Duale Hochschule Baden-Württemberg

Stuttgart Campus Horb

Untersuchung von Analysemöglichkeiten zur Identifikation ungenutzten Codes und Features

T3000 - Praxisarbeit

---

eingereicht von:	Moris Kotsch
Matrikelnummer:	1317681
Kurs:	TINF2018
Studiengang:	Informatik
Hochschule:	DHBW Stuttgart Campus Horb
Ausbildungsfirma:	ENISCO by FORCAM GmbH
Leitender Dozent:	Prof. Dr.-Ing. Olaf Herden
Betriebliche Ausbildungsleiterin:	Dipl.-Betriebsw. Angela Rasch
Betrieblicher Betreuer:	Dipl.-Ing. Florian Waldmann
Bearbeitungszeitraum:	21.12.2020 - 08.03.2021

Freudenstadt, 1. März 2021

Sperrvermerk

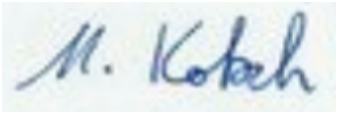
Die vorliegende Praxisarbeit zum Thema „Untersuchung von Analysemöglichkeiten zur Identifikation ungenutzten Codes und Features“ beinhaltet interne vertrauliche Informationen der Firma ENISCO by FORCAM GmbH. Die Weitergabe des Inhaltes der Arbeit und eventuell beiliegender Zeichnungen und Daten im Gesamten oder in Teilen ist untersagt. Es dürfen keinerlei Kopien oder Abschriften - auch in digitaler Form - gefertigt werden. Ausnahmen bedürfen der schriftlichen Genehmigung der Firma ENISCO by FORCAM GmbH.

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich:

1. dass ich meine Praxisarbeit mit dem Thema **„Untersuchung von Analysemöglichkeiten zur Identifikation ungenutzten Codes und Features“** ohne fremde Hilfe angefertigt habe;
2. dass ich die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe;
3. dass ich meine Praxisarbeit bei keiner anderen Prüfung vorgelegt habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.



Freudenstadt, 1. März 2021

Moris Kotsch

Zusammenfassung

Durch den stetigen Wachstum moderner Softwaresysteme wächst auch der Anteil von ungenutzten Code-Passagen und Features. Innerhalb dieser Praxisarbeit soll ein erster Einblick in die Analysemöglichkeiten für die Identifikation von ungenutztem Code aufgezeigt werden. Um solch eine Analyse durchzuführen, bedient man sich den Ergebnissen der beiden Analyse-Arten „statische Codeanalyse“ und „dynamische Codeanalyse“. Neben einer Erläuterung beider Analyse-Arten wird im Weiteren ein Konzept und die dazugehörige Umsetzung für die dynamische Codeanalyse, anhand des Enisco Manufacturing Execution System (kurz E-MES), der Firma Enisco by Forcam GmbH, aufgezeigt.

Abstract

Due to the constant growth of modern software systems, the share of unused code passages and features is also growing. Within this practice work a first view of the analysis possibilities for the identification of unused code is to be pointed out. In order to accomplish such an analysis, one uses the results of the two analysis kinds „static code analysis“ and „dynamic code analysis“. In addition to an explanation of both types of analysis, a concept and the associated implementation for dynamic code analysis will also be presented, based on the Enisco Manufacturing Execution System (E-MES for short) from the company Enisco by Forcam GmbH.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

1 Einleitung

1.1 Aufgabenstellung

Viele historisch gewachsene Systeme sammeln über die Jahre Code an, der ungenutzt ist oder im Laufe der Entwicklung ungenutzt wird. Gründe hierfür können zum einen sein, dass sich Anforderungen und damit der Code geändert haben, so dass Bereiche gar nicht mehr aufgerufen werden können. Zum Anderen kann es sich um Features handeln, die vom Benutzer nicht entdeckt und daher nie genutzt wurden. Da meist unbekannt ist, welcher Code nutzlos ist, verursacht er oft Kosten ohne Wert zu stiften. Bei großen grundlegenden Änderungen an der Software erhöht sich beispielsweise der Wartungsaufwand, da der unnütze Code weiterhin enthalten ist. Deshalb ist wünschenswert ungenutzten Code zu erkennen, um Aufwände einzusparen oder auch die Usability für den Benutzer zu verbessern, indem ungenutzte Features entfernt werden. In der Praxisarbeit sollen verschiedene Methoden und Werkzeuge untersucht werden, mit welchen man ungenutzte Codestellen finden kann. Ziel der Praxisarbeit ist die Entwicklung eines Konzeptes für die Einführung eines Verfahrens zu einer solchen Codeanalyse für das Enisco Manufacturing Execution System (kurz E-MES) und einer anschließenden möglichen Umsetzung.

1.2 Vorgehensweise

Zunächst soll in Kapitel 2 (Allgemeine Einführung) eine Einarbeitung in die Methoden zur Identifizierung von ungenutztem Code erfolgen. Dazu werden die verschiedenen Analysemöglichkeiten näher erläutert und gegenüber gestellt. Folgend an diese Einarbeitung soll anschließend in Kapitel 3 (Umsetzung) ein Konzept zur Implementierung der ausgewählten Analysemöglichkeiten erarbeitet werden. Ebenso in Kapitel 3 (Umsetzung) wird eine mögliche Umsetzung des Konzeptes vorgestellt.

Abschließend werden in Kapitel 4 (Reflexion und Ausblick) Anregungen für aufbauende Arbeiten gegeben.

Kapitel 2

2 Allgemeine Einführung

Jedes Unternehmen, welches Software entwickelt, ist damit bemüht die Qualität der Software stetig zu verbessern. Eine solche nachhaltige Qualitätssteigerung lässt sich durch die Kombination statischer und dynamischer Codeanalysen in einem kontinuierlichen Testprozess erzielen. Die Unterschiede der beiden Codeanalyse-Arten und die damit eingehende Frage, welche der beiden Codeanalyse-Arten geeigneter ist um ungenutzten Code zu identifizieren, werden in den folgenden Abschnitten erläutert.

2.1 Statische Codeanalyse

Bei der statischen Codeanalyse muss der Quellcode einer Software nicht ausgeführt werden. Die Analyse wird an der verfügbaren Codebasis der Software durchgeführt und liefert verschiedenste Metriken, wie zum Beispiel die Komplexität, Doppelungen, Wartbarkeit und Verlässlichkeit. Somit kann die statische Codeanalyse bereits in den frühen Phasen der Softwareentwicklung eingesetzt werden. [Son21]

Eine weit verbreitete Variante der statischen Codeanalyse ist das manuelle Code-Review. Hierbei wird nach festgelegten Abständen, zum Beispiel nach jeder abgeschlossenen Implementierung eines neuen Features, gemeinsam die Codebasis betrachtet und bewertet. Um diesen Vorgang zu automatisieren gibt es heutzutage verschiedenste Werkzeuge, welche in der Lage sind verschiedenste Metriken zu erfassen und aufbereitet darzustellen. Ein Beispiel für solche Produkte bietet die Firma SonarSource mit den Produkten „SonarLint“, „SonarCloud“ und „SonarQube“. [Son21]

Da die statische Codeanalyse keinerlei Informationen über das System während der Laufzeit besitzt, müssen die statischen Metriken der Codebasis für die Identifikation von ungenutztem Code ausreichen. Eine Identifikation von „nicht erreichbaren Code“ ist durch die statische Codeanalyse vollständig durchführbar, da hierbei lediglich die Codebasis nach Code durchsucht werden muss, welcher niemals von einem Ausführungsstrang aus aufgerufen wird.

Um trotz fehlender Laufzeitinformationen des System eine Aussage über den ungenutzten Code eines Systems treffen zu können, bedarf man sich den Metriken „Codestabilität“ und „Codezentralität“ aus der statischen Codeanalyse. Der Grund hierfür ist die Annah-

me, dass ein stabiler und dezentraler Code eine höhere Wahrscheinlichkeit besitzt, als ungenutzter Code identifiziert zu werden. [RRTS]

2.2 Dynamische Codeanalyse

Bei der dynamischen Codeanalyse erfolgen die eigentlichen Analysen während der Ausführungszeit der Software. Dies bedeutet implizit, dass die Software für eine dynamische Codeanalyse bereits frei von Kompilierungs- und Build-Fehlern sein muss, damit die Software ausgeführt werden kann [Ily19]. Klassischerweise werden bei der dynamischen Codeanalyse ein zuvor ausgewählter Satz an Daten an einen bestimmten Teil der Software übergeben und anschließend die Ausgabewerte überprüft. Somit hängt die Effizienz der Analyse direkt von der Qualität und Quantität der Testdaten ab.

Wie bei der statischen Codeanalyse, gibt es auch bei der dynamischen Codeanalyse verschiedene Werkzeuge und Frameworks, welche zum Einsatz kommen können. Ein Beispiel hierfür ist das Java-Test-Framework „JUnit“, welches besonders für automatisierte Unit-Tests einzelner Klassen oder Methoden geeignet ist. Es wird versucht mit den verschiedenen Tests eine vollständige Codeabdeckung zu erlangen.

Eine Vorteil gegenüber der statischen Codeanalyse ist die Möglichkeit auch Speicherlecks und Fehler im Zusammenhang mit der Parallelität zu erkennen. So wird Multithread-Code zur Ausführungszeit analysiert und Probleme bezüglich dem Zugriff auf gemeinsam genutzte Ressourcen oder mögliche Deadlocks gefunden.

Während es bei Test-Frameworks, wie „JUnit“, um die Frage geht, ob der zu untersuchende Codeabschnitt den vollen Funktionsumfang besitzt, geht es bei der Identifikation von ungenutztem Code um die Frage, welche Codeabschnitte aufgerufen werden und welche nicht.

Um nun mithilfe der dynamischen Codeanalyse innerhalb des Systems ungenutzten Code zu finden, bedarf es einer Überwachungen des Systems. Hierbei wird über einen längeren Zeitraum jeder Aufruf einer Methode beziehungsweise Klasse erfasst. Anhand der Ergebnisse einer solchen Überwachung lässt sich mutmaßlich ungenutzter Code identifizieren.

In Kapitel 3 (Umsetzung) wird für die dynamische Codeanalyse ein Konzept vorgestellt und die dazugehörige Umsetzung erläutert.

Kapitel 3

3 Umsetzung

Nachfolgend wird ein Konzept vorgestellt, mit welchem es möglich ist E-MES auf ungenutzten Code hin zu untersuchen. Um solchen Code zu identifizieren, wird eine dynamische Codeanalyse eingesetzt. Das dazu benötigte Konzept wird in Abschnitt 3.1 erläutert und die dazugehörige Implementierung wird in Abschnitt 3.2 aufgezeigt.

3.1 Konzept

Eine Möglichkeit um ungenutzten Code innerhalb von E-MES zu identifizieren, wäre es die Aufrufhäufigkeit von Klassen und Methoden zu überwachen, indem man vor der Ausführung des Codes in jede Methode einen Zähler implementiert, welcher beim Aufruf hochgezählt wird. Dies würde jedoch einen hohen Implementierungsaufwand und eine permanente Änderung der Codebasis zur Folge haben.

Ein anderer Ansatz beruht auf der Manipulation des, zur Laufzeit zur Verfügung stehenden Bytecodes. Solch eine Manipulation ist mithilfe von Java-Agenten und der Klassenbibliothek Javassist realisierbar.

Java-Agenten bestehen aus speziellen Klassen, welche in der Lage sind eine laufende Java-Instanz innerhalb einer JVM „abzufangen“ und deren Bytecode mithilfe von Javassist zu untersuchen und zu ändern. Man unterscheidet die Java-Agenten in statische und dynamische Agenten. Bei den statischen Agenten, wird die Manipulation des Bytecodes kurz vor der Ausführung der Codebasis durchgeführt (Bevor die Klassen von der JVM geladen werden). Bei dynamischen Agenten hingegen, ist eine Manipulation des Bytecodes auch während der Laufzeit möglich (Nachdem die Klassen von der JVM bereits geladen wurden). [Man20]

Das innerhalb dieser Praxisarbeit entworfene Konzept besteht nun daraus, einen dynamischen Java-Agent während der Laufzeit von E-MES auszuführen und eine Manipulation des Bytecodes zu erzeugen, um so jeden Methodenaufruf im System zu überwachen.

3.2 Implementierung

Wie bereits geschildert besteht die Implementierung für die Überwachung der Methodenaufrufe aus einem dynamischen Java-Agenten, welcher mithilfe von Javassist den

Bytecode von E-MES während der Laufzeit manipuliert.

Es sind drei unabhängige Programme entstanden:

EMESAgentAttacher:

Dem „Attacher“ wird die jeweilige Prozess-ID oder der Prozessname, des zu manipulierenden Programms, als Parameter übergeben. Mithilfe von diesem Parameter wird die dazugehörige virtuelle Maschine gesucht und das Programm „EMESDynamicAgent“ hinzu geladen.

EMESDynamicAgent:

Dieses Programm stellt den eigentlichen Java-Agenten dar. Ziel ist es, die Manipulation nach dem Start, also während der Laufzeit von E-MES, durchzuführen. Deshalb wurde ein dynamischer Agent implementiert. Dessen Einsprungsmethode ist die agentmain-Methode:

```
1  public static void agentmain(String agentArgs, Instrumentation inst){
2      ...
3  }
```

Mithilfe des übergebenen Parameters „inst“ erhält der Java-Agent alle geladenen Klassen. Gefiltert nach den Packages der Klassen wird nun entschieden, welche Klassen manipuliert werden sollen und welche nicht.

Wurde eine Klasse ausgesucht, werden nun alle Methoden dieser Klasse mithilfe von Javassist manipuliert. Hierfür werden die folgenden Codezeilen vom Agenten ausgeführt, welche dafür sorgen, dass am Anfang jeder manipulierten Methode ein Aufruf zu einer Java-Bean erfolgt.

```
1  StringBuilder insertString = new StringBuilder();
2  insertString.append("InitialContext context = new InitialContext();");
3  insertString.append("CountCollector collector = (CountCollector)
       context.lookup(CountCollector.LOOKUP);");
4  insertString.append("collector.notifyStats(\""+className+"\",
       \""+methodName+"\");");
5  method.insertBefore(insertString.toString());
```

In diesem Fall ist die Java-Bean die Klasse „CountCollector“ und wird über einen LOOKUP-String im Context gefunden.

addon-methodCounter:

Jede nun manipulierte Methode von E-MES ruft am Anfang ihrer Prozedur die Java-Bean auf. Übergeben werden der Klassen- und Methodenname. Die Bean überprüft anhand einer Java-Map, ob die Klassen-Methoden-Kombination bereits vorhanden ist und ergänzt diese, falls sie nicht vorhanden ist. Andernfalls wird die Anzahl der Aufrufhäufigkeit in der Map erhöht.

E-MES läuft auf Basis des Anwendungsservers Wildfly. Jener nach Java EE Standard implementierte Anwendungsserver weißt jedoch eine andere ClassLoader-Architektur auf, wie zum Beispiel die klassische ClassLoader-Architektur von Java. Üblicherweiße werden Java-Klassen zur Laufzeit von hierarchisch abhängigen ClassLoadern geladen. So genügt es den Java-Agenten mithilfe des „Attachers“ an die laufende JVM anzubinden, um vollen Zugriff auf die geladenen Klassen zu erhalten.

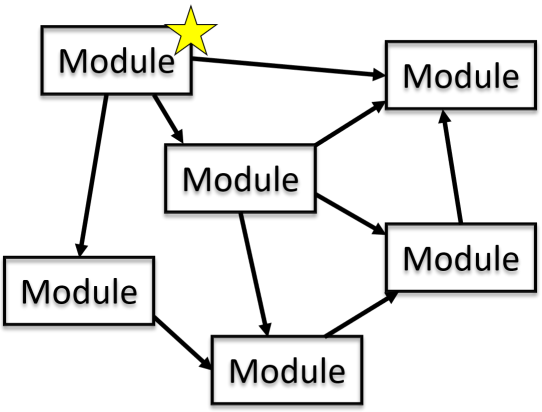


Abb. 3.1: ClassLoader-Architektur von Wildfly

Die ClassLoader-Architektur von Wildfly ist jedoch modular aufgebaut. Dies hat zur Folge, dass der Java-Agent einem Wildfly-Modul hinzugefügt werden muss, welches standartmäßig geladen wird und als Abhängigkeit von allen anderen Modulen dient. In Abbildung 3.1 ist eine solche modulare ClassLoader-Architektur abgebildet. Beispielhaft dient hier das Modul oben links (mit Stern markiert) als „Wurzel-Modul“. Für die Implementierung eines Java-Agenten in E-MES wurde hierfür das Modul „org.jboss.vfs“ ausgesucht, da es für das „Virtual File System“ innerhalb von Wildfly zuständig ist und daher als Abhängigkeit für andere Module dient.

Kapitel 4

4 Reflexion und Ausblick

Die gezeigte Analysemöglichkeit gibt Aufschluss darüber, welche Codeabschnitte während der Analysephase aufgerufen wurden und welche nicht aufgerufen wurden. Es ist jedoch nicht möglich anhand lediglich dieser Ergebnisse die vorhandene Codebasis in „nützlichen Code“ und „nutzlosen Code“ zu unterteilen.

Zunächst muss die komplette Codebasis anhand der Ergebnisse aus der Analyse in die Kategorien „nützlicher Code“ und „vermutlich nutzloser Code“ eingeteilt werden. Hierfür ist es von Nöten, dass die Überwachung des Systems mindestens ein Jahr lang durchgeführt wird. Grund hierfür ist die Tatsache, dass es in größeren Systemen auch Features geben kann, welche nur in bestimmten Zeitabschnitten ausgeführt werden. Darunter zählen zum Beispiel die Inventur oder der Jahresabschluss.

Zusätzlich ist zu beachten, dass es auch Codeabschnitte geben kann, welche nur im Falle eines Fehlers aufgerufen werden. Ein solches Beispiel ist die Fehlermeldung, welche erscheint, wenn sich ein Benutzer mit dem falschen Passwort angemeldet hat. Sollte sich kein Benutzer während des kompletten Analysezeitraums falsch angemeldet haben, wird dieser Teil der Codebasis auch nicht ausgeführt und somit fälschlicherweise als „nutzloser Code“ eingeteilt.

Aus genanten Gründen ist es deshalb unabdingbar nach erfolgter Analyse die Ergebnisse zu validieren um anschließend endgültig entscheiden zu können, ob ein gewisser Teil der Codebasis sicher entfernt werden kann.

Da der zeitliche Rahmen dieser Praxisarbeit nur die Konzeption und Umsetzung einer dynamischen Codeanalyse zugelassen hat, gilt es die Konzeption und Umsetzung einer statischen Codeanalyse, für die Identifikation von ungenutzten Code, in einer weiteren Arbeit zu behandeln.

Literaturverzeichnis

[Ily19] Ilya Gainulin ; DZone (Hrsg.): *What's the Use of Dynamic Analysis When You Have Static Analysis?* https://dzone.com/articles/whats-the-use-of-dynamic-analysis-when-you-have-st. 2019

[Man20] Manoj Debnath ; developer.com (Hrsg.): *What Is Java Agent?* https://www.developer.com/java/data/what-is-java-agent.html. 2020

[RRTS] Roman Haas ; Rainer Niedermayr ; Tobias Roehm ; Sven Apel: Is Static Analysis Able to Identify Unnecessary Source Code? https://dl.acm.org/toc/tosem/2020/29/1. In: *ACM Transactions on Software Engineering and Methodology* Bd. Volume 29

[Son21] SonarSource S.A ; SonarSource S.A (Hrsg.): *Metric Definitions*. https://docs.sonarqube.org/latest/user-guide/metric-definitions/. 2021