustellen, ob das System eine akzeptable Zeitersparnis bringt.

4.1 Bestimmung der Kontrolldaten und des allgemeinen Aufwands

Die Basis zur Korrektheitsbestimmung besteht aus 194 Dateien, die vorab durch menschliche Begutachtung der beiden für die Fehlerberichtsverwaltung zuständigen Entwickler im Rahmen ihres Tätigkeitsauftrages verarbeitet wurden. Bei der Verarbeitung ging es um die Bestimmung, ob der Fehlerbericht als neu, einem Bekannten zuordenbar oder als invalide klassifiziert werden kann. Nebst dem Zuordnungsergebnis wurde für die Diplomarbeit zusätzlich der Zeitaufwand der Bestimmung für jeden Fehlerbericht und der darauf folgenden Verarbeitungszeit, um den Fehlerbericht in das Bug-Tracking System einzupflegen, erfasst.

Die Zuordnung ergab, dass es ohne unterstützende Systeme zu 56 neuen Buckets und 29 als invalide deklarierten Fehlerberichten kam. Wobei die Definition „Invalide“ hier zum Einen fehlerhafte und nichtssagende Fehlerberichte, als auch Fehlerberichte, die durch Funktionen Dritter verursacht wurden und daher nicht behebbar sind, umfasst. 109 Fehlerberichte konnten bestehenden Buckets zugeordnet werden.
Diese Zuordnung kann allerdings nur als Richtwert genommen werden, da die zuvor geschilderten Probleme bei der Bestimmung eine Rolle gespielt haben. Die einzigen Hilfsmittel, die die zuständigen Verarbeiter bei der Erstellung der Kontrollliste hatten, war ihr Wissen und das Bug-Tracking System mit seiner Suchfunktion.

Der Zeitaufwand, die Fehlerberichte zu analysieren, lag bei 1244 Minuten. Ihre weitere Verarbeitung nahm 411 Minuten in Anspruch und mit ihnen ergibt sich ein zeitlicher Gesamtaufwand von 1655 Minuten bzw. 27 Stunden und 35 Minuten. Diese Daten wurden über eine Zeitspanne von 102 Tagen erhoben.

4.2 Voll automatisiertes Ergebnis

Um das System als vollautomatisches System zu betrachten, muss das System in der Lage sein, auf Grund der gegebenen Ähnlichkeitsmaße zu entscheiden, wie es handeln soll: Ob es einen Fehlerbericht als neu klassifiziert oder ihn an einem bestehenden Bucket anfügt.
Hierzu gibt es viele Möglichkeiten der Entscheidung, daher wurden vorab drei Metriken bestimmt, die unter verschiedenen Aspekten unterschiedlich reagieren.

Simple Metrik: Es wird vorab ein für das Ähnlichkeitsmaß definierter Grenzwert festgelegt, ab dessen Erreichen die Fehlermeldung für den bestehenden Bucket vorgesehen ist. Sollten mehrere Ähnlichkeitswerte diesen Grenzwert überschreiten, so wird der größte Wert genommen. Gibt es mehrere Buckets mit demselben größten Wert, soll der Erste in dieser Liste - was hier, seiner Eintragsnummer nach, dem Ältesten entspricht - gewählt werden.

Revisionsbezogene Metrik: Bei der simplen Metrik gibt allein ein Grenzwert die Entscheidungsbasis vor. Die revisionsbezogene Metrik erweitert dies, sodass im Falle eines mehrfach vorkommenden größten Ähnlichkeitswertes nicht der älteste Eintrag in der Fehlerberichtsdatenbank sondern derjenige gewählt wird, dessen Vergleichsdatei die Revisionsnummer hat, die am nächsten zur Revisionsnummer des eingegangenen Fehlerberichts ist.
Sind auch hier wieder mehrere identisch passende Versionen vorhanden, wird, wie bei der simplen Metrik, der erste Eintrag, also der Bucket mit der niedrigsten Eintragsnummer, in der Liste genommen. Befindet sich die Fehlerberichtsversion exakt zwischen zwei passenden Versionen, wird die höhere Revisionsnummer gewählt, da der Fehler immer noch zu einem späteren Zeitpunkt, als zur Verwendung der gemeldeten Revision, Bestand hat.

Offene Fehler Metrik: Auch hier gibt ein Grenzwert wieder vor, ab wann ein Bucket relevant wird. Sollten mehrere Buckets denselben größten Ähnlichkeitswert haben, wird zuerst geschaut, welcher Bucket noch als offen klassifiziert ist. Sind mehrere noch offen, wird derjenige Bucket gewählt, dessen Vergleichsdatei im Bezug auf die Revisionsnummer dem eingegangenen Bericht am Nächsten liegt. Sind mehrere identische Revisionsnummern vorhanden, wählt das System den ersten Eintrag der in Frage kommenden Liste.
Offen bedeutet hierbei, dass der Eintrag noch als unbehandelt gilt und auch nicht als unbehandelbar eingestuft wurde.

4.2.1 Simples Verfahren

Die Festlegung des Grenzwertes erfolgt unter der Prämisse, dass das Ähnlichkeitsmaß entweder aufzeigt, dass die Call Stacks sich vom i-ten Frame an bis zum Ende hin komplett gleichen, der Weg zum Fehler also derselbe ist, oder der Call Stack vom Frame 0 bis zum $(i-1)$-ten Frame identisch ist, in der Annahme, dass in diesem Fall der Fehler in den identischen Frames zu finden ist.

Für diesen Durchlauf wurde der Grenzwert auf 49 Prozent gesetzt. Dieser Grenzwert definiert sich aus der Tatsache, dass Call Stacks als zuordenbar angesehen werden, sobald der oberste Frame identisch ist, oder sie vom zweiten Frame bis zum letzten Frame, sofern insgesamt mehr als zehn Frames im Stack sind, übereinstimmen. Sind es weniger als zehn Frames, besteht in den Testdaten immer eine Übereinstimmung des obersten Frames.
Rechnerisch lässt sich der Wert ermitteln, indem man aus der Gleichung 3.3 von Seite 35 den framebezogenen Teil der Rechnung nimmt, wie in Gleichung 4.1. In dieser Gleichung wird jeder Wert $i = [0..n]$ angewandt und dadurch der maximal erreichbare Faktor berechnet, den eine absolute Übereinstimmung der Frames an der Stelle i mit sich bringen kann.

$$tresh(i) = e^{-c*min(i)} \tag{4.1}$$

Unter der gegebenen Voraussetzung für c, dass die Summe aller Folgewerte von i nicht größer oder gleich des Ergebnisses für i sein darf, ergab sich der Grenzwert von 49. Dieser Wert garantiert die Übereinstimmung der Frames 0 bis $i - 1$ bzw. von i bis zum letzten Frame n, wenn der Ähnlichkeitswert über dem Grenzwert liegt. Die Framezahl i ging aus den zur Verfügung stehenden Fehlermeldungen hervor und dem Index der Frames in dem der Fehler im Schnitt am Häufigsten verursacht wurde.

Wie von Krohn-Hansen [KH12] dargelegt, soll der Fehler in fast allen Fällen innerhalb dieser ersten 10 Frames liegen. Damit der Fehler korrekt behandelt werden kann, muss ab seinem Auftreten seine Entwicklung jedes Mal einen sehr ähnlichen Verlauf nehmen.

Für die Bewertung der Entscheidungen des vollautomatischen Systems bleibt vorweg zu sagen, dass die von den Entwicklern erhobenen Kontrolldaten vorerst als korrekt angesehen werden, obwohl sie unter den Umständen erhoben wurden, die zu Beginn dieses Dokuments als unzureichend und fehlerbehaftet beschrieben wurden.

Belegt durch die erhobenen Daten, die in Abbildung 4.1 gezeigt werden, kann man im Vergleich zu den Kontrolldaten bei der Automatisierung auf Basis der simplen Verfahrensmetrik einen Trend zu Fehlentscheidungen feststellen.
Invalide Dateien werden vom System nicht erkannt. Dies wäre zwar im Fall der anfallenden Fehlerberichte, die dieser Arbeit zu Grunde liegen, meist möglich, aber mit einem erheblichen Aufwand verbunden. Es ist leichter, die Buckets, die aus invaliden Fehlerberichten bestehen, für die Entwickler zu kennzeichnen und zukünftige invalide Fehlerberichte so herausfiltern zu lassen. Viel gravierender als diese 9 Fehlzuweisungen sind die fehlerhaften Zuweisungen von Fehlerberichten zu Buckets, deren Fehler vollkommen anders ist, ihre Call Stacks aber genügend Ähnlichkeit haben.
Aus 194 Fällen sind es 6 Stück, die einem anderen Bucket zugeteilt werden und in Folge dessen in der Fehlerdiagnose eher Verwirrung erzeugen, als zum globalen Fehlerverständnis beizutragen.
Weitere 20 als invalide klassifizierte Fehlerberichte werden Buckets zugeteilt, in denen der Fehler tatsächlich mit dem Fehlerbericht korrespondiert. Die Zuweisung dieser 20 Stück bestätigt die Annahme, dass menschliche Verwaltung ohne Hilfsmittel das Bug-Tracking System in seinem Nutzen verzerrt, da das Wissen um Ähnlichkeiten fehlen kann und die Suche auf Grund subjektiver Suchparameter ungenügende Ergebnisse liefert.

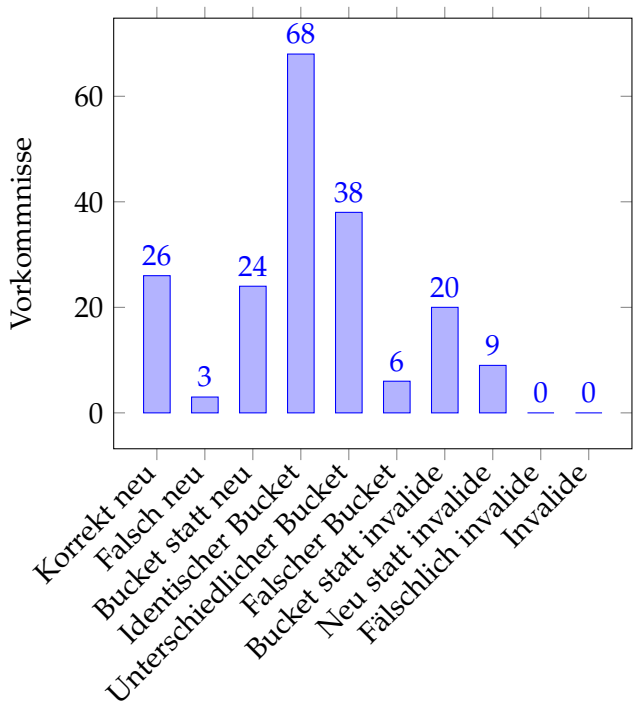


Abbildung 4.1: Ergebnisse des vollautomatischen Durchlaufs nach der simplen Verfahrensmetrik

Die 38 Fälle, in denen unterschiedliche Buckets als von den Kontrolldaten vorgesehen gewählt worden waren, sind Fälle in denen eine absolute Ähnlichkeit bzw. Fehlerverwandtschaft besteht und die vorgegebene Metrik nicht den Kontrollbucket wählen kann. Sie sind aber betreffend ihres Fehlers korrekt zugewiesen, und können höchstens noch Auswirkungen auf die Fehlerpriorität haben.

Bezugskategorie	F-Wert
Neue Buckets	0,4643
Selber Bucket	0,6239
Invalide	0

Tabelle 4.1: F-Wert Ergebnisse des simplen Verfahrens, bezogen auf die Kontrolldaten

Bezogen auf die Kontrolldaten lassen sich verschiedene F-Werte ableiten, welche in Tabelle 4.1 zu sehen sind.

Insgesamt lässt sich sagen, dass das simple Verfahren einen absoluten F-Wert von $0,4845$ im Bezug zu den Kontrolldaten hat. Dieser F-Wert liegt weit unter dem von Dang et al. [YDN12] postulierten durchschnittlichen F-Wert von $0,876$. Selbst wenn die 38 Zuweisungen, die wegen absolut ähnlicher Buckets unterschiedlich zur Kontrollliste zugewiesen worden sind, als korrekt gewertet werden, käme man hier nur auf einen F-Wert von $0,6804$.

4.2.2 Revisionsabhängiges Verfahren

Das revisionsabhängige Verfahren hatte bei den 194 Fehlerberichten 16 mal Anwendung gefunden. Von diesen 16 Anwendungen wurden 5 mit Änderungen gegenüber dem simplen Zuordnungsverfahren versehen. Diese Änderungen in der Gesamtverteilung und weitere Änderungen zeigen sich in Abbildung 4.2. Bezogen auf die Kontrolldaten hat sich die Ähnlichkeit zu den Kontrolldaten um 2,57% erhöht.

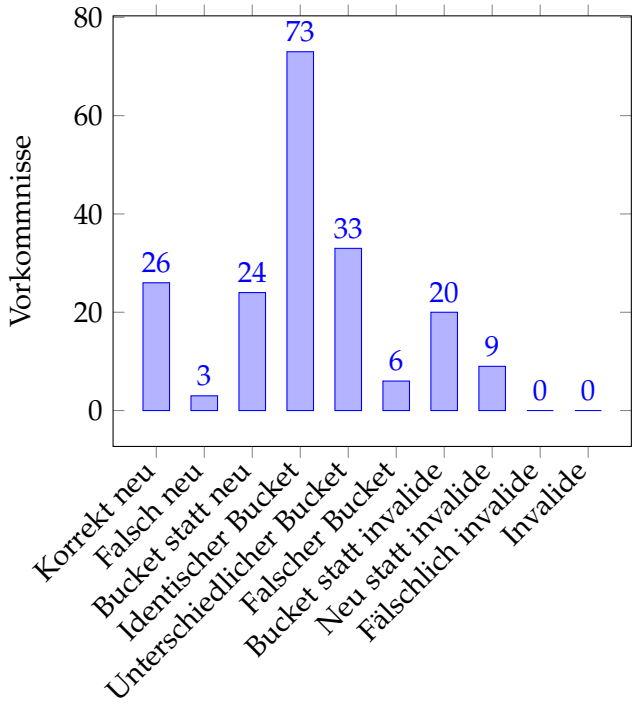


Abbildung 4.2: Ergebnisse des vollautomatischen Durchlaufs nach der revisionsabhängigen Verfahrensmetrik

Durch diese geänderte Zuweisung würde sich die Priorisierung der Fehler auch ändern. Die Chance, dass der Fehler zuerst in einer neueren Version der Software bzw. in derjenigen, die den Fehler am häufigsten verursacht, behoben wird, steigt mit der revisionsbezogenen Zuweisung deutlich an. Dies liegt daran, dass ab einem gewissen Alter der Revision zwar nach wie vor eine Priorisierung der Fehlerbehebung stattfindet, diese aber immer mehr als revisionsbezogen angesehen wird, während die Fehler neuerer Revisionen noch im Wertebezug zueinander gesehen werden. Fehler älterer Revisionen werden in ihrer revisionsbezogenen Behebung eher nur dann angegangen, wenn Zeit zur Verfügung steht.

Durch die Änderung bezüglich der Ähnlichkeit zu den Kontrolldaten ergibt sich eine Änderung der F-Werte, wie aus Tabelle 4.2 hervorgeht. Der F-Wert in Bezug auf das Gesamtbild aus Abbildung 4.2 beläuft sich auf 0,5103. Da es aber nur eine Umschichtung der erkannten

Buckets von unterschiedlich auf identisch bezüglich der Kontrollliste ist, kann keine Verbesserung des F-Werts auf zulässige Zuweisungen festgestellt werden. Er liegt nach wie vor bei 0,6804.

Bezugskategorie	F-Wert
Neue Buckets	0,4643
Selber Bucket	0,6697
Invalide	0

Tabelle 4.2: F-Wert Ergebnisse des revisionsbezogenen Verfahrens, bezogen auf die Kontrolldaten

4.2.3 Verfahren der offenen Fehlerbehandlung

Wie beim revisionbezogenen Verfahren, kam das Verfahren bezogen auf die als offen klassifizierten Buckets 16 mal zur Anwendung und brachte im Vergleich mit den Ergebnissen des simplen Verfahrens 7 Änderungen im Sinne der Kontrolldaten mit sich. Abbildung 4.3 zeigt die Verteilung bezüglich der Anwendung der offenen Metrik.
Somit ergibt sich auch hier wieder eine positive Änderung für den F-Wert bezüglich der Zuweisung zum selben Bucket, wie aus Tabelle 4.3 hervorgeht.
Der F-Wert für das gesamte Verfahren basierend auf den Kontrolldaten beläuft sich auf 0,5206.
Da auch hier nur wieder eine Umschichtung bezüglich der erkannten Bucketzuweisungen von unterschiedlich auf identisch statt gefunden hat, kann sich der F-Wert bezüglich der zulässigen Zuweisungen nicht verbessern. Er liegt nach wie vor bei 0,6804.

Bezugskategorie	F-Wert
Neue Buckets	0,4643
Selber Bucket	0,6881
Invalide	0

Tabelle 4.3: F-Wert Ergebnisse der offenen Fehlerbehandlung, bezogen auf die Kontrolldaten

4.2.4 Vergleich der automatisierten Verfahren

An Hand der Ergebnisse lässt sich sagen, dass, sofern man nur die Übereinstimmung der Kontrolldaten mit den Ergebnissen vergleicht, die automatisierten Verfahren deutliche Abweichungen zeigen.
Diese Abweichungen sind keine Aussage über die Inkorrektheit des Vergleichsverfahrens. Sie geben jedoch Anlass, an den Zuweisungen in den Kontrolldaten zu zweifeln.

Die Daten in Abbildung 4.4 lassen deutlich erkennen, dass je komplexer das Verfahren, desto besser die Entscheidungsgenauigkeit. Leider kann man auch erkennen, dass keine dieser

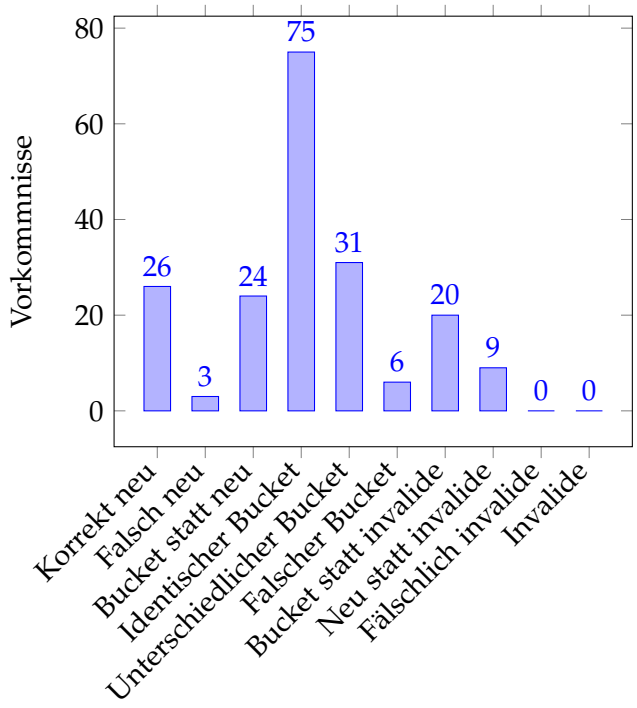


Abbildung 4.3: Ergebnisse des vollautomatischen Durchlaufs nach der offenen Verfahrensmetrik

Metriken eine Verbesserung der fehlerhaften Zuweisungen von invaliden Fehlerberichten mit sich bringt.

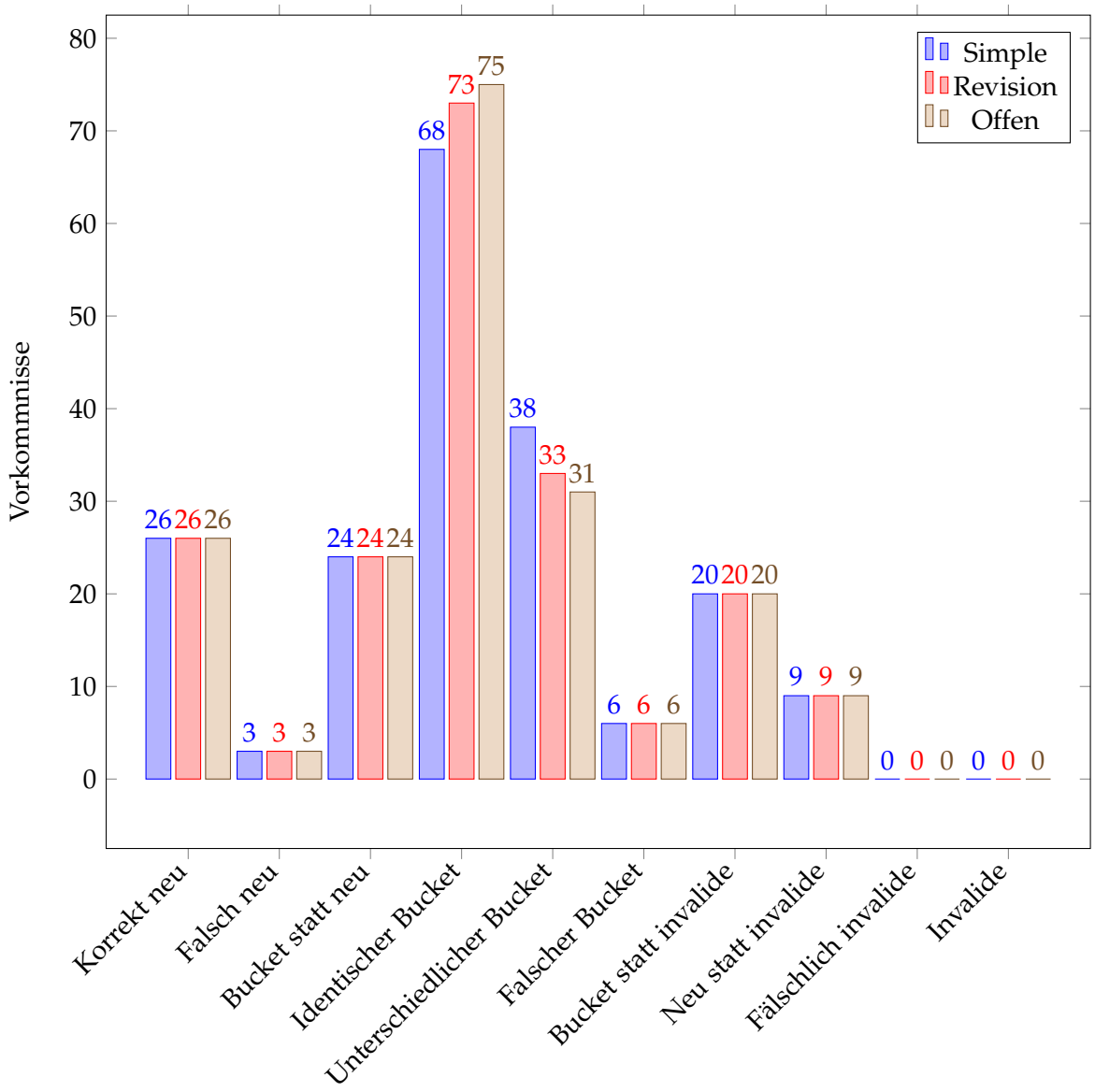


Abbildung 4.4: Ergebnisse der vollautomatischen Durchläufe im Vergleich

4.3 Menschliche Steuerung der Entscheidung

Wie in Kapitel 3 geschildert, ist in diesem System ein menschlicher Eingriff in die Entscheidungsfindung vorgesehen. Diese Entscheidungen wurden mit denselben Daten, die schon zur Kontrollbasisfindung und für die vollautomatischen Durchläufe genutzt worden waren, durchgeführt. Die Entscheidungen wurden von den zwei für den Bug-Tracking-Prozess zuständigen Entwicklern getroffen. Ihre Entscheidungen sind in Abbildung 4.5 festgehalten.

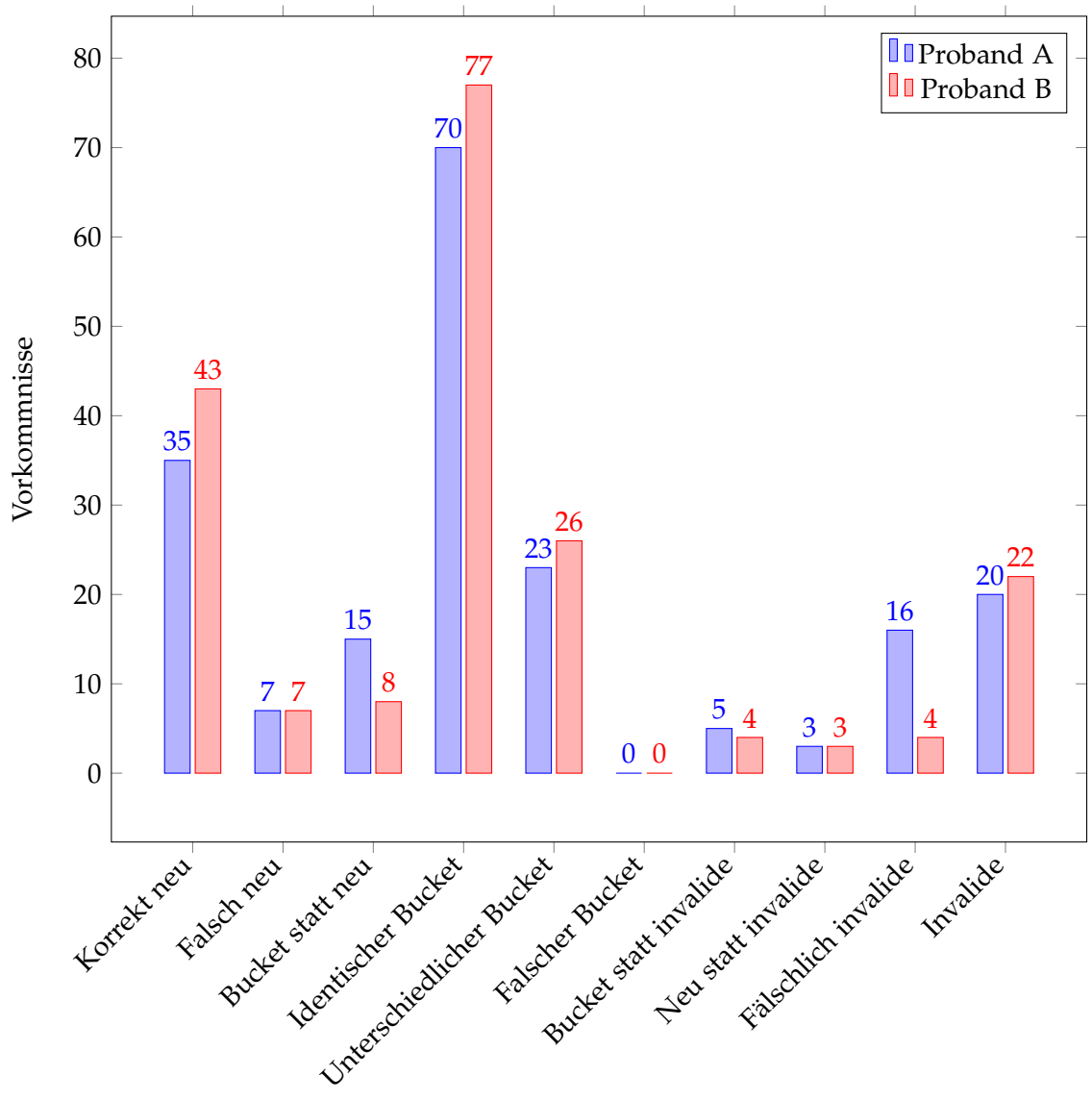


Abbildung 4.5: Ergebnisse des menschlichen Eingreifens in das System

Deutlich erkennbar ist, dass die Nutzung des Systems eindeutig zu anderen Ergebnissen führt, als wenn die beiden Entwickler sich nur auf ihr Wissen und die Suchfunktion des Bug-Tracking-Systems verlassen müssen. Dies zeigen die Werte der Spalten „Falsch neu“, „Bucket statt neu“ und „Unterschiedlicher Bucket“ in Abbildung 4.5.

Zu den bei beiden Probanden als „Falsch neu“ vermerkten 7 Fällen, muss gesagt werden, dass diese Dateien bei späterer Prüfung gegen die Kontrollmenge nicht denselben Fehlerverlauf haben, wie die Buckets, denen sie laut Kontrollliste zugewiesen hätten werden sollen. Sie sind also entgegen der Kontrollliste tatsächlich neue gültige Fehler und begründen folglich zu Recht neue Buckets. Ihre Ähnlichkeit reicht zwar aus, um bei schneller Sichtung eine Zuweisung zu bestehenden Buckets zu tätigen, was bei den automatischen Verarbeitungen auch passiert ist, doch wäre diese dann tatsächlich falsch. Das Feld „Falsch neu“ ist für die menschliche Entscheidung falsch benannt.

Bei Proband B waren die gegenüber den Kontrolldaten als „Fälschlich invalide“ klassifizierten Daten 4 Dateien, in denen der Windows Debugger nur einen nichts sagenden Thread erbracht hatte, wodurch kein Fehler durch Benutzung des Systems erkennbar war. Man müsste hierfür auf die alte Methode des Auslesens mittels Compiler zurückgreifen.
Proband A brachte zu diesen 4 Dateien auch noch subjektive Fehler hinein. Er erklärte einige Fehlerberichte, die Treiberversagen Dritter zur Ursache hatten, als invalide, anstatt sie, wie den Kontrolldaten nach intendiert, den zugehörigen Buckets zuzuordnen. Dies führt bei ihm zur erhöhten Summe von 16 bei seinen als „Fälschlich invalide“ definierten Fehlerberichten.

Die Werte in „Bucket statt invalide“ sind bei beiden Probanden damit zu begründen, dass es bei der Erhebung der Kontrolldaten subjektive Differenzen zur Gültigkeit einiger Fehlermeldungen gegeben hat, diese aber im System dann nicht mehr nachvollziehbar waren.

Die F-Werte beider Probanden sind der Tabelle 4.4 zu entnehmen.

Bezugskategorie	F-Wert Proband A	F-Wert Proband B
Neue Buckets	0,625	0,7679
Selber Bucket	0,6422	0,7064
Invalide	0,7241	0,7931

Tabelle 4.4: F-Wert Ergebnisse der menschlichen Entscheidung, bezogen auf die Kontrolldaten

Proband B schafft bezüglich der Kontrolldaten einen Gesamt-F-Wert von $0,7526$, Proband A hingegen auf Grund subjektiver Entscheidungen nur $0,6598$.

Nach wiederholten Kontrollen kann man bei genauer Betrachtung des Ergebnisses - mit Ausnahme einiger Ausreißer - sagen, dass die Zuordnungen von Proband B korrekter sind, als die Datenbasis der Kontrollliste, die von Hand erstellt worden war. Damit ist bestätigt, dass das Ergebnis der Nutzung des Systems gegenüber der menschlichen Verwaltung auf Basis von Wissen und der Bug-Tracking-Suchfunktion einen deutlich höheren Korrektheitsgrad hat.

4.4 Vollautomatisches System gegen menschliche Entscheidung

Wie im Abschnitt 4.3 als Schlussfolgerung festgestellt wurde, ist die handerstellte Datenbasis in sich, wie erwartet, fehlerbehaftet, wodurch Abweichungen von ihr durchaus korrekt und begründet sind.
Im Folgenden wird zuerst der Vergleich des vollautomatischen Systems gegen die menschliche Endentscheidung auf Basis der alten Datenbasis vollzogen. Danach wird von einer neu definierten Datenbasis durch die menschliche Nutzung des Systems ausgegangen, um die Korrektheit des automatischen Systems tatsächlich zu prüfen.
Auch werden die vollautomatischen Ergebnisse gegen die Ergebnisse von Proband B gewertet, da dieser nach wiederholter Prüfung der Daten am objektivsten war und seine Ergebnisse auch sinnvoller als die der erhobenen Kontrollliste waren.

	Simple	Revision	Offen	Proband B	Proband A
Korrekt neu	26	26	26	43	35
Falsch neu	3	3	3	7	7
Bucket statt neu	24	24	24	8	15
Identischer Bucket	68	73	75	77	70
Unterschiedlicher Bucket	38	33	31	26	23
Falscher Bucket	6	6	6	0	0
Bucket statt invalid	20	20	20	4	5
Neu statt invalid	9	9	9	3	3
Fälschlich invalid	0	0	0	4	16
Invalide	0	0	0	22	20

Tabelle 4.5: Ergebniswerte der automatischen und des menschlichen Zuordnens basierend auf den alten Vergleichswerten

	Simple	Revision	Offen	Proband B
Korrekt neu	27	27	27	53
Falsch neu	3	3	3	-
Bucket statt neu	23	23	23	115
Identischer Bucket	84	88	86	-
Unterschiedlicher Bucket	27	23	25	-
Falscher Bucket	4	4	4	-
Bucket statt invalid	18	18	18	-
Neu statt invalid	8	8	8	-
Fälschlich invalid	0	0	0	-
Invalide	0	0	0	26

Tabelle 4.6: Ergebniswerte der automatischen Zuordnungen basierend auf den Vergleichswerten gegeben von Proband B Entscheidungen

Der Vergleich der beiden Tabellen 4.5 und 4.6 ergibt eine Umschichtung von einer zuvor als invalide definierten Datei, die dann als neuer Bucket vermerkt wurde, direkt zu einer von Grund auf als neu zu definierenden Datei.

Auch verliert sich eine Datei aus der „Bucket statt neu“ Menge, da diese tatsächlich schon einen passenden Bucket im Bug-Tracking-System hatte, der allerdings bei der Kontrolldatenerstellung übersehen worden war. Der Anstieg in der Zuweisung zu identischen Buckets vom simplen Verfahren zum revisonsabhängigen Verfahren ist in den automatisierten Systemergebnissen alleine begründet. Der Abstieg von diesem Wert vom revisionsabhängigen zum offenen Bucket Verfahren lässt sich auch erklären. Allerdings würde man, wie zuvor mit den händisch erstellten Kontrolldaten, einen Anstieg der Zuweisung bei identischen Buckets erwarten. Hier spiegelt sich aber die nicht vollends ausräumbare Subjektivität der Probanden wider, die die Kontrolldaten beim Vergleich der Effizienz Mensch zu Maschine mitbestimmt. Auf Basis dieser nun durch die Vergleiche erstellten Daten erhält die automatische Auswahl keine kontinuierliche Übereinstimmung, sondern kann in den betreffenden Fällen den Fehlerbericht nur verwandten Buckets zuweisen.

Nach wie vor verteilen die automatischen Systeme die als neu definierten Buckets über die verschiedenen Systemansätze hinweg gleichermaßen. Die Änderung der Verteilung bezüglich der alten und neuen Kontrollwerte ergibt sich aus der Veränderung der Kontrolldaten. Die als invalide geltenden Dateien werden entweder bestehenden Buckets zugeordnet oder die System legen neue Buckets an, was 18 verteilte invalide Dateien bzw. 8 als neue Fehler kategorisierte Dateien erklärt.

Geht man nun davon aus, dass das rechnergestützte Verfahren mit menschlicher Kontrolle absolute Korrektheit garantiert, verändern sich natürlich auch die F-Werte für die einzelnen automatisierten Verfahren.

Bezugskategorie	Simple	Revision	Offen
Neue Buckets	0,5094	0,5094	0,5094
Selber Bucket	0,7304	0,7652	0,7478
Invalide	0	0	0

Tabelle 4.7: F-Wert Ergebnisse der automatisierten Verfahren, bezogen auf die menschlichen Entscheidungen mit Hilfe des computergestützten Zuordnungssystems

Tabelle 4.7 zeigt, dass es in den einzelnen Vergleichskategorien, bezogen auf die Daten in den Tabellen 4.1, 4.2 und 4.3, Verbesserungen um bis zu fast 11 Prozent gegeben hat.

Unter Berücksichtigung der Tatsache, dass fast alle Entscheidungen für einen anderen Bucket darauf beruhen, dass bei der Wahl zwischen zwei Buckets, es sich jeweils um eine Kopie oder eine Variation desselben Fehlers handelte, kommen alle automatischen Systeme, bezogen auf die von Proband B erhobenen Kontrolldaten, auf einen Gesamt-F-Wert von $0,7113$. Proband A kommt auf Basis dieser Berücksichtigung auf einen Gesamt-F-Wert von $0,7164$.
Dadurch liegt der Mensch, wenn auch nur knapp und trotz Missachtung mancher Zuweisungsmetriken, die zuvor vom Team aufgestellt worden waren, vor den Maschinen.

4.5 Nutzen des sich ändernden Bucket-Repräsentanten

Es ist über die Erhebung der Daten hinweg beobachtbar, dass sich der Repräsentant durchaus mit hinzukommenden Daten ändern kann. Dem erhofften Nutzen, somit eine bessere Zuordnung zu ermöglichen und die Aktualität des Bezugspunktes für den Vergleich zu garantieren, kann in seinen Auswirkungen jedoch nur weiterhin in der Theorie Bestand gegeben werden. Um die Annahme dahingehend zu stützen, dass man die Zuordnung bezüglich stark veränderter Codestrukturen, die aber nach wie vor derselbe Fehler sind, verbessern würde, würde es mehr Daten und vor allem eine längere Erhebungsperiode über deutlich mehr Revisionen benötigen.

Bei den Buckets, in denen es zu Änderungen des Repräsentanten kam, ergaben sich bezüglich der darauf folgenden Zuordnungswerte nur marginale Wertveränderungen, die sich im Schnitt maximal um 1 Prozent geändert hatten.
Mögliche zeitliche Beschleunigungen des Vergleichsvorgangs waren auch nicht erkennbar und es wurde nach wie vor dieselbe Datenmenge pro Datei und Durchlauf, mit oder ohne dem sich ändernden Bucket-Repräsentanten, verglichen.

4.6 Zeitliche Ersparnis

	Listenerstellung	Proband A	Proband B
Analysezeit	20:43:59	3:00:42	3:41:05
Erstellungszeit	6:51:00	2:00:00	3:41:00
Gesamtzeit	27:34:59	6:00:42	7:22:05
Durchschnittlicher Zeitaufwand	0:06:25	0:01:32	0:02:16

Tabelle 4.8: Zeitaufwände für die Verarbeitung der Fehlerberichte vor und nach der Systemerstellung zur Arbeitserleichterung

Die Daten zum Zeitaufwand in der Tabelle 4.8 zeigen deutlich, dass es eine Zeiteinsparung mit Benutzung des Systems gibt. Die Zeit, die der Entwickler damit verbracht hat, das Dump File zu laden und auf seinen Fehler hin zu analysieren und eine Entscheidung zur Zuweisung im Bug-Tracking-System zu treffen, ist um den durchschnittlichen Faktor von 6,27 gefallen.
Die für die Erstellung eines neuen Eintrags oder die Aktualisierung eines alten Eintrags im Bug-Tracking-System benötigte Zeit ist um den durchschnittlichen Faktor von 2,64 gefallen.
Bei der Erhebung von den Erstellungszeiten der Probanden wurde vom ungünstigsten Fall ausgegangen, der bei der Neuanlage von Buckets denselben Zeitaufwand in Anspruch nehmen würde, wie er es beim händischen Verfahren tun würde. Da das meiste aber schon automatisch durch das System vordefiniert wird, ist eine Verbesserung dieses Faktors gewiss.
Bezogen auf das Gesamtbild ergibt sich ein durchschnittlicher Zeitersparnisfaktor von 4,17 basierend auf der Annahme des ungünstigsten Zeitaufwandes.

Der durchschnittliche Zeitaufwand ist in diesem Rahmen um den Durchschnittsfaktor von 3,51 beschleunigt worden.

4.7 Einsatznutzen des Programms

Bei den Daten, die zur Verfügung gestellt worden waren, waren 87 Revisionen durchlaufen worden. Schon die Zuweisungsergebnisse der Maschinen sprechen dafür, dass unter anderem die Intervallregelung des Vergleichsalgorithmus eine nützliche Entscheidung war. Bisher hätte der Entwickler bei unterschiedlichen Revisionsnummern zwei verschiedene Entwicklungsumgebungsinstanzen mit der jeweiligen Revision öffnen müssen, um sich zu vergewissern, dass es sich bei den jeweiligen Funktionsaufrufen, deren Zeilennummern sich geändert haben, um dieselbe Stelle im Code handelt. Das System reduziert diesen Aufwand, indem es dem Entwickler diese Suche in den einzelnen Code Dateien abnimmt und die relevanten Daten auch noch vergleichbar im selben Fenster präsentiert. Dies ermöglicht zwei Fehler gleichzeitig miteinander zu vergleichen, ohne Entwicklungsumgebungen oder Editoren zu bemühen. Es wären hier auch mehrere Dateien im detaillierten Vergleich denkbar, der Übersichtlichkeit wäre dies aber nicht dienlich.
Trotzdem erspart dieser auf zwei Dateien bezogene Detailvergleich deutlich Zeit. Zumal während der Erhebungsphase beobachtet werden konnte, dass der Entwickler seltenst mehr als ein oder zwei Paarungen im detaillierten Vergleich pro zu verarbeitenden Fehlerbericht aufrief, um sich zu vergewissern, wo die Ähnlichkeiten oder Unterschiede liegen.

Weiterhin kann das System auch dabei helfen, die Anzahl an Buckets zu reduzieren und mehr Struktur in das Bug-Tracking-System zu bringen. Alleine bei der normalen Nutzung kann dem Entwickler beim Sichten der einfachen Aufführung von Ähnlichkeiten schon auffallen, dass Fälle existieren, in denen mehrere Einträge denselben Fehler betreffen. Diese Mehrfacheintragung eines Fehlers kann verschiedene Gründe haben:
Sei es weil die Korrespondenz nicht gefunden wurde oder Absicht, da es ein Fehler ist, der revisionsäbhängig behandelt werden soll.
Die revisionsabhängigen Buckets gleicher Fehler sind meistens schon verlinkt, da sie bewusst angelegt wurden. Dennoch hat man durch das Aufzeigen der Ähnlichkeiten eine Basis zur Kontrolle, welche Relationen bestehen oder fehlen. Bei Mehrfacheinträgen hat man nun die Möglichkeit entweder Relationen zueinander aufzubauen oder die Einträge zusammenzulegen. Damit verbessert sich auch die Prioritätensetzung, da, dieses erhaltene Wissen zur Anwendung gebracht, eine viel kleinere, dafür genauere Bucketmenge bewertet werden muss.
Diese Verbesserungen und Einsatzgebiete sind allerdings nur nebensächlich und müssen nach wie vor von einem Entwickler im Bug-Tracking-System umgesetzt werden.

Hauptnutzen sind das Vermindern des Zeitaufwandes bei der Verwaltung von eingehenden Fehlerberichtsmeldungen und das deutlich verbesserte und korrektere Einteilen dieser Fehlerberichte.
Die Entwickler erhalten somit mehr Zeit für die Fehlerbehebung oder für Weiterentwicklungen.

Die erstellte .json-Datei mit dem enthaltenen Call Stack bleibt vom System auch soweit erhalten, folglich bleibt auch der Call Stack in verarbeiteter lesbarer Form erhalten. Der Entwickler muss nicht mehr eine Entwicklungsumgebung starten und das zum Fehler gehörende Dump File einlesen lassen, um die wichtigsten Daten zum Fehlerverlauf zu erhalten. Er muss nur die .json-Datei in einem Texteditor öffnen und lesen. Selbst wenn er den Source Code dazu in Bezug setzen will, hat er es noch recht einfach, da die via .pdb-Daten auflösbaren Funktionen auch den Pfad zur Source Code Datei mit aufführen. Er muss den Pfad lediglich öffnen, was auch weitere Zeit spart.

5 Zusammenfassung und Ausblick

Die Aufgabe dieser Diplomarbeit war es, ein System zur Unterstützung und weitestgehender Automatisierung der Fehlerberichtsverwaltung zu entwickeln und zu prüfen, ob der menschliche Eingriff bei der Zuweisungsfindung gegenüber einem vollautomatischen System von Vorteil ist.

Bezüglich des alleinigen Aspekts der Zeitaufwandsreduzierung würde man sich verständlicherweise nicht für ein halb-automatisches System entscheiden, da seine Anwendung menschliche Ressourcen bindet. Gegenüber der händischen Verarbeitung ist das halb-automatische System jedoch in der Lage einen Ersparnisfaktor von mindestens $3,51$ gegenüber der vorher benötigten Zeit zu liefern.
Die Zuweisungsgenauigkeit des automatischen Systems steht denen der menschlichen Zuweisung nach, was auch die Ergebnisse belegt haben. Der Mensch ist in der Lage ein Grundverständnis für den Fehler verursachenden Code aufzubringen, während das automatische System nur Strings vergleichen und vorab definierten Regeln folgen kann. Dieses Grundverständnis ermöglicht dem Menschen bessere Entscheidungen zu treffen, wo die Maschine auf Grund ihrer Vorgaben versagen muss. Somit kann die Maschine ohne erheblichen Entwicklungsaufwand vorab nicht besser sein als der Mensch.
Damit der Mensch allerdings besser bleibt als die objektive Entscheidung der Maschine, muss die Subjektivität des Einzelnen weitestgehend eingeschränkt werden, indem man vorab Regeln und Metriken zu den Entscheidungen festlegt und sich daran hält. Der Unterschied der Ergebnisse zwischen Proband A und Proband B und ihr Zustandekommen zeigt deutlich auf, dass die vorab firmeninternen Metriken nicht eingehalten wurden. Werden die Metriken nicht eingehalten, so kann bei diesem System, sobald es von mehreren Benutzern gesteuert wird, der Nutzen schnell dezimiert werden, wie es sich bei Proband A angedeutet hat.

Bei der Entwicklung des Systems zur Fehlerberichtsverwaltung gab es einiges zu beachten, um von Nutzen für die Anwender zu sein.
Zu erwähnen ist die Berücksichtigung der revisionsbedingten Zeilenverschiebungen, die nicht im Voraus berechenbar sind. Dafür muss ein Intervall mit der höchsten Wahrscheinlichkeit festgelegt werden, um diese Verschiebung zu decken. Diese Wahrscheinlichkeitsbasierung des Intervallwerts führt nicht immer zu korrekten Ergebnissen. Die Abweichungen wurden jedoch von den Nutzern festgestellt und korrekt verarbeitet.
Weiters die Tatsache, dass mancher Call Stack ohne Einsicht in den Code eher nichtssagend ist und somit eine Detailansicht entwickelt werden musste, um dem Entwickler zusätzliche Informationen zu geben und ihm bei seiner Entscheidung der Zuordnung zu helfen. Außerdem verringert diese Detailansicht den Aufwand für den Entwickler in unsicheren Fällen, da er zur schnellen Code Analyse weder Entwicklungsumgebung noch Editor aufrufen muss.

Letztendlich wurde ein System entwickelt, das die Arbeit um mindestens Faktor 3,51 beschleunigen kann und dabei noch recht einfach im Aufbau und seiner Bedienung ist. Durch die Hilfe bei der Zuordnung kann das Bug-Tracking-System besser genutzt werden. Eine Priorisierung der Buckets ist nun möglich, da die Wahrscheinlichkeit sehr viel größer ist, dass die Anzahl der Fehlerberichte in einem Bucket auch der wirklich existenten Anzahl an Fehler entspricht, die der Eintrag im Bug-Tracking-System beschreibt.
Der Nebeneffekt diesen Systems ist es, dass die Bug-Tracking-Datenbank indirekt eine Konsolidierung erfährt, da Verwandtschaften, die den Entwicklern noch nicht klar waren, aufgezeigt werden können. Sie müssen nur von einem Entwickler im Bug-Tracking-System umgesetzt werden.

Würde man sich dennoch für die voll-automatische Variante entscheiden, um Zeit zu sparen, sollte man sich im Klaren sein, dass es vorerst nur eine vermeintliche Zeitersparnis ist. Diese Diplomarbeit hat sich damit zwar nicht beschäftigt, aber im Rahmen der Tests und der Zusammenarbeit mit der forschungsauftraggebenden Firma bildete sich ein Wissen diesbezüglich. Wo die Automatisierung Zeit beim Verwalten und Eintragen in ein Bug-Tracking-System einspart, kann diese Automatisierung im Nachhinein erneuten, ungewollten Aufwand fordern, nämlich durch Fehlentscheidungen. Diese Fehlentscheidungen müssen dann bei der Fehlerbehebung immer beachtet werden, weil die Buckets, welche Fehlentscheidungen enthalten, den Entwickler bei seiner Fehler behebenden Arbeit in die Irre führen können, da falsch zugewiesene Dumps im Bucket sind. Wenn ein Entwickler sich nicht mehr sicher sein kann, dass das System ihm korrekte Daten zur Verfügung stellt, wird der Nutzen des gesamten Systems in Frage gestellt.

Ausblick

Ein Ziel, welches für diese Arbeit wie auch für so viele andere Arbeiten auf diesem Themengebiet gilt, ist am Ende ein intelligentes System zur Fehlererkennung und Zuweisung zu entwickeln. Sei das System schon voll automatisiert oder noch mit menschlichem Kontrollschritt, Ziel muss es sein, die Systeme voll automatisiert laufen lassen zu können und eine höhere Trefferquote als 90 Prozent zu erreichen. Hierzu muss aber auch einiges bezüglich der Erkennung und Verwertung getan werden.
Dieses System könnte beispielsweise als Basis zum gesteuerten Lernen eines später voll-automatischen Programms angewandt werden. Der Mensch steuert hierbei anfangs noch alle Entscheidungen und das System lernt aus den Entscheidungen und wird somit immer mehr Entscheidungsmuster erkennen und verstehen, bis es schließlich autonom agieren kann.

Ein Schritt in Folge dieser Arbeit bezüglich des bestehenden Systems wäre es, über Hilfsmittel zur Erkennung, wie Einfärben der gemeinten Sequenz in den verglichen Call Stacks, im Web Interface nachzudenken und diese umzusetzen, um eine noch schnellere Erkennbarkeit der Ähnlichkeit zu ermöglichen. Dadurch würde der Zeitaufwand bei Unsicherheiten des Entwicklers noch weiter sinken. Die bislang einfach wirkende Oberfläche war allerdings eine Anforderung, um die Nutzer vorerst nicht vom Wesentlichen abzulenken.

Was sich definitiv aus dieser Arbeit ableiten lässt, ist die Möglichkeit zur Erforschung der Frage, wo mehr Zeitaufwand entsteht: Bei der halbautomatischen Fehlerberichtsverwaltung oder bei der automatischen Fehlerberichtsverwaltung bezüglich der Zeit, die es benötigt die Fehler im Bezug auf fehlerhafte Bucketzuweisungen zu beheben.

Auch ließe sich im Anschluss an diese Arbeit ein System zur automatischen Fehlerpriorisierung entwickeln.

Literaturverzeichnis

[Bug] Bugzilla. URL https://bugzilla.mozilla.org/. (Zitiert auf Seite 9)

[dai08] Get stack trace from dump file, 2008. URL http://daikof622.wordpress.com/2008/10/17/get-stack-trace-from-dump-file/. (Zitiert auf Seite 30)

[Hei] Heisenbug. URL http://en.wikipedia.org/wiki/Heisenbug. (Zitiert auf Seite 13)

[JL13] M. M. Johannes Lerch. Finding Duplicates of Your Yet Unwritten Bug Report. 2013. (Zitiert auf den Seiten 21 und 27)

[jso] JavaScript Object Notation. URL http://json.org/. (Zitiert auf Seite 29)

[KH12] H. Krohn-Hansen. Program Crash Analysis: Evaluation and Application of Current Methods, 2012. URL https://www.duo.uio.no/handle/10852/9057. (Zitiert auf den Seiten 22 und 53)

[Man] Mantis Bug Tracking. URL http://www.mantisbt.org/. (Zitiert auf Seite 19)

[MB05] G. L. L. M. M. W. J. C. P. S. Mark Brodie, Sheng Ma. Quickly Finding Known Software Problems via Automated Symptom Matching. 2005. (Zitiert auf den Seiten 20, 21 und 27)

[MSV] Microsoft Visual Studio. URL http://msdn.microsoft.com/de-de/vstudio. (Zitiert auf Seite 14)

[MSW] Windows Error Reporting. URL http://msdn.microsoft.com/en-us/library/bb513641(v=vs.85).aspx. (Zitiert auf den Seiten 9 und 14)

[pdb] Programmdatenbankdatei. URL http://msdn.microsoft.com/de-de/library/aa292304%28v=vs.71%29.aspx. (Zitiert auf Seite 16)

[Polo8a] J. Poley. Getting the Crash Details from a .DMP File (Automating Crash Dump Analysis Part 3), 2008. URL http://blogs.msdn.com/b/joshpoley/archive/2008/06/06/getting-the-crash-details-from-a-dmp-file-automating-crash-dump-analysis-part-3.aspx. (Zitiert auf Seite 30)

[Polo8b] J. Poley. Prolific Usage of MiniDumpWriteDump (Automating Crash Dump Analysis Part 0), 2008. URL http://blogs.msdn.com/b/joshpoley/archive/2008/05/19/prolific-usage-of-minidumpwritedump-automating-crash-dump-analysis-part-0.aspx. (Zitiert auf Seite 30)

[pro] google-breakpad Exception Handling. URL http://code.google.com/p/google-breakpad/wiki/ExceptionHandling. (Zitiert auf Seite 14)

[THC07] R. R. u. C. S. Thomas H. Cormen, Charles E. Leiserson. *Algorithmen - Eine Einführung*. Oldenbourg Wissenschaftsverlag, 2. auflage Auflage, 2007. (Zitiert auf den Seiten 23 und 35)

[Win] Microsoft Windows Debugger. URL http://msdn.microsoft.com/en-us/library/windows/hardware/ff551063%28v=vs.85%29.aspx. (Zitiert auf Seite 16)

[Win11] Exception Handling - Inform your users! Part 2, 2011. URL http://win32easy.blogspot.de/2011/03/exception-handling-inform-your-users_26.html. (Zitiert auf Seite 30)

[YDN12] H. Z. D. Z. Yingnong Dang, Rogxin Wu, P. Nobel. ReBucket: A Method for Clustering Duplicate Crash Reports Based on Call Stack Similarity, 2012. (Zitiert auf den Seiten 21, 23, 27, 35, 39 und 54)

Alle URLs wurden zuletzt am 18. 12. 2013 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift