Einstiegspunkte für Design und Codierung bei einer Wartungsaufgabe oder Fehlermeldung

Universität Hamburg
Fakultät für Mathematik,
Informatik und Naturwissenschaften
Department Informatik

Abstract. Bei der Entwicklung und Wartung von Software spielt die Identifikation des Nutzen von Programmabschnitten eine große Rolle, um entsprechende Funktionen einer Software zu ändern oder zu reparieren. Diese *Feature Locations* stellen einen Einstigespunkt in den Quelltext für Design- oder Code-Änderungen dar und können mithilfe verschiedener Technologien ermittelt werden. In dieser Seminararbeit stellen wir unterschiedliche Technologien und deren Verfahren vor und geben einen Überblick über geeignete Nutzungsfelder.

1 Einleitung

In Bereich der Softwareentwicklung und der Wartung von Software kommt es immer wieder zu Änderungen im Code, die eine bestimmte Funktionalität verändern oder verbessern sollen. Das Finden dieser zu verändernden Code-Segmente ist insbesondere bei großen Projekten alles andere als trivial. Schlecht bis gar nicht dokumentierter Code und missachtete oder falsch umgesetzte Design und Architektur-Patterns stehen dabei an der Tagesordnung. Um dennoch möglichst zuverlässig die Abschnitte im Quelltext lokalisieren zu können, die zu einer bestimmten Funktionalität gehören, werden Techniken der sogenannten *Feature Location* angewendet.

Diese erlauben es dem Nutzer, nach bestimmten Mustern im Quelltext zu suchen und so einen geeigneten Einstiegspunkt in den Code zu finden. Dieser kann sowohl für eine Änderung des Designs oder des Codes im Sinne einer Wartungsaufgabe oder eines Change-Requests erfolgen, als auch bei einer Fehlermeldung schnell für einen Überblick der betroffenen Code-Abschnitte sorgen.

Es werden zwischen statischen und dynamischen Analysetechniken unterschieden, die einen Quelltext vor oder während der Laufzeit analysieren können. In den folgenden Abschnitten dieser Seminararbeit werden zunächst wichtige Begriffe definiert, dann verschiedene Analyseverfahren erläutert und anschließend Technologien vorgestellt, die mit Hilfe dieser Verfahren Features im Code lokalisieren.

Ziel dieser Seminararbeit ist es, die verschiedenen Analyseverfahren und Technologien zur Erkennung von Features im Code zu beschreiben und zu vergleichen.

Es wird weiterhin eine Empfehlungen gegeben, wann welche Techniken angewendet werden sollten.

2 Begriffe

Für die Vorstellung der Analyseverfahren sollen zunächst der Begriff des *Features* sowie der *Feature Location* definiert und erklärt werden.

Feature

Ein Feature ist ein Software Artefakt, das eine spezifische Funktionalität implementiert [1]. Diese Funktionalität wird in natürlicher Sprache beschrieben und wird von einem Programmabschnitt wiedergespiegelt. Ein Feature besteht üblicherweise aus einen Namen, einer Bedeutung (Intension) und einer Erweiterung (Extension) [2].

Feature Location

Der Prozess der Feature Location beschreibt die Identifikation der Beziehung zwischen Features und deren Implementierung. Dabei liegt die Beschreibung des Features in natürlicher Sprache vor, die dann einem entsprechenden Codeabschnitt zugeordnet werden soll (Mapping) [3].

3 Analyseverfahren

In diesem Abschnitt sollen verschiedene Analyseverfahren vorgestellt werden, die das Finden von Feature Locations ermöglichen.

Program Dependence Analysis (PDA)

Die Analyse eines Programms auf interne Abhängigkeiten wird Abhängigkeitsanalyse (engl. Dependence Analysis) genannt. Sie umfasst in der Regel Kontrollflussabhängigkeiten und Datenabhängigkeiten innerhalb eines Programmes. Diese werden zur Übersetzungszeit mithilfe des Compilers festgestellt. Ziel dieser Analyse ist es zum Beispiel, zu überprüfen, ob ein Programm parallelisiert ausgeführt werden kann. Für die Analyse von Features im Quelltext ist diese Methode hilfreich, da durch sowohl Kontrollflussabhängigkeiten, als auch Datenabhängigkeiten auch semantische Verknüpfungen verschiedener Abschnitte des Codes hergestellt werden können [4].

Trace Analysis

Bei der Trace Analysis findet eine Auswertung der Traces (Spuren) statt. Das Erstellen von Spuren ist eine Form des Loggings bei der das Verhalten eines Programms aufgezeichnet wird. Traces unterscheiden sich vom Logging dadurch, dass sie eher von Entwicklern als Administratoren verwendet werden und deutlich weniger abstrahieren sondern eine Menge Ausgabedaten auf niedrigen Ausführungsebenen liefern. Meistens werden mehrere Spuren verwendet, wobei es eine Spur sämtliche wichtigen Ereignisse aufzeichnet und die anderen diese komplexe, quantitativ auffallende Spur nach bestimmten Kriterien filtern.

Latent Semantic Indexing (LSI)

Das Verfahren des Latent Semantix Indexing wird zum Indexieren von Abschnitten und Mustern eines Textes mithilfe von Singulärwertzerlegung der Ausdruck-Dokument-Matrix verwendet. Die Ausdruck-Dokument-Matrix (engl. Document-Term-Matrix) stellt die Frequenz der Ausdrücke innerhalb eines Dokumentes oder Textes dar. Die Singulärwertzerlegung dieser Matrix ist die Darstellung der ursprünglichen Matrix als Produkt dreier spezieller Matrizen, deren Singulärwerte auf bestimmte Eigenschaften der Matrix schließen lassen. LSI geht davon aus, dass Wörter, die in einem bestimmten Kontext verwendet werden, eine ähnliche Bedeutung haben. Dieses Verfahren lässt sich von natürlicher Sprache auch auf Quellcode übertragen, insbesondere auf Kommentare innerhalb des Codes [5].

3.1 Statische Feature Location Technologien

Im folgenden Abschnitt sollen statische Technologien zum Finden von Features im Quelltext vorgestellt werden.

Statische Analyse

Die Analyse von Quelltext zu einem Zeitpunkt, bei dem das Programm kompiliert wird und noch nicht ausgeführt wurde, wird statische Analyse genannt. Hierbei werden mithilfe von zum Beispeil Datenfluss-Analyse und Kontrollgraphen alle Abhängigkeiten und Funktionsaufrufe innerhalb des Codes analysiert und es können unter anderem Fehler wie zum Beispiel Race-Conditions oder Buffer-Overflows identifiziert werden. Dieser Prozess wird häufig von automatisierten Tools durchgeführt [6].

Technologie Beispiele - Plain Output

Eine Technologie von Shepherd, Fry und Vijay-Shanker basiert auf der Analyse natürlicher Sprache in objektorientierten Programmen, wie zum Beispiel Methodennamen, Variablennamen oder Kommentare [7]. Dabei wird davon ausgegangen, dass Verben für Methodennamen stehen und Substantive für Objektnamen.

Das Ergebnis der Programmanalyse ist ein *action-oriented identifier graph model (AOIG)*, wobei eine *action* zum Beispiel ein Verb bzw. Methode sein kann, das auf einem Substantiv bzw. Objekt ausgeführt wird. In einem AOIG können vier verschiedene Knotentypen auftreten: *verb nodes, direct-object nodes (DO), verb-DO nodes* und *use nodes*. Die *verb nodes* entsprechen den Methoden in einem Programm, die *direct-object nodes* den Objekten, die *verb-DO nodes* einem Paar aus Methode und Objekt, die zusammen gehören und die *used nodes* jeweils einem verb-DO Paar, das in einem Kommentar oder im Quelltext auftritt.

Ein AOIG hat weiterhin zwei unterschiedliche Kantentypen: die *pairing edges* und *use edges*, wobei erstere je ein Verb oder DO Knoten mit einem entsprechenden verb-DO Paar verbinden und letztere ein verb-DO Paar mit allen anderen Knoten verbindet, die es benutzt.

Der Input des Algorithmus ist zunächst eine abstrakte Query, die dann in eine vom Nutzer formulierte Query als Menge von Verb-DO Paaren umformuliert wird. Diese Query soll etwaige interessante Features im Programm beschreiben, indem ausschließlich Verben und DOs aufgelistet werden. Diese werden anschließend vom Algorithmus um von den Wörtern abstammende Wörter erweitert, das sogenannte *stemming*. Im nächsten Schritt werden dem Nutzer Begriffe vorgeschlagen, die sich ähnlich sind. Solche Begriffe können zum Beispiel Synonyme oder umgangssprachliche Bedeutungen von Wörtern sein. Der Nutzer kann hier aus einer Liste von maximal zehn vorgeschlagener Wörter aussuchen, welche zusammen mit ihren abstammenden Wörtern der Query hinzugefügt werden. Ein detaillierter Ablauf des Algorithmus ist in Tabelle 3.1 beschrieben.

Das Erweitern der Query wird so lange vom Nutzer fortgeführt, bis dieser mit der Zusammensetzung zufrieden ist. Anschließend wird der AOIG nach allen Verb-DO Paaren abgesucht, die Wörter der Query enthalten. Es werden alle Methoden extrahiert, bei denen die gefundenen Paare verwendet werden. Zwischen diesen Methoden werden dann die Aufruf-Beziehungen untereinander mittels PDA analysiert (siehe Abschnitt 3).

Als Ergebnis der Suche wird ein Ergebnis-Graph generiert. Die Knoten des Graphen repräsentieren alle gefundenen Methoden und die Kanten deren strukturellen Beziehungen untereinander.

Mit diesem Graphen können gesuchte Features anhand der Methoden lokalisiert werden. Durch die strukturellen Beziehungen zu anderen Methoden kann zusätzlich die Einbindung einer Funktion in den Quellcode erfasst werden, was das Verändern oder Warten dieser Funktionalität vereinfacht.

Schritt	Zusammensetzung der Query
1	Abstrakte Suchquery, zB. 'Suche automatisch nach Entitäten'
2	Konkrete Suchquery, zB. 'Suche Entitäten'
3	Aufgeteilte Suchquery, zB. 'Verb: suchen' 'Objekt: Entität'
4	Erste erweiterte Suchquery, zB. 'Verb: suchen, gesucht, gesuchte' 'Objekt: 'die Entität, alle Entitäten, erste Entität'
5	Vorgeschlagene Erweiterungen der Suchquery, zB. 'Verb: absuchen, durchsuchen, aufspüren, stöbern, ...' *Nutzer wählt aus* 'Objekt: 'Instanz, Objekt, Symbol, ...' *Nutzer wählt aus*
6	Wiederholung von Schritt 5 bis die Query ausreichend erweitert wurde

Tabelle 1. Zusammensetzung der Such-Query für den AOIG

Technologie Beispiele - Guided Output

Andrian Marcus hat unter anderem zusammen mit Andrey Sergeyev, Václav Rajlich und Jonathan I. Maletic als einer der ersten einen Ansatz für Feature Location vorgestellt, der auf *Information Retrieval (IR)* basiert [8][9]. Als Basis dienen Text-Dokumente, die bestimmte Softwareelemente wie zum Beispiel Methoden oder Datentypen beschreiben. Diese Dokumente werden domänenspezifisch durch Identifier (Methodennamen, Objektnamen, Klassennamen, etc.) und Kommentare erstellt. Hierbei werden die Identifier durch bekannte Code-Styles voneinander differenziert. Ein solcher Code-Style kann zum Beispiel das Teilen von Wörtern durch einen Unterstrich '_' oder das Kleinschreiben von Methodennamen, bei denen die Folgewörter jedoch groß geschrieben werden (zum Beispiel *'updateMessageAndSendToReceiver'*), sein. Jedes Softwareelement wird dabei von einem Dokument beschrieben und mittels der Identifier in LSI Vektoren transformiert.

Der Nutzer kann in natürlicher Sprache nach Features suchen. Bei jeder Suche wird anhand der Identifiers im Quelltext und der vom Nutzer erstellten Phrasen zum Beschreiben der Features Dokumente im LSI Raum erstellt. Diese Dokumente werden mit Hilfe von Ähnlichkeitsberechnungen zwischen der Query und den Dokumenten differenziert. Die Dokumente, die der Query am ähnlichsten sind, werden als Ergebnis der Suche zurückgegeben.

Im folgenden Schritt kann der Nutzer die ihm präsentierten Ergebnisse bewerten und deren Tauglichkeit bestimmen. Falls weitere Dokumente gefunden werden, die in den Kontext der Suche passen, werden diese der Query hinzugefügt. Erst wenn keine weiteren relevanten Dokumente mehr gefunden werden, stoppt der Algorithmus. Das Ergebnis ist dann eine Menge von Dokumenten, die vom Nutzer als relevant angesehen werden und nach der Ähnlichkeitsberechnung des Algorithmus geordnet sind.

Ein anderer Ansatz von Peng Shao erweitert den von Marcus et al. ([8][9]), indem das LSI Ranking mit einem Ranking durch einen statischen Kontrollflußgraphen kombiniert wird [10].

Wie bereits oben beschrieben wird jede Methode des Programms durch ein Dokument mit den entsprechenden Identifiern beschrieben. Durch den LSI Algorithmus wird unter Berücksichtigung der Eingabe-Query das Ranking für jedes Dokument berechnet. Danach wird eine Menge der Methoden aus Dokumenten erstellt, die einen höheren Wert als ein bestimmter empirisch ermittelter Grenzwert im Ranking erreichen. Von diesen Methoden wird dann eine Menge der *callers* und *callees* erstellt.

Die Methoden werden anhand der LSI Werte und der Werte des Rankings des Kontrollflußgraphen absteigend ausgegeben. Somit kombiniert dieser Ansatz beide Rankings und erhöht dadurch die Genauigkeit der Ausgabe.

Durch die Kombination beider Verfahren können Features anhand der Dokumente identifiziert und deren Wirkungsweise mit Hilfe des Kontrollflußgraphen genauer erkannt werden.

3.2 Dynamische Feature Location Technologien

Dynamische Analyse

Hauptmerkmal der dynamischen Analyse ist ihre Durchführung während der Laufzeit eines Programms. Das bedeutet, die Ausführung des Programms ist zwingend erforderlich und während der Laufzeit werden Informationen gesammelt, um wahrscheinliche Features zu lokalisieren. Dabei können ausschließlich funktionale Features gefunden werden, da die dynamische Analysetechnik an eine eingeschränkte Sicht auf das Programm gebunden ist, die der Sicht des Anwenders entspricht.

Ähnlich wie bei der statischen Analyse unterscheidet man zwischen Plain-Output-Techniken, bei denen mögliche, zu einem Feature gehörende, Quellcodebestandteile ungeordnet zurückgeliefert werden und Guided-Output-Techniken, bei denen die ausgegebenen Komponenten durch zusätzliche Informationen in Zusammenhang gebracht werden und ihr Kontext genauer erläutert wird.

Technologie Beispiel - Plain Output

In ihrer wissenschaftlichen Arbeit Location Program Features by using Execution Slices benutzen die Autoren W. Eric Wong, Swapna S. Gokhale, Joseph R. Horgan und Kishor S. Trivedi ein dynamisches Plain-Output-Verfahren zur Analyse von Code [11]. Sie verwenden dabei Execution Slicing im Gegensatz zum Static Slicing. Beide Techniken werden unter Program Slicing zusammengefasst.

Slices können verschiedene Bestandteile eines Programms sein wie z.B. Code-Blöcke oder Kontrollstrukturen. Auch sogenannte c- und p-uses können Bestandteile einer Slice sein.

C- und p - uses beschreiben beide den Pfad einer Variable im Programmfluss ohne, dass die Variable verändert wird, wobei c-uses Variablen zur Berechnung dienen und p-uses-Variablen für eine Prädikat oder eine Entscheidung verwendet werden.

Faktoren wie Heuristiken, Test Cases, Detailgenauigkeit und Tool-Unterstützung sind ausschlaggebend für den Erfolg einer dynamischen Analysetechnik. Besonderem Augenmerk gilt dabei der Menge der Tests, die dem Programm als Eingabe übergeben werden. Die Autoren heben hervor, wie wichtig die richtigen Testmengen sind und unterscheiden zwischen aufrufenden (invoking) Tests, die sich explizit auf ein bestimmtes Feature und seine Funktionalität konzentrieren und ausschließenden Tests, die alle anderen Test-Sets enthalten, die nicht auf das Feature fokussiert sind. Darunter sind auch die zu finden, die das gesuchte Feature und weitere Komponenten umfassen, d.h. Tests, bei denen das Feature nur ein Bestandteil von vielen ist.

Eine für die Autoren erfolgreiche Herangehensweise war zuerst die Menge aller aufrufenden Tests zu bilden, von denen einige auch die Funktionalität anderer Features testen. Dann bildet man die Menge aller ausschließenden Tests, wobei diese jedoch auch Abschnitte des Quellcodes enthalten können, die vom gesuchten Feature benutzt werden. Die Differenz der beiden ist die Teilmenge, die zur Lokalisierung des gesuchten Features genutzt wird.

Die aufrufenden Tests sollen sich dabei sehr stark unterscheiden, während die ausschließenden Tests sehr ähnlich sein sollen. Ohne Tool-Unterstützung wäre es sehr Aufwending, Slices, die wie erwähnt aus Code-Blöcken, Kontrollstrukturen, c- oder p-uses bestehen, für jeden einzelnen Testfall zu sammeln. Die Autoren verwendeten deshalb xVue. Dieses Programm zählt für jeden Test, wie oft die angegebenen execution slices verwendet wurden und ordnet dadurch jedem Test die dazugehörigen Slices zu. Weiterhin teilt es die Tests, sofern das noch nicht geschehen ist, in aufrufende und ausschließenden Tests.

Die Autoren erläuterten das Verfahren an einer Fallstudie mit der wissenschaftlichen Software SHARPE, die aus 35 412 Zeilen C-Code besteht. Sie fokussierten sich auf 5 Features und sammelten aufrufende und ausschließende Tests mit dem heuristischen Verfahren, das bereits beschrieben wurde.

xVue lieferte im Bezug auf Code-Blöcke und Kontrollstrukturen als Slices gute Ergebnisse, was die Lokalisierung der Features im Code betraf. Die dann zurückgelieferten Ergebnisse wurden von Experten bezüglich SHARPE verifiziert. Die Zuordnung von p- und c-uses ließ sich nicht vollständig verifizieren, da die Software derart komplex ist, das selbst die Experten nicht sämtliche Ergebnisse nachvollziehen konnten.

Nach der Auswertung der Studie kamen die Autoren zu dem Schluss, dass die auf execution slices basierende Methode als ein Einstiegspunkt für Feature Location genutzt werden kann und stellten ihre Eignung dieser Analysetechnik besonders für komplexe System fest, in denen sich implementierte Features über mehrere Module erstrecken. Dennoch steht und fällt das Verfahren mit den Test-Mengen, mit denen das Programm gestartet wird und die Autoren stellten die Vermutung auf, dass auch in ihrer Fallstudie nicht der gesamte relevante Code bezüglich eines Features gefunden wurde. Auch wenn die Verifizierung der c- und p-uses nicht vollständig durchgeführt werden konnte, waren die SHARPE-Experten doch überrascht über die Detailgenauigkeit, mit der xVue Features lokalisiert hatte.

Die Autoren hoben auch die einfache Anwendung des Verfahrens hervor, denn die Berechnung der execution slices war ja durch das Tool vereinfacht und automatisiert worden. Einzig allein die aufrufenden und ausschließenden Tests mussten durch den Anwender definiert werden. Die Identifizierung von c- und p-uses bot den weiteren Vorteil, dass diese Information zur Code Coverage genutzt werden konnte. Das heuristische Verfahren zur Ermittlung der Test-Mengen, welches sie genutzt hatten, bewerteten sie als nützlich für den untersuchten Code, doch generell müssen die Heuristiken jeweils an die Art der Implementation angepasst werden.

Die Autoren kommen zu dem Schluss, dass sich ihr Verfahren durchaus für die Lokalisierung von relevantem Code bezüglich eines Features eignet. Es ist besonders nützlich, um in großen Code-Bereichen die relevanten Zeilen zu finden, die ein solches Feature implementieren. Dennoch hat das Verfahren die typische Schwäche der dynamischen Analyse: es ist schwierig, alle Komponenten im Code, die einem Feature zuzuordnen sind, zu entdecken und die Wahrscheinlichkeit ist groß, dass einige Komponenten nicht entdeckt werden. Der große Vorteil der Methode besteht in der Tool-Unterstützung, wodurch sie sehr einfach durchzuführen ist.

Technologie Beispiel - Guided Output

In dem Paper Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace von Dapeng Liu, Andrian Marcus, Denys Poshyvanyk und Vaclav Rajlich werden die Techniken LSI und Execution-Trace-Analyse zusammen genutzt.

3.3 Textuelle Feature Location Technologien

Textuelle Suche

Die Textuelle Suche nach Features im Quellcode eines Programms ist eine der ersten Ansätze der *Feature Location*. Hierbei wird der Code als ein zusammenhängender Text oder in Form von mehreren Textdateien oder -abschnitten eingelesen und auf bestimmte Suchmuster überprüft. Meist findet bei der textuellen Suche eine Erweiterung der Sucheingabe statt, zum Beispiel in Form des in Abschnitt 3.1 beschriebenen *stemmings* oder durch die Verwendung von regulären Ausdrücken [12].

Technologie Beispiel - Textuelle Suche

Bei einem geringen Umfang des Programmcodes können für die Suche innerhalb des Codes einfache Tools wie zum Beispiel das Linux/Unix-Kommando *grep* verwendet werden, das gegebene Texte oder Textdateien nach bestimmten eingegebenen Mustern absucht und diese ausgibt. Tools dieser Art können häufig mit regulären Ausdrücken verwendet werden, die die textuelle Suche erheblich mächtiger und vielseitiger machen.

Bei einer größeren Menge an Lines of Code (LOC) eines Programms werden üblicherweise Methoden des *Indexing and Searching* angewandt. Ein Tools, das mit dieser Methode arbeitet, ist das Feature Location And Textual Tracing Tool (FLOAT^3) [13]. Gesucht wird üblicherweise mit einer Suchquery. Die Indexierung des Codes besteht aus dem Erstellen von sogenannten Dokumenten für jede Methode oder jedes Feld im Code, das alle Wörter enthält, die in der Methode oder dem Datenfeld verwendet werden. Hierbei werden üblicherweise Füll- und Stoppwörter wie 'the' oder 'a' automatisch entfernt. Zusätzlich wird das *stemming* auf die Wörter in den Dokumenten angewandt und zusammengesetzte Wörter wie zum Beispiel 'updateMessageAndSendToReceiver' werden in die einzelnen Bestandteile ohne Füllwörter zerlegt, in diesem Fall in 'update', 'message', 'send' und 'receiver'.

Die Query und jedes Dokument werden dann in einen Vektor konvertiert. Bei einer Suche wird dann der Query-Vektor mit den verschiedenen Dokumenten-Vektoren auf ihre Ähnlichkeit verglichen und ein Score bestimmt. Die Dokumente mit dem höchsten Score werden dann als bestes Suchergebnis vorgeschlagen.

Mit diesem Verfahren können auch große Mengen an LOCs schnell und gut skalierbar durchsucht werden. Tools wie FLOAT^3 sind unter anderem auch als Plugin für verschiedene Entwicklungsumgebungen verfügbar und dadurch leicht zu integrieren.

4 Vergleich

Die statische Analyse untersucht den Quelltext vor der Ausführung und hat somit Zugriff auf sämtliche Bestandteile des Programms. Dadurch können Features jeglicher Art gefunden werden, seien sie nun funktional oder nicht nichtfunktional. Eine mehrmalige Wiederholung der statischen Analyse eines Programms ist idempotent und liefert immer wieder das gleiche Ergebnis.

Während also die statische Analyse einen größeren Feature-Raum bezüglich ihrer Quantität und Diversität untersuchen kann, ist die dynamische Analyse an das Programm zu Ausführung gebunden und damit auch an spezifische Eingabeparameter, die den Verlauf bestimmen. Somit kann das Auffinden von Features stark eingeschränkt sein und selbst ein gefundenes Feature lässt sich nur schwer generalisieren.

Betrachtet man Verfahren zu statischen und dynamischen Analyse als binären Klassifikator, dann liefern statische Verfahren mit größerer Wahrscheinlichkeit eine Menge false-positives. Diese resultieren nicht nur aus den quantitativ höheren Featureraum sondern auch aus der Feststellung, dass die Wahrscheinlichkeit eines gefundenen Features nie genau bestimmt, sondern nur approximiert werden kann. Statische Verfahren finden also mit größerer Wahrscheinlichkeit Features, die eigentlich gar keine sind, die sogenannte Überapproximiation.

Dynamische Verfahren dagegen, auf einen kleineren Feature-Raum beschränkt, liefern vermehrt false-negatives. Existierende Features werden in diesem Fall nicht als solche erkannt und falsch klassifiziert, was als Unterapproximation bezeichnet wird.

Worin sich dynamische und statische Verfahren weiterhin unterscheiden, ist die Vorarbeit, die bezüglich der Eingabedaten geleistet werden muss. Während statische Verfahren, wie bereits erwähnt, bei mehrmaliger Durchführung das gleiche Ergebnis liefern sollten, sind dynamische Verfahren nur so gut in der Identifizierung von Features im Quellcode wie die Eingabeparameter, mit denen das Programm gestartet wird. Diese müssen mit verschiedenen Heuristiken genau evaluiert werden, da eine ungenaue Testmenge in völlig unbrauchbare Ausgaben münden kann. Es ist in dieser Hinsicht aufwändiger als bei einer statischen Analyse und der Anwender, der die Testfälle auswählt, muss qualifiziert und erfahren sein.

5 Fazit

Das Lokalisieren von Features im Quellcode eines Programms stellt die Basis für Veränderungen oder Wartungsaufgaben an Programmen dar. Der Aufwand, die korrekten Abschnitte des Codes zu finden, kann unter Umständen erheblich sein. Dadurch wird die Wiederherstellung des Programmstatus in einen wartbaren und funktionierenden Zustand zunehmend erschwert oder verhindert.

Um diesem Problem entgegenwirken zu können, werden verschiedene Techniken zur *Feature Location* verwendet. Diese ermöglichen das statische oder dynamische Lokalisieren von Funktionen oder Methoden innerhalb des Codes.

Hierbei stehen dem Anwender eine vielzahl unterschiedlicher Technologien zur Verfügung, die sich nicht pauschal vergleichen lassen. Die Frage, ob ein statisches oder dynamisches Verfahren verwendet werden sollte, hängt unter anderem davon ab, ob das Programm überhaupt ausführbar ist. Weiterhin unterscheiden sich viele Technologien hinsichtlich des Nutzeraufwandes oder der Art und Weise, wie eine Suchquery oder ähliches aufgebaut und spezifiziert werden kann. Hierbei entsteht unter Umständen ein Trade-Off zwischen der Genauigkeit oder Relevanz des Outputs und dem Aufwand für den Nutzer.

Letztendlich sollte die zu verwendende Technologie je nach Bedarf und Anforderungen ausgewählt oder mit anderen kombiniert werden.

Literaturverzeichnis

[1] IEEE. Std. 829;. Available from: https://standards.ieee.org/findstds/standard/829-2008.html.

[2] Chen K, Rajlich V. Case Study of Feature Location Using Dependence Graph. Wayne State University Department of Computer Science;. Available from: http://www.cs.wayne.edu/~severe/publications/Chen.IWPC.2000.FeatureLocation.pdf.

[3] Rubin J, Chechik M. A Survey of Feature Location Techniques. University of Toronto, Department of Computer Science; 2013.

[4] Stafford J. A Formal, Language-Independent, and Compositional Approach to Interprocedural Control Dependence Analysis. University of Colorado;. Available from: ftp://ftp.sei.cmu.edu/pub/jas/Stafford-thesis.pdf.

[5] Deerwester S, Dumais S, Furnas G, Landauer T, Harshman R. Indexing by Latent Semantic Analysis. University of Colorado;. Available from: http://lsa.colorado.edu/papers/JASIS.lsi.90.pdf.

[6] Open Web Application Security Project (OWASP). Static Code Analysis. Open Web Application Security Project (OWASP);. Available from: https://www.owasp.org/index.php/Static_Code_Analysis.

[7] David Shepherd EHLP Zachary Fry, Vijay-Shanker K. Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns. University of Delaware;. Available from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.136.4512&rep=rep1&type=pdf.

[8] Marcus A. Semantic Driven Program Analysis. Wayne State University;. Available from: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1357837.

[9] Marcus A, Sergeyev A, Rajlich V, Maletic J. An Information Retrieval Approach to Concept Location in Source Code. Wayne State University, Kent State University;. Available from: https://pdfs.semanticscholar.org/a252/07e1b93fb57ba017c846c858f665dda0eecd.pdf.

[10] Peng Shao RS. Feature Location by IR Modules and Call Graph. University of Alabama;. Available from: http://delivery.acm.org/10.1145/1570000/1566539/a70-shao.pdf?ip=134.100.5.65&id=1566539&acc=ACTIVE%20SERVICE&key=2BA2C432AB83DA15%2EBB626F2563133BE7%2E4D4702B0C3E38B35%2E4D4702B0C3E38B35&CFID=939130543&CFTOKEN=24947363&__acm__=1495374196_a0b456fb5b027e3daa05c1d361fad2e4.

[11] Wong WE, Gokhale SS, und Kishor S Trivedi JRH. Location Program Features by using Execution Slices. University of California;. Available from: https://www.researchgate.net/profile/Kishor_Trivedi2/publication/3795664_Locating_Program_Features_by_using_Execution_Slices/links/0fcfd50dc6cac8f70e000000/Locating-Program-Features-by-using-Execution-Slices.pdf.

[12] William E Shotts, Jr . Grep Manual Page. www.linuxcommand.org;. Available from: http://linuxcommand.org/man_pages/grep1.html.

[13] Trevor Savage DP Meghan Revelle. FLAT^3: Feature Location and Textual Tracing Tool. College of William and Mary Department of Computer Science;.