Techniken zum Schätzen des Aufwands für Fehlerbehebung

Tim Jammer und Sammantha Strobel

Im Seminar Software-Reengineering
bei Prof Dr.-Ing.habil. Matthias Riebisch &
Dipl.-Inform. Sebastian Gerdes

an der Universität Hamburg
MIN-Fankultät, FB Informatik
Softwareentwicklungs- und –konstruktionsmethoden SWK

Zusammenfassung. Diese Arbeit vergleicht verschiedene Schätzverfahren für Softwareprojekte dahin gehend, wie geeignet diese Schätzmethoden zur Schätzung des Aufwandes der Fehlerbehebung sind.
Dazu werden in Abschnitt 2 die Grundlagen eines Fehlerbehebungsprozesses erläutert.
In Abschnitt 3 wird erläutert, wie man Fehler klassifizieren kann.
Der Abschnitt 4 bildet den Hauptteil der Arbeit. In ihm werden zunächst die Cocomo-Methode, das Function-Point-Verfahren, sowie das Verfahren von Walston und Felix und die Delphi-Methode verglichen.
Danach folgt ein Vergleich mit einer neueren, aus dem Bereich Big Data und Machine Learning stammenden Methode (Abschnitt 4.3), um anschließend (in Abschnitt 4.4) festzustellen, welche Faktoren den Aufwand für die Fehlerbehebung maßgeblich beeinflussen. Wichtiges Ergebnis dabei ist, dass die Qualifikation, der am Fehlerbehebungsprozess beteiligten Personen, einer der ausschlaggebensten Faktoren ist.
Im die Arbeit abrundenden Fazit (Abschnitt 5) werden die Ergebnisse noch einmal übersichtlich in Tabelle 2 zusammengefasst.

Schlüsselwörter: Softwarefehler, Aufwandsschätzung, Fehlerbehebung, Fehlerbehebungsaufwand

1 Einleitung

„Fehlerfreie Programme gibt es nicht, sagen Entwickler“ [Klo05]. Deshalb ist es umso essenzieller, der Fehlerbehebung von Software eine besonders wichtige Bedeutung zuzuschreiben.

Software nimmt im heutigen Zeitalter stetig an Komplexität und Umfang zu und wird demnach von Jahr zu Jahr fehleranfälliger. Darüber hinaus bestimmen Programme über einen Großteil unseres Alltags und sind wesentlich für die Sicherheit des Menschen verantwortlich, sei es für lebenserhaltende Geräte im Krankenhaus oder Software in Transportmitteln wie Auto und Flugzeug. „Welch gefährliche

Folgen das haben kann, zeigte jüngst das Beispiel von General Motors: Ein Software-Defekt bei Fahrzeugen der Marke Chevrolet soll Menschen das Leben gekostet haben, indem er das Auslösen von Airbags bei Unfällen verhinderte“ [Boh14]. Fakt ist, ein Menschenleben darf nicht gefährdet werden.

Jedoch sind Software unternehmen laut Bohnet [Boh14] meist unter immensem Zeit- und Kostendruck, sodass die Entwickler darunter leiden und durch sogenannte Hacks, Fehler zwar kurzfristig beheben, aber im Gegenzug weitere Fehler provozieren. Des Weiteren stellt die Fehlerbehebung oftmals einen enormen Aufwand dar, da es für Entwickler meist schwer ist den Fehler zu reproduzieren, aufgrund von schlechten Fehlerberichten und mangelnder Dokumentation [SKP14]. Darüber hinaus sind Fehler in weiterentwickelter und wiederverwendeter Software riskant zu beheben, da diese mehrere unterschiedliche Komponenten beeinflussen können [SKP14]. Jedoch diese zu vernachlässigen oder gar nicht zu beheben darf keine Alternative darstellen. Aus diesem Grund ist es unerlässlich der sorgfältigen Zeit- und Aufwandsplanung der Behebung von Fehlern eine bedeutende Rolle zuzusprechen, sodass Entwickler den Ansprüchen nach einer funktionierenden und sicheren Software gerecht werden können.

Nun stellt sich die Frage, wie dieser Aufwand zur Fehlerbehebung möglichst genau bestimmt werden kann, damit den Entwicklern ausreichend Zeit zur Verfügung steht. Explizit mit dieser Thematik beschäftigt sich diese Seminararbeit. Die Problematik bezieht sich demnach auf die Findung einer geeigneten Methode zur Aufwandsschätzung der Fehlerbehebung.

In Kapitel 2 werden die Grundlagen der Fehlerbehebung thematisiert. Anschließend werden in Kapitel 3 verschiedene Arten von Bugs vorgestellt, sowie Klassifizierungsmerkmale um Bugs unterschiedlich bewerten zu können. An dieser Stelle sei gesagt, dass „Bug“ im Folgenden als Synonym für Fehler genutzt wird. Kapitel 4 stellt den Hauptteil dieser Seminararbeit dar. In diesem werden unterschiedliche Methoden zur Aufwandsschätzung der Fehlerbehebung präsentiert und darauf aufbauend nach ihrer Güte bewertet. Da es hierzu jedoch noch einen Mangel an Literatur gibt und die Methoden zur Fehlerbehebung sich ausschließlich auf Big Data und Machine Learning beziehen, werden die Methoden zur Schätzung des Aufwands der Softwareentwicklung auch untersucht. Diese werden vorgestellt und darauf untersucht, inwiefern diese für die Aufwandsschätzung der Fehlerbehebung geeignet sind. Daraufhin werden Methoden, welche ausschließlich für die Fehlerbehebung entwickelt sind, präsentiert. Anschließend werden in Kapitel 5 die Einflussfaktoren zur Aufwandsschätzung behandelt, sowie deren Gewichtung in den einzelnen Methoden. Die Seminararbeit endet mit der Zusammenfassung der Untersuchungsergebnisse in Kapitel 6 und der darauffolgenden Präsentation der verwendeten Literatur.

2 Grundlagen der Fehlerbehebung

In diesem Kapitel wird erläutert was mit einem Bug passiert bzw. welche Stadien ein Bug durchläuft nachdem dieser gemeldet wurde. Er stellt das Grundprinzip der Fehlerbehebung vor.

2.1 Der Bug-Begriff

Unter Bug versteht man in der Programmierwelt einen Software-Fehler. „Im Allgemeinen handelt es sich dabei um ein Fehlverhalten eines Computerprogramms. Unterschieden wird dabei zwischen Syntaxfehlern, Laufzeitfehlern, Designfehlern, logischen Fehlern und Fehlern im Bedienkonzept“ [AG]. Bug kommt aus dem Englischen und bedeutet übersetzt Käfer. Die am häufigsten beschriebene Theorie, weshalb sich das Wort „Käfer“ als Synonym für Software-Fehler etabliert hat, ist auf das Jahr 1947 zurückzuführen, als Fehlfunktionen an den damals noch mechanisch laufenden Großrechnern durch Insekten, welche sich im Rechner verfangen haben, verursacht wurde [ITw].

2.2 Der Bug Lebenszyklus

In Abbildung 1 wird der Lebenszyklus eines Bugs, vom Zeitpunkt der Fehlermeldung bis zur Behebung, vereinfacht dargestellt und spielt sich wie folgt ab.

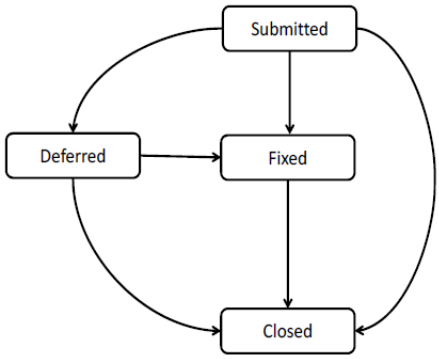

Abb. 1: The simplified bug-handling process [ZGV13]

Sobald ein Bug identifiziert wird, wird dieser in das Bug Tracking System übernommen. Neue Bugs werden dem Entwicklungsmanager des betreffenden Releases zugewiesen. Dabei stellt der Antragsteller den Fehlerbericht mit Informationen bezüglich des Bugs bereit, um schnellstmöglich den Bug identifizieren zu können und zu beseitigen. Sobald ein Bug erzeugt wird, wird dieser umgehend vom Entwicklungsmanager geprüft. Dieser prüft ob alle des Bugs bezüglichen

Informationen erfasst und korrekt sind, prüft ob die Prioritätenzuteilung zu den Bugs korrekt ist und leitet diese an einen Entwickler weiter, welcher nun für die Behebung des Fehlers zuständig ist. Daraufhin versuchen die Entwickler die Fehlerursache aufgrund des Fehlerberichtes zu bestimmen. Dauert die Behebung des Bugs länger und kann somit nicht mehr im aktuellen Release behoben werden, übergeht der Bug in den Zustand „Deferred“. Kann der Entwickler den Bug sofort beheben, müssen die Änderungen im Quellcode angepasst werden und der Bug wird als „Fixed“ markiert. Anschließend wird dieser vom verantwortlichen Entwicklungsmanager geprüft und geschlossen und verlässt den bug-handling process (vgl. [ZGV13]). Nach dem simplified bug-handling process wird deutlich, dass das Beheben von Bugs kein schnelles Verfahren ist, sondern sukzessive abgearbeitet wird. Eine sehr wichtige Rolle spielt dabei die Dokumentation, die auch ausschlaggebend für die schnelle Behebung des Bugs ist [ZGV13].

2.3 Timeline eines Bugs

Eine weitere, sehr informative Möglichkeit der Behandlung eines Bugs haben Saha et. Al [SKP14] in ihrem Paper aufgezeigt. Anhand einer Timeline werden die verschiedenen Zeitspannen zwischen den Ereignissen dargestellt. Abbildung 2 zeigt eine so etwaige Bug Timeline.

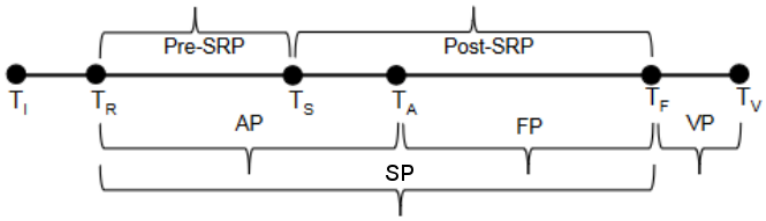

Abb. 2: Timeline of a Bug. nach [SKP14]

T_I bezeichnet den Zeitpunkt an dem der Code, in welchem sich der zu untersuchende Bug befindet, das erste Mal ausgeliefert wird. T_R bezieht sich auf das Ereignis, wenn der Bug erstmals von einem Entwickler oder Nutzer entdeckt und festgehalten wird. T_S steht für Severity, den Schweregrad des Bugs, und bezeichnet demnach den Zeitpunkt, an welchem die Einordnung in einen Schweregrad das letzte Mal von einem Entwickler gemacht wurde. T_A stellt den Zeitpunkt dar, an welchem der Bug dem richtigen Entwickler bzw. den richtigen Entwicklern, meist vom Bugtracking System, zugewiesen wird. T_F steht für Fixed und bezeichnet den Zeitpunkt an dem der Bug von einem Entwickler als Fixed markiert wird. Demnach stellt T_V den Zeitpunkt da, an welchem der Bug als behoben bestätigt wird. AP bezeichnet die Dauer, ausgehend von T_R , der Eröffnung eines Bugs bis zur Zuweisung zu einem der Entwickler. Die Fixing Period FP beschreibt, wie lange ein Entwickler benötigt um einen bestimmten

Bug zu beheben. FP ist jedoch nicht die reine Codierzeit. VP bezeichnet dabei die Zeitspanne, ausgehend von T_F, bis der Bug verifiziert wird. Die essenzielle Lebenszeit eines Bugs kennzeichnet SP, denn SP ist die Zeitspanne zwischen dem Ereignis der Entdeckung des Bugs bis zu seiner Behebung. Die Pre-Severity Realization Period (Pre-SRP) stellt die Festlegungsphase auf einen bestimmten Schweregrad des Bugs dar, d.h. wie viel Zeit beansprucht wird um die Schwere des Bugs festzulegen. Die Post-Severity Realization Period (Post-SRP) ist demzufolge die Zeit die aufgewendet wird um den Bug, nach Festlegung des Schweregrads, zu beheben (vgl. [SKP14]).
Um den Aufwand zur Fehlerbehebung minimieren zu können, müssen die einzelnen Zeitspannen AP, FP und VP so gering wie nötig gehalten werden. Ziel, um den genauen Zeitaufwand vorhersagen zu können, ist es, diese Zeitspannen möglichst präzise ermitteln zu können

3 Klassifizierung von Bugs

Dieses Kapitel stellt ein Beispiel dar, wie Bugs sinnvoll gegliedert und kategorisiert werden können, wobei hier lediglich Laufzeitfehler betrachtet werden, da diese als sehr kritisch gelten [TLL+14]. Dieses Kapitel dient zur Verständlichkeit und soll das Bug-handling etwas näherbringen, ist jedoch nicht Hauptteil dieser Arbeit. Im Folgenden werden verschiedene Kategorien vorgestellt, in welche Bugs zugeordnet werden und anschließend aufgrund dieser ihren Klassen zugeordnet werden.

3.1 Arten von Bugs

Diese Arbeit umfasst lediglich die Semantic Bugs, Concurrency Bugs und Memory Bugs. Diese lassen sich wiederum untergliedern, was jedoch den Rahmen dieser Seminararbeit überziehen würde.

Um den zu betrachtenden Bug etwas näher identifizieren, die Ursache und den Einfluss herauszufinden und ihn darauffolgend klassifizieren zu können, lehnt sich diese Arbeit an das Konzept von Tan et. Al. [TLL+14]. Laut Tan et. al. gibt es drei dominante Arten von Bugs, welche im Folgenden vorgestellt werden.

1. Memory Bugs sind Fehler, die durch unsachgemäßes handhaben von Datenobjekten entstehen.
2. Concurrency Bugs sind Synchronisationsprobleme zwischen nebenläufigen Tasks in konkurrenten Programmen, einschließlich Deadlocks.
3. Semantic Bugs stellen Inkonsistenzen mit den Anforderungen oder der Intention der Entwickler dar und sind weder Memory Bugs, noch Concurrency Bugs.

3.2 Analysieren der Bugs

Eine genauere Analyse des Bugs ist erforderlich um diesen anschließend richtig einordnen zu können und um eine nähere Fehlerprognose zu erstellen, um das

Bug-Fixing zu vereinfachen. Wir lehnen uns auch hier an das Konzept der Bug categories von Tan et. Al. [TLL+14].
Um nachfolgend beurteilen zu können welche Priorität und welche Schwere der Bug hat, ist es notwendig die Auswirkungen des Bugs zu erkennen.
Die Bugs werden in die nachstehenden Einfluss-Dimensionen eingeteilt:

- Confidentiality: berücksichtigt unautorisierte Offenlegung von Informationen
- Integrity: berücksichtigt unautorisierte Modifizierung
- Availability: berücksichtigt Störungen des Service
- Access: betrachtet den unautorisierten Zugriff

Tan et. al. [TLL+14] Unterscheidet zwischen den folgenden Auswirkungen:

- Hang: Die Software funktioniert, aber reagiert nicht
- Crash: Die Software unterbricht/blockiert ungewöhnlich
- Data Corruption: Fehlerhaftes ändern der Nutzerdaten
- Performance Degradation: Das Programm läuft korrekt, ist/reagiert aber sehr langsam
- Incorrect Functionality: Unerwartetes Systemverhalten
- Other: Bugs, die andere Auswirkungen beinhalten

Des Weiteren werden die Bugs in die nachstehenden Software Komponenten untergliedert:

- Core: Bugs, welche die Kernfunktionalität beeinträchtigt
- GUI: Bugs, welche das graphische User Interface beeinträchtigen
- Network: Bugs, welche die Netzwerkumgebung und die Kommunikation beschränken
- I/O: Bugs, welche die I/O Handhabung beeinträchtigen

Ferner gliedert Tan et. al. [TLL+14] Bugs in fünf unterschiedliche OS Komponenten:

- Drivers: Bugs in Bezug auf die Gerätetreiber
- Core: Bugs im Kernel Verzeichnis „mm“, „kernel“ und „include“
- Network: Bugs, welche die Netzwerkumgebung und die Netzwerkkommunikation beeinträchtigen
- File system: Bugs, welche das Dateisystem beeinträchtigen
- Architecture: Bugs, welche die Hardware-Architektur beeinflussen

3.3 Einordnung nach Schweregrad

Nachdem ein Bug gründlich analysiert wurde, kann dieser in unterschiedliche Schweregrade eingeordnet werden und anschließend einem geeigneten Entwickler zum Beheben zugeordnet werden. Die verschiedenen Level des Schweregrades stellen den potenziellen Schaden dar und lehnen sich an die Arbeit von Saha et. Al. [SKP14] und werden im Folgenden vorgestellt.

Blocker – Ein Blocker bezeichnet einen Bug, welcher die Entwicklung blockiert und/oder die zu testenden Komponenten. Für einen Blocker gibt es keine provisorische Lösung. Ein solcher Bug sollte schnellstmöglich bereinigt und geschlossen werden.

Critical – Critical Bugs verursachen das Abstürzen des Programmes, können Datenverluste verschulden und ernstzunehmende Speicherlecks.

Major – Durch einen Major Bug hat einen entscheidenden Verlust an Funktionalität zur Folge.

Normal – Normal Bugs sind häufig vorkommende Fehler. Unter spezifischen Umständen können auch diese einen Verlust an Funktionalität verursachen.

Minor – Diese Bugs können einen geringen Verlust an Funktionalität verursachen, sowie andere Probleme, meist ausgelöst durch eine Übergangslösung eines anderen Bugs.

Trivial – Triviale Bugs sind hauptsächlich Schönheitsfehler, wie falsch geschriebene Wörter oder versetzter Text. Diese haben jedoch keine beeinträchtigenden Auswirkungen auf ein Fehlverhalten des Systems.

Der Schweregrad nimmt sukzessive, anfangend beim Blocker, ab. Deshalb sollte Blockern, Critical Bugs und Major Bugs eine sehr hohe Priorität zugewiesen werden und unverzüglich behoben werden, auch wenn diese einen sehr hohen Aufwand mit sich bringen. Major Bugs und Normal Bugs sollten in jeden Fall behoben werden, sind aber nicht kritisch für das System und können demnach auch erst einmal in die Warteschlange gesetzt werden. Triviale Bugs hingegen sind Schönheitsfehler und haben keine Auswirkungen auf die Funktionalität des Systems. Die Behebung dieser dient dazu, andere Bugs einfacher erkennen zu können, aber es ist nicht zwingend notwendig diese zu korrigieren.

4 Methoden zur Aufwandsschätzung der Fehlerbehebung

In diesem Teil der Seminararbeit werden verschiedene Techniken zum Schätzen des Aufwands der Fehlerbehebung von Software vorgestellt und anschließend bewertet. Es werden zuerst die Kriterien, anhand welcher die Methoden bewertet werden, vorgestellt. Anschließend werden vier allgemeine Techniken zur Aufwandschätzung der Softwareentwicklung präsentiert und diese in Bezug auf ihre Eignung bewertet.

4.1 Bewertungskriterien

Um die unterschiedlichen Techniken sinnvoll bewerten und beurteilen zu können, inwiefern welche Technik für die Aufwandschätzung der Fehlerbehebung geeignet ist, werden drei Kriterien festgelegt. Bei der Wahl der Kriterien haben wir uns

erstrangig auf die Frage - „Was macht ein gutes Verfahren aus?“ - fokussiert. Aufgrund dieser Fragestellung sind wir zu dem Entschluss gekommen, dass die im Folgenden beschriebenen Kriterien wegen ihrer Aussagekraft bezüglich einer sinnvollen Bewertung für diese Arbeit sehr zutreffend sind.

Genauigkeit – Die Genauigkeit beurteilt die Exaktheit der Ergebnisse dieser Methode bzw. wie hoch die Fehlerquote der Aufwandsschätzung ist. Denn sind die Ergebnisse ungenau, stellen diese keinen Mehrwert dar. Liegt beispielsweise die Genauigkeitsquote einer Methode bei 30%, so ist sie nicht sehr zuverlässig und die Ergebnisse nicht aussagekräftig genug.

Aufwand – Das Kriterium Aufwand bezieht sich auf die Komplexität der Methode und den dadurch resultierenden Aufwand, welcher notwendig für die Vorbereitung, Durchführung und Auswertung dieser Methode ist. Der Aufwand im Sinne dieser Seminararbeit stellt keinen Bezug zum materiellen Aufwand dar.

Güte der Ergebnisse – Unter Güte der Ergebnisse wird in dieser Arbeit verstanden, inwiefern das Resultat nützlich für eine aussagekräftige Schätzung ist. Ist das Resultat einer Methode z.B. zwei Bananen, lassen sich daraus keine sinnvollen Aussagen über die Methode machen. Dieses Kriterium ist aus Autorensicht sehr beträchtlich, da in dieser Seminararbeit hauptsächlich Methoden zur Aufwandsschätzung der Softwareentwicklung untersucht werden und es essenziell ist, deren Eignung zu prüfen.

4.2 Methoden zur Aufwandsschätzung der Softwareentwicklung

In diesem Kapitel werden vier ausgewählte Verfahren zur Aufwandschätzung der Softwareentwicklung vorgestellt und nach den oben genannten Kriterien bewertet, inwiefern diese für eine Schätzung des Aufwands der Fehlerbehebung geeignet sind. Die Wahl auf diese vier Techniken lässt sich damit begründen, dass ein breites Spektrum an unterschiedlichen Techniken abgedeckt werden sollte. Des Weiteren werden die ausgewählten Techniken häufig genutzt und gelten daher als qualifiziert. Ein weiteres wichtiges Kriterium war die Aktualität. Durch diese Auswahlfaktoren haben sich schlussendlich die im Folgenden vorgestellten Techniken als geeignet herauskristallisiert.

Cocomo II Das Cocomo (Constructive Cost Model) ist eine der populärsten algorithmischen Verfahren zur einfachen Aufwandschätzung von Software. Cocomo II ist die neue Version von Cocomo und wurde neukalibriert und upgedated, sodass die Ergebnisse exakter sind als noch in Cocomo [Boe00].

Diese Methode berechnet den Aufwand eines Softwareprojekts in Personenmonaten (PM). Wie auch in anderen Methoden werden als Bezugsgrößen die Lines of Code (LOC), welche zur gesamten Projektgröße (Size) saldiert werden. Die in Anhang A.1 dargestellten Kosteneinflussgrößen stellen dabei alle kostenrelevanten Faktoren dar, welche sich jeweils multiplikativ (Aufwandsmultiplikatoren, EM) oder exponentiell (Skalenfaktoren, SF) auf den Projektaufwand auswirken können. Des Weiteren können Unternehmen spezifische Anpassungen, des Cocomo m.H. Kalibrierungsfaktoren (A und B), vornehmen. Verfügt ein Unternehmen über historische Daten, können diese verwendet werden, ansonsten werden A und B auf die empirisch ermittelten Werte $A = 2,95$ und $B = 0,91$ gesetzt. Diese sind aber für diese Arbeit nicht weiter von Bedeutung. Das Cocomo II Grundmodell sieht wie folgt aus und ist sehr simpel gehalte [WG06]:

$$PM = A \cdot EM \cdot Size^{B+SF}$$

Laut Boehm [Boe00] ggibt es keine Garantie, dass Cocomo II allen Unternehmen repräsentative Ergebnisse liefert, da die zu benutzenden Daten sorgfältig gesammelt und auf Cocomo II angepasst werden müssen. Somit kann eine genaue Kostenschätzung nur aufgrund von Anpassung auf Cocomo II erfolgen. Deshalb kann das Bewertungskriterium „Genauigkeit“ in diesem Fall nicht genau bestimmt werden, da diese unternehmensspezifisch ausfällt.

Cocomo II ist im Allgemeinen sehr simpel und einfach gehalten und für jedes Unternehmen durchführbar. Voraussetzung ist jedoch, dass die benötigten Daten zuvor gesammelt und aufbereitet wurden.

Die Güte der Methode für eine Aufwandsschätzung der Fehlerbehebung ist eher unbrauchbar, da viele der Kosteneinflussgrößen, wie beispielsweise „Developed for reuse“ oder „Data base size“, nicht auf die Aufwandschätzung der Fehlerbehebung zutreffen. Jedoch könnte mit den brauchbaren Kosteneinflussfaktoren durchaus eine Schätzung durchgeführt werden. Ob diese dann sehr aussagekräftig ist, hängt dann von der Anzahl der brauchbaren Kosteneinflussfaktoren ab.

Function-Point-Verfahren Als ein populärer Vertreter von vergleichenden Methoden sei in diesem Abschnitt das Function-Point-Verfahren vorgestellt.

Beim Function-Point-Verfahren wird das Projekt in die vom System durchgeführten Transaktionen zerlegt. Jeder dieser Transaktionen wird dann eine Anzahl an Function-Points zugeordnet. Jede einzelne Transaktion wird dabei nach der für die Transaktion nötigen Aktivitäten sowie Komplexität (Anzahl der genutzten Datenwerte) bewertet. Die Function-Points für die einzelnen Aktivitäten einer Transaktion sind in Tabelle 1 dargestellt. Die Komplexität des gesamten Systems ergibt sich demnach aus der Summe aller Function-Points aller möglichen einzelnen Transaktionen.

Funktionstyp	Komplexität		
	Gering	Mittel	Hoch
Interne logische Datei (ILF)	7	10	15
Externe Datei (EIF)	5	7	10
Externe Eingabe (EI)	3	4	6
Externe Ausgabe (EO)	4	5	7
Externe Anfrage (EQ)	3	4	6

Tabelle 1: Function-Points für Transaktionen eines Programms [Rö08]

Je nach dem wie genau und detailliert man die Bewertung der einzelnen Transaktionen durchführt, kann die Function-Point-Methode sehr aufwendig sein. In der Praxis werden daher die Function-Points eines Projekts mit dem "Rapid" Verfahren angenähert [Poe12]. Bei diesem Vorgehen werden die einzelnen Transaktionen nicht mehr für sich bewertet, sondern generell für eine Eingabe 4 Function-Points (fp), eine Ausgabe 5 fp, eine Abfrage 4 fp, einem internen Datenbestand 7 fp und einem externen Datenbestand 5 fp festgelegt. Insbesondere für größere Projekte ist der dadurch entstehende Fehler, in Relation zu dem für eine detailliertere Bewertung benötigtem Aufwand, nicht so ausschlaggebend.

Die Ergebnisse dieses Verfahrens lassen sich prinzipiell als Komplexitätsmaß für das Projekt verstehen, um den dafür nötigen Aufwand dann z. B. in Personenmonaten anzugeben, können andere Verfahren wie z. B. das oben vorgestellte Cocomo-Verfahren genutzt werden, da die Functon-Point-Methode ursprünglich als Messgröße, um die Produktivität verschiedener Projekte bei IBM vergleichen zu können, entwickelt wurde. Daher auch die Klassifizierung als vergleichende Methode.

Prinzipiell kann man mithilfe dieser Methode feststellen, wie schwierig es sein könnte einen Fehler zu beheben, indem man die Function-Points der Transaktion, in der der Fehler auftritt, dafür als Größe heranzieht. Dies ist insbesondere dann sinnvoll, wenn man das Projekt im Vorfeld durch Function-Points bewertet hatte und daher noch auf diese Daten zurückgreifen kann. Wir sind allerdings der Meinung, dass man dieses Komplexitätsmaß nur als grobe Orientierung verwenden sollte, da es auch sein kann, dass der Fehler in einem komplexen Modul sehr eindeutig ist und schnell erkannt und behoben werden kann. Insbesondere bei Fehlern, die nur im Zusammenhang bestimmter Aktionen im System auftreten, kann sich die Fehlersuche allerdings auch sehr schwierig gestalten.

Delphi-Methode Die Delphi-Metode stellt eine besondere Art einer Expertenbefragung dar. Bei dieser Methode werden verschiedene Experten zu den zu schätzenden Arbeiten befragt. Sie geben zunächst unabhängig voneinander ihre Schätzung ab. Zentraler Punkt dieser Methode ist, dass die einzelnen Experten aufgrund der Schätzungen der anderen Experten ihre Schätzung anpassen können, bis sich ein Konsens über den Umfang der zu schätzenden Arbeit eingestellt hat.

Diese Methode ist sehr (Zeit-)aufwendig, da mehrere Experten ihre Schätzungen in mehreren Runden iterativ verbessern. Wir halten dies für das Schätzen des Aufwands der Fehlerbehebung für zu aufwendig.

Im Allgemeinen ist die Genauigkeit der Methode und die Güte der Ergebnisse - je nach Erfahrung der befragten Experten - sehr gut.

Im Kontext des Schätzens des Aufwands der Fehlerbehebung lässt die Güte der Ergebnisse allerdings zu wünschen übrig. Die Experten sind oft nicht in der Lage, eine fundierte Schätzung abzugeben. Wenn der Experte eine fundierte Schätzung über die mit der Fehlerbehebung verbundenen Arbeit abgeben kann, so sollte ihm der Ursprung des Fehlers bekannt sein. Allerdings ist das *finden* des Fehlers im Allgemeinen viel aufwendiger als das Beheben desselben. Daher finden wir, dass es sinnvoller ist, die Zeit, die man für die Delphi-Methode benötigt, direkt in die Fehlerbehebung zu stecken.

Walston-Felix-Methode An dieser Stelle wird noch kurz die Methode von Walston und Felix [WF77] erläutert, die als frühe Kennzahlmethode spätere Methoden in ihren grundlegenden Prinzipien stark beeinflusst hat. Als grundlegendes Prinzip der Methode gilt, dass man den Aufwand nach folgender Gleichung berechnet:

$$\text{Aufwand} = 5.2 \cdot \text{Lines of Code}^{0.91+I}$$

Die Werte 5.2 und 0.91 sind dabei empirisch aus vorangegangenen Projekten gewonnene Werte. Walston und Felix schlagen dabei 29 Einflussfaktoren vor, um den projektspezifischen Produktivitätsfaktor I zu ermitteln. Die vorgeschlagenen Einflussfaktoren sind im Anhang A.2 dargestellt. Walston und Felix gewichten diese Faktoren nach ihrem Einfluss auf die analysierten Projekte von IBM und ermitteln den Produktivitätsfaktor I dann nach der Formel $I = \sum_{i=1}^{29} W_i \cdot X_i$. Wobei W_i das ermittelte Gewicht dieses Faktors und $X_i \in (0, 1)$ der Einfluss dieser Größe auf den Aufwand für dieses Projekt ist (0= vereinfacht die Entwicklung, 1 = dieser Einfluss erschwert die Entwicklung).

Diese Methode legte einen wichtigen Grundstein für andere Methoden, wie beispielsweise für die oben vorgestellte Cocomo Methode. Da modernere Methoden wie Cocomo oft bessere Einschätzungen liefern, als die Methode von Walston und Felix [KP13], würden wir Cocomo der Methode von Walston und Felix vorziehen. Insbesondere im Zusammenhang mit der Fehlerbehebung, sehen wir allerdings, dass die Methode von Walston und Felix genauere und ausschlaggebendere Faktoren beinhaltet (z.B. "Complexity of program flow" oder "Use of design and code inspections"), sodass man die Cocomo Methode noch um einige Faktoren aus der Methode von Walston und Felix ergänzen könnte.

4.3 An den Bereich Big Data und Machine Learning angelehnte Methoden

Die obigen Methoden beziehen zur Verbesserung der Schätzung oft Daten von vorherigen Projekten mit ein. Daher verfolgt man auch die Möglichkeit, aus diesen Daten eine Schätzung mithilfe von Methoden aus dem Bereich Big-Data und Machine-learning abzuleiten. Insbesondere beim Aufwand für die Fehlerbehebung scheint dies vielversprechend, da vorherige Daten (also bereits behobene Fehler) sich viel stärker auf das aktuelle Projekt beziehen, als Daten zur Produktivität anderer, möglicherweise komplett unterschiedlicher Projekte.[1]

Wir stellen hier daher stellvertretend einen Ansatz von Zhang et. al. [ZGV13] aus diesem Bereich vor.
Als Ziel wollen sie nur vorhersagen, ob ein Bug verglichen mit anderen Bugs schnell behoben werden kann oder dies einen bestimmten Aufwand überschreitet. Dazu nutzen sie die von einem Bug-Tracking System erhobenen Daten:

- Person, die den Bug gemeldet hat
- Entwickler, der für das Beheben des Fehlers verantwortlich ist
- Schwere der Auswirkungen
- Priorität
- Fachliche Kategorie, in der der Fehler auftritt
- Beschreibung des Fehlers

Als Klassifizierungsalgorithmus dient der k-nearest-neighbour Algorithmus. Bei diesem Algorithmus werden im Prinzip zu einem bestimmten Fehler die k "nächsten" d.h. ähnlichsten Fehler aus den vergangenen Daten berechnet. Falls diese Fehler überwiegend schnell zu beheben waren, dann ergibt dies als Vorhersage, dass auch dieser Fehler schnell zu beheben sein wird.[2] Um die k "nächsten" Bug-Reports zu berechnen, ist es nötig, eine Distanzfunktion zu definieren. Oft wird dabei die Euklidische Distanz im N-Dimensionalen Raum genommen, wobei die N-Dimensionen den N verschiedenen Kategorien an gesammelten Daten ("features") entsprechen. Daher muss man für jede der vom Bug-Tracking System erfassten Eigenschaften eine Distanzmetrik definieren. Die Abbildung 3 zeigt die Definition der Distanzmetrik beispielhaft für die Kategorie "Priorität". Man kann für die anderen vom Bug-Tracking System gesammelten Daten ähnliche Distanzmetriken definieren. Für die Distanz zweier textueller Beschreibungen zweier Fehler, kann man verschiedene, im Bereich Big Data und Machine Learning diskutierten Metriken, wie z. B. die relative Worthäufigkeit, verwenden [Seb02].

Je nachdem ab welchem Aufwand ein Bug als schwierig gilt, erreicht das Modell von Zhang et al [ZGV13] eine Genauigkeit der Vorhersage von 65% bis 85% der betrachteten Fehler. Ein Nachteil ist allerdings, dass man nur in die Klassen

[1] Der Begriff "Projekt" nach DIN 69901 impliziert ja bereits die Einmaligkeit des Vorhaben

[2] [Kun16]. Wir gehen hier der Einfachheit wegen nicht auf die Wahl des k ein.

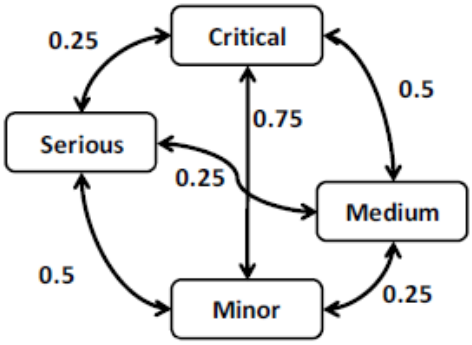

Abb. 3: Distanz zwischen den Bug-Prioritäten [ZGV13]]

"schwierig" und "relativ einfach" zu beheben einteilt. Für die Praxis ist diese Einteilung allerdings oft schon von Nutzen, um seine Ressourcen den Fehlern sinnvoll zuzuteilen.

Die Methode benötigt vergleichsweise wenig Aufwand, da man sie nur vollständig automatisiert benutzen muss. Der Erstellungsprozess der Methode – wie werden die einzelnen Distanzmetriken definiert - ist allerdings sehr aufwendig. Wenn man diesen Aufwand mit betrachtet, lohnt sich eine Schätzung mit Techniken aus dem Bereich Big Data und Machine Learning nicht, andernfalls können sie aber eine vergleichsweise gute Einschätzung liefern, sofern man bereits über möglichst viele Daten von vorhergehenden Fehlerbehebungen verfügt.

4.4 Relevanz der Einflussfaktoren bei der Aufwandsschätzung

In dem Bereich Big Data und Machine Learning gibt es Methoden, um aussagen zu können, welche Eigenschaften der analysierten Datensätze (Bugs) bezüglich des gefragten Kriteriums (Aufwand zur Behebung) von statistisch besonderer Bedeutung sind [Min89]. In der Abbildung 4 kann man sehen, dass von den in Abschnitt 4.3 zur Analyse herangezogenen Einflussfaktoren vor allem der Melder des Fehlers, sowie der mit der Behebung des Fehlers beauftragte Entwickler ausschlaggebende Faktoren sind. Das bedeutet für uns, dass die Qualifikation, der im Fehlerbehebungsprozess involvierten Personen, von *entscheidender Bedeutung* für einen möglichst effizienten Fehlerbehebungsprozess ist. Erstaunlicherweise hat die Schwere der Auswirkungen des Fehlers einen vergleichsweise geringen Einfluss. Wir vermuten, dass dies daran liegt, dass die Schwere der Auswirkungen eines Fehlers lediglich die Wahrscheinlichkeit diesen Fehler zu bemerken stark beeinflusst. Es bei der Behebung des Fehlers dann allerdings von viel entscheidenderer Bedeutung ist, dass eine möglichst qualifizierte Person eine gute Fehlerbeschreibung erstellt, um den Aufwand bei der Fehlersuche möglichst gering zu halten.

Auch die Methode von Walston und Felix sowie die Cocomo-Methode ziehen zur

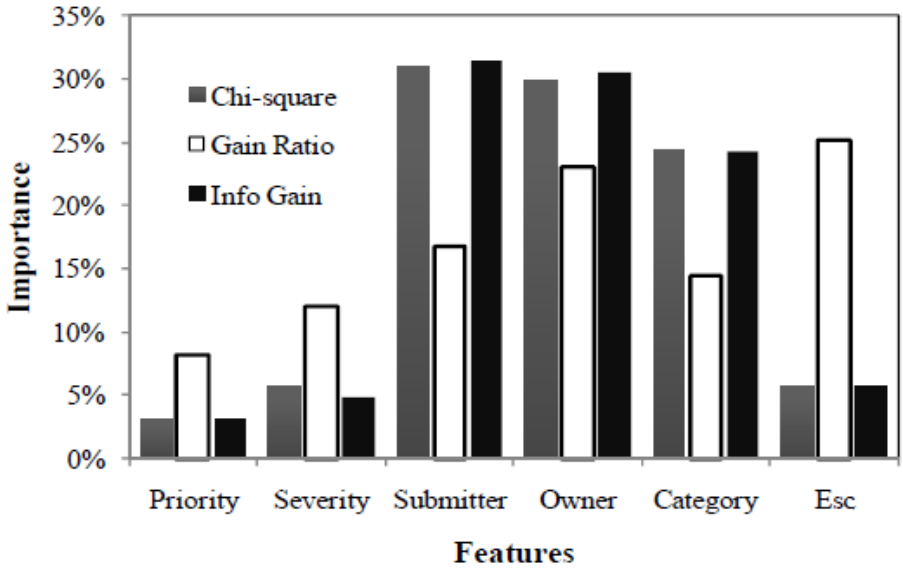

Abb. 4: Einfluss der verschiedenen Kategorien der für die Schätzung erhobenen Daten, nach 3 verschiedenen Metriken [ZGV13]

Aufwandsschätzung die Qualifikation des Entwicklungsteams als Faktor heran. Viele der anderen Faktoren sind unserer Meinung nach allerdings für den Aufwand der Fehlerbehebung nicht so stark ausschlaggebend, da es oft viel aufwendiger ist, den Fehler erst mal zu finden und zu verstehen, als ihn zu beheben. Die Fähigkeit einen Fehler zu finden wird zwar durch andere Faktoren, wie den Support von Tools, unterstützt, allerdings ist die Qualifikation derjenigen Entwickler, die den Fehler finden und beheben von so dominierender Bedeutung, dass es schwerfällt, Aussagen darüber zu treffen welche der anderen "Unterstützungsfaktoren" noch besonders wichtig sind.

5 Fazit

Die Arbeit hat gezeigt, dass das Schätzen des Aufwands von Fehlerbehebungen mit einigen Schwierigkeiten verbunden ist und man nicht unbedingt immer eine akkurate Schätzung erwarten sollte.

Wir halten eine Expertenbefragung für eine ungeeignete Methode, da die Experten ohne genaue Kenntnis des Fehlers keine fundierte Schätzung abgeben können und das Verstehen des Fehlers oft den wesentlichen Teil des Aufwands für die Fehlerbehebung einnimmt. Insbesondere dann, wenn der Experte nicht der Entwickler ist, der den vermutlich betroffenen Code geschrieben hat, kann er kaum eine fundierte Schätzung abgeben.

Auch das Verwenden der Cocomo-Methode ist nicht immer zu empfehlen. Insbesondere ist es nicht möglich a priori die Größe des Fehlers (z. B. in anzupassenden Lines of Code) anzugeben. Diese Größe abzuschätzen muss aber beim Schätzen des Aufwandes für die Fehlerbehebung Teil eines Verfahrens sein.

Die Ergebnisse einer Schätzung mit der Function-Point Methode kann als grundlegende Einordnung dienen, wie komplex das fehlerhafte Modul ist. Man kann dies als groben Richtwert bei einer Schätzung des Aufwands mit Cocomo verwenden. Allerdings ist der Aufwand für die Fehlerbehebung nicht mit dem für eine Neuentwicklung des Moduls zu vergleichen.

Methoden aus dem Bereich Big Data und Machine Learning bieten einen anderen interessanten Ansatz, eine grobe Einteilung der Fehler in "einfache" und aufwendig zu behebende Fehler vorzunehmen. Sie lassen sich allerdings nur dann anwenden, wenn möglichst viele Daten von vorangegangenen Fehlerbehebungen vorhanden sind.

Methode	Aufwand	Geauigkeit	Güte der Ergebnisse
Cocomo II	einfach	unternehmens-spezifisch	je nach Einflussfaktoren
Walston-Felix	einfach	schlechter als Cocomo II	je nach Einflussfaktoren
Function-Points	einfach	relativ genau	als grobe Einschätzung
Delphi	sehr hoch	sehr genau möglich	nur wenig fundierte Schätzung möglich
Zhang et. al.	komplett automatisiert	bereits ziemlich genau	nur 2 Klassen

Tabelle 2: Zusammenfassung des Vergleichs der vorgestellten Methoden

Es ist schwierig, Aussagen darüber zu treffen, welche Klassen von Bugs besonders einfach bzw. schwierig zu beheben sind.
Allerdings ist die Qualifikation, der am Fehlerbehebungsprozess beteiligten Personen von entscheidender Bedeutung, um den Aufwand für die Fehlerbehebung so gering wie möglich zu halten. Aus unserer Sicht ist es daher von wichtiger Bedeutung die Verfahren, um Bugs zu finden und zu verstehen, weiter zu verbessern und in der Ausbildung stärker hervorzuheben. Qualifikation bezieht sich dabei nicht nur auf die Entwickler, die den Fehler beheben, sondern insbesondere auch auf die, die den ursprünglichen Code geschrieben haben. Die Qualifikation, gut verständlichen Code schreiben zu können, hat eine sehr hohe Bedeutung, da gut verständlicher Code es ermöglicht, dass die Fehler schneller gefunden und verstanden werden können.

Literatur

AG. Novalnet AG. E-commerce-lexikon: Bug. http://www.novalnet.de/ecommerce-lexikon/bug [zugriff 25.06.17].

Boe00. Barry Boehm. Safe and simple software cost analysis. *IEEE software*, 17(5):14–17, 2000.

Boh14. Johannes Bohnet. Warum software oft so viele macken hat. http://www.focus.de/finanzen/experten/johannes_bohnet/toedliche-pannen-warum-software-oft-so-viele-macken-hat_id_4098533.html [zugriff 17.04.17], 2014.

ITw. ITwissen.info. Programmfehler. http://www.itwissen.info/Programmfehler-bug.html [zugriff 25.06.17].

Klo05. Karlhorst Klotz. Der traum von software ohne bugs. http://www.spiegel.de/netzwelt/tech/fehler-im-system-der-traum-von-software-ohne-bugs-a-363056.html [zugriff 17.04.17], 2005.

KP13. Sweta Kumari and Shashank Pushkar. Performance analysis of the software cost estimation methods: a review. *International Journal of Advanced Research in Computer Science and Software Engineering*, 3(7), 2013.

Kun16. Julian M. Kunkel. Lecture bigdata analytics: Machine learning. https://wr.informatik.uni-hamburg.de/_media/teaching/wintersemester_2016_2017/bd-06.pdf [zugriff 15.05.17], 2016.

Min89. John Mingers. An empirical comparison of selection measures for decision-tree induction. *Machine learning*, 3:319–342, 1989.

Poe12. Benjamin Poensgen. *Function-Point-Analyse. Ein Praxishandbuch.* dpunkt.verlag GmbH, 2012.

Rö08. Holger Röder. Methoden und werkzeuge des software engineering - kostenschätzung am beispiel. http://www.iste.uni-stuttgart.de/fileadmin/user_upload/iste/se/teaching/courses/muw/materialien/MuW2008-HR-Kostenschaetzung.pdf [zugriff 15.05.17], 2008.

Seb02. Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM computing surveys (CSUR)*, 34(1):1–47, 2002.

SKP14. Ripon K Saha, Sarfraz Khurshid, and Dewayne E Perry. An empirical study of long lived bugs. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 144–153. IEEE, 2014.

TLL+14. Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. Bug characteristics in open source software. *Empirical Software Engineering*, 19(6):1665–1705, 2014.

WF77. Claude E. Walston and Charles P. Felix. A method of programming measurement and estimation. *IBM Systems Journal*, 16(1):54–73, 1977.

WG06. Alexander Wehrmann and Daniel Gull. Ein cocomo-basierter ansatz zur entscheidungsunterstützung beim offshoring von softwareentwicklungsprojekten. *Wirtschaftsinformatik*, 48(6):407–417, 2006.

ZGV13. Hongyu Zhang, Liang Gong, and Steve Versteeg. Predicting bug-fixing time: an empirical study of commercial software projects. In *Proceedings of the 2013 international conference on software engineering*, pages 1042–1051. IEEE Press, 2013.

A Anhang

Im Anhang werden die von der Cocomo II Methode, sowie der Methode von Walston und Felix verwendeten Kosteneinflussfaktoren aufgelistet.

A.1 Einflussfaktoren der Cocomo II Methode

Kosteneinflussgrößen	Aufwandsmultiplikatoren/Skalenfaktoren
Development flexibility	1.26
Team cohesion	1.29
Developed for reuse	1.31
Precedentedness	1.33
Architecture and risk resolution	1.39
Platform experience	1.40
Data base size	1.42
Requiered development schedule	1.43
Language & tools experience	1.43
Process maturity	1.43
Storage constraint	1.46
Platform volatility	1.49
Use of software tools	1.50
Applications experience	1.51
Personnel continuity	1.51
Documentation match to lifecycle needs	1.52
Multisite development	1.53
Required software reliability	1.54
Time constraint	1.63
Product complexity	2.38
Personnel/team capability	3.53

A.2 Einflussfaktoren der Methode von Walston und Felix

- Customer interface complexity
- Use of design and code inspections
- User participation in requirements definition
- Use of top-down development
- Customer-originated program design changes
- Use of a chief programmer team
- Customer experience with the application area
- Overall complexity of code
- Overall personnel experience
- Complexity of application processing
- Percentage of development programmers who participated in the design of functional specifications
- Complexity of program flow
- Previous experience with the operational computer

- Overall constraints on program's design
- Previous experience with the programming language
- Design constraints on the program's main storage
- Previous experience with applications of similar size and complexity
- Design constraints on the program's timing
- Ratio of average staff size to project duration (people per month)
- Code for real-time or interactive operation or for execution under severe time constraints
- Hardware under concurrent development
- Percentage of code for delivery
- Access to development computer open under special request
- Code classified as nonmathematical application and input/output formatting programs
- Access to development computer closed
- Classified security environment for computer and at least 25% of programs and data
- Number of pages of delivered documentation per 1000 lines of code
- Use of structured programming
- Design constraints on the program's main storage
- Number of classes of items in the database per 1000 lines of code