

Nachhaltigkeit als Qualitätskriterium von Software - Den Blick auf ressourcen-sparsame Softwareentwicklung schärfen

Oliver Radfelder,¹ Karin Vosseberg²

Abstract: In der Software Engineering Ausbildung an Hochschulen liegt meist der Schwerpunkt auf Methoden und Verfahren der Konstruktion komplexerer Softwaresysteme. Mit dem vorliegenden Beitrag wird diskutiert, wie Studierende durch analytische Maßnahmen der Qualitätssicherung für eine ressourcen-sparsame Softwareentwicklung sensibilisiert werden können. Den Studierenden werden einfache Werkzeuge an die Hand gegeben, um insbesondere die nicht-funktionalen Eigenschaften eines Softwaresystems zu untersuchen. Bei nicht-funktionalen Eigenschaften spielen neben den neu oder weiterentwickelten Softwarekomponenten auch die Laufzeitumgebung eine wesentliche Rolle, so dass auch diese in die Betrachtung der Qualitätssicherung integriert werden muss und damit auch immer Teil der Entwicklung eines Softwaresystems ist. Anhand eines konkreten Beispiels wird der Einfluss der Laufzeitumgebung auf das Verhalten und dem Ressourcenverbrauch einer Webanwendung mit den Studierenden diskutiert, um den Blick auf das Zusammenspiel zwischen Applikation und Laufzeitumgebung zu schärfen. Zudem eignet sich das Beispiel, um Vorteile von DevOps als Vorgehensmodell erlebbar zu machen.

Keywords: Softwareentwicklung; Softwarekonfiguration; analytische Qualitätssicherung; konstruktive Qualitätssicherung; Messen von Software; nichtfunktionale Qualitätskriterien; Ressourcenverbrauch; Nachhaltigkeit

1 Einleitung

Im Curriculum der Bachelorstudiengänge Informatik und Wirtschaftsinformatik an der Hochschule Bremerhaven wird neben der Programmierausbildung dem Software Engineering mit drei Modulen viel Zeit eingeräumt, um die verschiedenen Aspekte der Entwicklung komplexerer Softwaresysteme aufeinander aufbauend einzuführen: *Software Engineering I* mit Schwerpunkt Modellbildung, *Software Engineering II* mit Schwerpunkt Requirements Engineering und einer prototypischen Umsetzung zur Evaluierung von Requirements sowie *Software Engineering III* mit dem Schwerpunkt Softwarearchitekturen und der Umsetzung einer langlebigen, skalierbaren Webanwendung. Das Thema Qualität von Software spielt in allen drei Veranstaltungen eine große Rolle, so dass in den Veranstaltungen Maßnahmen der Qualitätssicherung auf den unterschiedlichen Ebenen der Softwareentwicklung eingeführt werden, die neben der funktionalen Qualität insbesondere auch die nicht-funktionalen Qualitätsmerkmale von Software betrachten. Ein besonderer Fokus liegt hierbei auf Langlebigkeit, Skalierbarkeit und Ressourcen-Sparsamkeit von Software.

¹ Hochschule Bremerhaven, An der Karlstadt 8, 27568 Bremerhaven oradfelder@hs-bremerhaven.de

² Hochschule Bremerhaven, An der Karlstadt 8, 27568 Bremerhaven kvosseberg@hs-bremerhaven.de

Geschuldet dem gemeinsamen Interesse der Kolleg:innen dieser Veranstaltungen an einer ökologischen und sozialen Nachhaltigkeit [ERV20] in der IT zieht sich das Thema des ressourcen-sparsamen Umgangs in der Softwareentwicklung durch weitere Module des Curriculums. Bereits in der *Studieneingangsphase* werden die Studierenden der Informatik und Wirtschaftsinformatik an eine ressourcen-sparsame Umsetzung von kleinen Automatisierungsketten herangeführt und in Modulen wie *Infrastrukturen* für Informatikstudierende oder *Technik für Wirtschaftsinformatik* lernen sie den Ressourcenverbrauch mit einfachen Werkzeugen zu beobachten, z.B. das Messen der Ausführungszeit einer Funktion, des Durchsatzes paralleler Anfragen oder von Speicherverbrauch. Damit wird auch der Mehrwert aus den Beobachtungen des operativen Betriebs thematisiert. Um das Thema der Nachhaltigkeit explizit in der Kultur des Informatikbereichs an der Hochschule sichtbar zu machen, wurde im Rahmen der Klimawoche an der Hochschule Bremerhaven ein ganztägiger Workshop *Ressourcensparsames Programmieren* angeboten.³

Darauf aufbauend thematisieren wir in dem Wahlpflicht-Modul *Grundlagen der Qualitätsmanagements* auf Basis von funktionalen und nicht-funktionalen Qualitätskriterien [SL19] die verschiedenen Facetten einer nachhaltigen Softwareentwicklung. Neben Energieverbrauch und Ressourcen-Sparsamkeit spielen Themen wie Wartbarkeit genauso wie Langlebigkeit von Software eine wichtige Rolle in der Diskussion um Nachhaltigkeit. In der Veranstaltung wird der Fokus gelegt auf Verfahren der analytische Qualitätssicherung und die Frage, was wir aus diesen Betrachtungen für eine konstruktive Qualitätssicherung gemeinsam lernen können. Im folgenden wird das Ziel der Veranstaltung beschrieben und anhand eines Beispiels thematisiert, wie Studierende an eine analytische Betrachtung von Software und dem Zusammenspiel mit der operativen Laufzeitumgebung der Software mit einem konkreten Versuchsaufbau herangeführt werden können. Hierbei wird ein Aspekt der Nachhaltigkeit - der ressourcen-sparsame Umgang mit Speicher in der genutzten Docker-Laufzeitumgebung - diskutiert. Außerdem fungiert das Beispiel zur Motivation eines DevOps-Vorgehensmodells, weniger im Sinne einer schnelleren Produktauslieferung [KS19] sondern eher in der gewinnbringenden Nutzung des Wissens aus dem operativen Betrieb eines Softwaresystems.

2 Analytische und konstruktive Qualitätssicherung

Im Rahmen des Wahlpflicht-Moduls *Grundlagen des Qualitätsmanagements* haben wir den Schwerpunkt auf das Zusammenspiel von Maßnahmen konstruktiver und analytischer Qualitätssicherung gelegt. Im Wechsel werden analytische Verfahren der Qualitätssicherung vorgestellt und daraufhin diskutiert wie dieses Wissen auch in der Entwicklung von Softwaresystemen genutzt werden kann. Exemplarisch wird mit den Studierenden gemeinsam erarbeitet, wie eine Umsetzung in einer Java-Umgebung aussehen kann.

³ <https://informatik.hs-bremerhaven.de/uerb/Klimawoche/>

In den Diskussionen und der Literatur zu Verfahren der analytischen Qualitätssicherung wird sehr stark auf die funktionalen Eigenschaften von Software fokussiert. Hier sind etablierte Verfahren bekannt, die einen systematischen Zugang zur Qualitätssicherung ermöglichen (siehe z.B. [SL19]). Solche Verfahren wurden in der Veranstaltung auf Basis des Lehrplans zum *Certified Tester Foundation Level* [GTB20] vorgestellt. Es wurde diskutiert, wie die Erkenntnisse aus den analytischen Testverfahren für die Programmierung der Funktionalität von Software genutzt werden können, um typische Fehler zu vermeiden und in Programmbeispielen gemeinsam umgesetzt.

Die Qualitätssicherung nicht-funktionaler Eigenschaften von Softwaresystemen wird in der Praxis oft als *Sonderaufgabe* abgeschoben und geht nicht wirklich in Alltagshandeln der Projekte über. Tests von nicht-funktionalen Eigenschaften, wie beispielsweise Last- und Performance oder effizienter Umgang mit Ressourcen scheinen unabhängig vom Vorgehensmodell kaum eine Rolle zu spielen. Dies spiegeln auch die Ergebnisse der Umfrage *Softwaretest in Praxis und Forschung* aus den Jahren 2011, 2016 und 2020 wider [WVS21]. Im Vergleich über die Jahre ist erkennbar, dass die Bedeutung unterschiedlicher Testarten seit 2011 sogar abnimmt (siehe Abbildung 1).

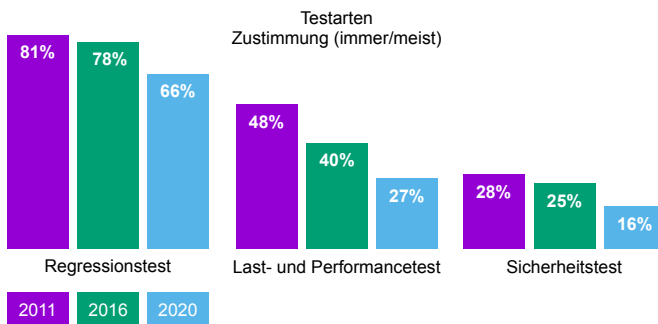


Abb. 1: Bedeutung von unterschiedlichen Testarten in Projekten

Diese Entwicklung erstaunt um so mehr im Kontext agiler Vorgehensmodelle. Mit der Transformation zu agilen Vorgehensmodellen wird die Verantwortung für Qualität in die Projekte gelegt und Maßnahmen der Qualitätssicherung in die Softwareentwicklung integriert. Dies ist auch an einem hohen Grad an Testautomatisierung auf der Unittest-Ebene zu beobachten. Kurze Entwicklungszyklen und die Diskussionen zu DevOps erhöhen die Verbindung zum operativen Betrieb. Es scheint jedoch keine Auswirkungen auf die Bedeutung nicht-funktionaler Qualitätssicherung zu haben.

In Diskussionen mit Projektmitarbeiter:innen aus Unternehmen wird die Lösung von Last- und Performanceproblemen in der horizontalen Skalierung oft mit Parallelisierung und Hinzufügen von weiteren Ressourcen gesehen, ohne den Blick auf die entstehende zusätzliche Komplexität oder einen nachhaltigen Umgang mit Ressourcen zu nehmen. Für eine Untersuchung, ob zusätzliche Ressourcen die beste Lösung zur Erhöhung der Skalierbarkeit ist, ist eine analytische Betrachtung des Gesamtsystems - Applikation und

Laufzeitumgebung - notwendig. Dies scheint jedoch in den Projekten, die den Blick haben auf der Entwicklung neuer Features ihrer (Teil-)Applikation, kaum Raum zu finden. Leider liegen solche Fragestellungen oft auch nicht im Bereich von zentralen Qualitätssicherungsteams, deren Aufgabe ist, eine Laufzeitumgebung zur Verfügung zu stellen und weiterzuentwickeln.

Die Umsetzung nicht-funktionaler Eigenschaften eines Softwaresystems ist jedoch eine wesentliche Aufgabe in der Softwareentwicklung und gehört zu den Aspekten der konstruktiven Maßnahmen der Qualitätssicherung. Werden nicht-funktionale Qualitätseigenschaften eines Systems nicht bereits in der Entwicklung und Umsetzung von Softwarearchitekturen berücksichtigt, wird es schwierig diese Eigenschaften zu erreichen. Dabei spielen gerade die Erkenntnisse aus analytischen Maßnahmen der Qualitätssicherung nicht-funktionaler Eigenschaften des Gesamtsystems und insbesondere dem Zusammenspiel zwischen Applikation und Laufzeitumgebung eine wesentliche Rolle. Sie geben wertvolle Hinweise für die Konstruktion der Softwaresysteme und sollten in das Alltagshandeln von Entwickler:innen verankert werden. Ziel der Veranstaltung ist gerade diese Aspekte der Softwareentwicklung zu adressieren, um anhand von Beispielen gemeinsam mit den Studierenden Lösungsansätze für die Analyseaufgaben zu entwickeln und ein Monitoring bereits während der Softwareentwicklung aufzubauen. Aus diesem Grund haben wir in der Veranstaltung neben den Verfahren der analytischen Qualitätssicherung funktionaler Qualitätseigenschaften auch die Analyse und das Monitoring von Systemen aufgenommen, um nicht-funktionale Eigenschaften während des Entwicklungsprozesses zu untersuchen.

Das im Folgenden beschriebene Problem des *Off-Heap Memory Leaks in Servlet-Containern* ist ein solches konkretes Beispiel, um ein Gesamtsystem zu analysieren und die Ursachen der Probleme zu lokalisieren, die oft nicht unbedingt in der Umsetzung einer Applikation liegen, sondern im Zusammenspiel mit der Laufzeitumgebung. Um dieses Problem zu lösen wird nicht selten die Strategie verwendet eine containerisierte Laufzeitumgebung mit zusätzlichen Ressourcen zu versehen, in diesem Fall zusätzlichem Speicher, und die Umgebung regelmäßig neu zu starten. Unser Ansinnen ist, das dahinterstehende Problem zu verstehen, zu lokalisieren und mit weiteren Lösungsmöglichkeiten zu vergleichen. Gemeinsam mit den Studierenden wurde das beobachtete Phänomen diskutiert und eine Hypothese formuliert. Schritt für Schritt wurde ein Versuchsaufbau entwickelt und umgesetzt, um die Hypothese zu verifizieren. Ziel ist den Studierenden Handlungsoptionen sichtbar zu machen.

Gerade im Kontext der Betrachtung einer ökologischen Nachhaltigkeit in der Entwicklung von Softwaresystemen [GGP22] wird die Analyse des Gesamtsystems ein wichtiger Bestandteil in der Beurteilung dieses Qualitätskriteriums sein, um die Hinweise für eine Einordnung zu nutzen aber auch um in der Weiterentwicklung des Gesamtsystems die Ergebnisse einfließen zu lassen. Ein Aufsetzen eines entsprechenden Monitorings bereits während der Softwareentwicklung unterstützt, dass Veränderungen, die durch Weiterentwicklungen oder Migrationen in eine neue Laufzeitumgebung entstehen, sofort erkannt werden können. Damit begleiten solche Maßnahmen intensiv den Softwareentwicklungsprozess und unterstützen die kontinuierliche Qualitätsverbesserung. Insbesondere im Kontext von *DevOps* entsteht so

eine enge Verbindung zwischen der (Weiter-)Entwicklung von Software und dem Betrieb der Softwaresysteme.

3 Beispiel: Off-Heap Memory Leaks in Servlet-Containern

In der Infrastruktur der Informatik an der Hochschule Bremerhaven werden den Studierenden Docker-Container zur Verfügung gestellt, in denen als Servlet-Engines *tomcat*, *wildfly* und *jetty* installiert sind. Die Java-Versionen 8, 11 und 17 stehen ebenfalls bereit. Die Container selbst werden mit je vier virtuellen CPUs und zwei GB virtuellem Speicher gestartet. Die Container sind sehr bewusst mit beschränkten Ressourcen ausgestattet, um einen ressourcen-sparsamen Umgang in der Softwareentwicklung zu forcieren.

In aktuellen Webanwendungen werden Websockets als Technologie eingesetzt. Seit einigen Jahren ist bereits zu beobachten, dass eine Webanwendung mit Websockets in einer so RAM-beschränkten Umgebung im *tomcat* in den Versionen 8, 9 und nun auch 10 unter Last schnell so viele Ressourcen verbraucht, dass der *OOM-Killer (OutOfMemory)* den Java-Prozess beendet. Das betrifft alle drei verfügbaren Java-Versionen.

3.1 Testkonstellation

Um festzustellen, ob das Problem durch einen Programmierfehler im *tomcat* oder der Java-Runtime verursacht wird, oder aber die beschränkten Ressourcen als solche der Grund sind, wird eine kleine Webanwendung deployt. Sie besteht lediglich aus einem WebSocket-Endpunkt und einem kleinen `ContextListener`, der sekundlich den aktuell genutzten Heap, die Anzahl der empfangenen Nachrichten sowie den vom Tomcat und dem Docker belegten Hauptspeicher (`Resident Set Size - RSS`) ins Log schreibt.

Zusätzlich werden zwei *node*-Prozesse auf einem weiteren Host gestartet, die jeweils 2500 Websockets öffnen und sequenziell Nachrichten an den Endpunkt senden und die Antwort abwarten (Echo-Service). Nach 100 Nachrichten wird der WebSocket geschlossen und ein neuer geöffnet, so dass insgesamt durchgängig über vier Stunden ca. 5000 aktive Websockets verbunden sind. Die Experimente wurden zu unterschiedlichen Tageszeiten durchgeführt.

3.2 Erste Beobachtung

In der Standardkonfiguration des *tomcat 10* unter Java SE 17 stellt sich heraus, dass bei 2 GB virtuellem Speicher der OOM-Killer den Java-Prozess innerhalb von 10 Minuten beendet. Bei zugewiesenen 3 GB virtuellem Speicher läuft der Prozess auch nach Stunden noch. Allerdings steigt der Verbrauch auf über 2.5 GB (siehe Abbildung 2).

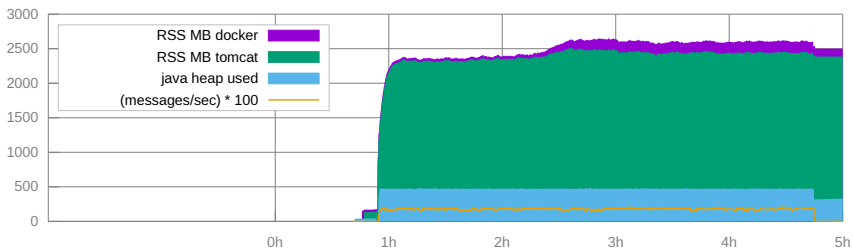


Abb. 2: *tomcat 10* im Docker-Container: der Off-Heap Speicherverbrauch steigt über 2 GB bei maximalem Heap von 512 MB

Zusätzlich ist zu beobachten:

- die Anzahl der Nachrichten pro Sekunde liegt bei etwa 20.000
- der Java-Heap liegt tatsächlich nahezu durchgehend bei 500 MB

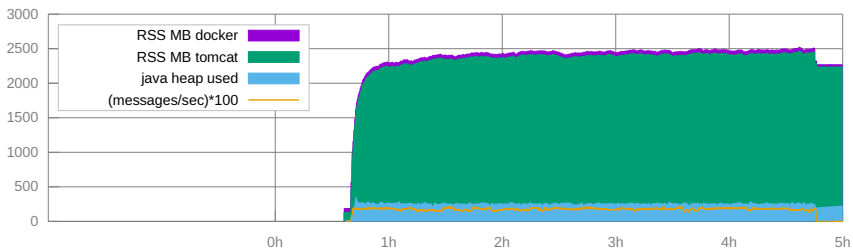
Diese Kombination lässt darauf schließen, dass sowohl auf den Garbage-Collector als auch auf das Verarbeiten der Websocket-Nachrichten nicht unerhebliche Anteile an dem CPU-Verbrauch fallen. In dem vorliegenden Beispiel haben wir den CPU-Verbrauch nicht weiter betrachtet, sondern das Augenmerk auf den Speicherverbrauch gelegt.

Die gleichen Ergebnisse sind mit *tomcat* in den Versionen 8 und 9 sowie den Java-Versionen 9 und 11 zu beobachten. Als Werkzeuge haben sich nach mehreren Experimenten u.a. mit *pmap* und *ps* letztlich *docker stats*, *pidstat* und *jcmd* als besonders geeignet herausgestellt, mit denen die Daten für die Aufbereitungen erzeugt wurden.

Die erste Arbeitshypothese: Irgendwo in der Implementierung des *tomcat* existiert ein Problem, das zu einem übermäßigen Off-Heap Verbrauch führt.

3.3 Prüfung der Hypothese mit *jetty*

Wenn es denn an der Implementierung des *tomcat* liegt, sollte eine andere Servlet-Engine ein anderes Verhalten zeigen. Daher wird die gleiche Anwendung mit den gleichen Parametern (512 MB max Heap, etc.) in *jetty* deployt und mit den gleichen Clients getestet. In Abbildung 3 ist zu erkennen, dass beim *jetty* das gleiche - oder zumindest ein vergleichbares - Problem auftritt: nach sehr kurzer Zeit steigt der von *docker stats* angezeigte Speicherverbrauch auf deutlich über 2 GB und erreicht nach etwas über vier Stunden 2.5 GB. Bekommt der Docker-Container lediglich 2 GB virtuellen Speicher zugewiesen, wird der Java-Prozess nach weniger als 10 Minuten von dem OOM-Killer beendet.

Abb. 3: *jetty 11* im Docker-Container

Weiterhin ist zu beobachten:

- auch hier beträgt die Anzahl der abgearbeiteten Nachrichten pro Sekunde etwa 20.000
- der tatsächlich gebrauchte Heap beträgt im Mittel nur etwa die Hälfte von dem, den der *tomcat* benötigt

Damit ist zwar die Hypothese des Implementierungsproblems im *tomcat* nicht widerlegt, aber da beide Servlet-Engines das gleiche Phänomen zeigen, könnte hier auch ein fundamentales Problem zu Tage treten: es könnte ein generelles Problem in der JVM-Implementierung bzw. in den Standardbibliotheken sein.

3.4 Prüfung der Hypothese mit *wildfly*

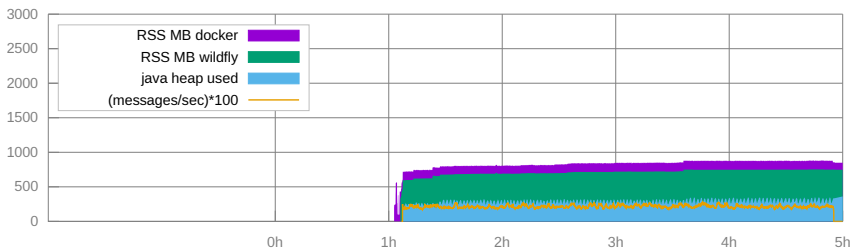
Als weitere Variante wird die Webanwendung im *wildfly* deployt. Interessanterweise zeigt sich hier nicht das Verhalten der beiden anderen Servlet-Engines. Wie in Abbildung 4 zu sehen, steigt der Gesamtverbrauch auf knapp unter 800 MB (und bleibt auch nach zwölf Stunden dort).

Zusätzlich ist zu beobachten:

- die Anzahl der abgearbeiteten Nachrichten pro Sekunde beträgt etwas unter 20.000
- der tatsächlich gebrauchte Heap beträgt im Mittel deutlich weniger als bei *jetty*

3.5 Analyse und zweite Hypothese

Die beiden Servlet-Engines *jetty* und *tomcat* nutzen in der jeweiligen Standardkonfiguration *per-message-deflate* bei Websockets. Bei *wildfly* ist das nicht der Fall.

Abb. 4: *wildfly preview 26* im Docker-Container

Mindestens seit Java SE 8 gibt es offensichtlich häufiger Schwierigkeiten mit Off-Heap Memory Leaks bei Anwendungen, die die Kompressionsalgorithmen aus dem Paket *java.util* nutzen. Augenscheinlich ist die Implementierung recht schwierig, korrekt anzuwenden, so dass die Vermutung naheliegt, dass die Servlet-Engines zur Implementierung des *per-message-deflate*-Features diese Klassen nutzen und es daher zum übermäßigen Speicherverbrauch kommt.

Um die Hypothese zu prüfen, wird das Feature im *tomcat* deaktiviert. Hierbei ist zu beobachten, dass über zwölf Stunden hinweg die Testanwendung im vertretbaren Rahmen unter 1 GB bleibt. Besonders interessant ist, dass die Anzahl der abgearbeiteten Requests pro Sekunde sich mit knapp unter 40.000 fast verdoppelt. Um das Experiment abzuschließen, könnte noch versucht werden, bei *jetty* das Feature zu deaktivieren und es beim *wildfly* zu aktivieren. Beides ist schwierig und mit erheblichem Aufwand verbunden.

3.6 Technisches Zwischenfazit

Für eine produktive Anwendung, in der mit Websockets gearbeitet wird und die über Tage - vielleicht sogar Wochen stabil laufen soll, sind *wildfly* in der Standardkonfiguration und *tomcat* mit deaktiviertem *per-message-deflate* vermutlich gleichermaßen geeignet. *tomcat* stellt sich als leistungsfähiger in Sachen *Messages per Second* heraus, *wildfly* scheint in Bezug auf Heap-Auslastung deutlich effizienter. Hier wären weitere Experimente durchzuführen, um herauszufinden, wie sich die beiden Engines in realeren Anwendungen mit Sessions und mehr fachlichen Objekten verhalten: es mag sein, dass der *tomcat* auf Dauer mehr und mehr Ressourcen auf den Garbage-Collector verwendet und der Heap deutlich vergrößert werden müsste.

In allen Varianten steigt der von *docker stats* angezeigte Speicher sehr langsam aber kontinuierlich an. Keiner der Prozesse im Docker-Container rechtfertigt diese Beobachtung, die beim *tomcat* am deutlichsten auftritt. Zudem - auch wenn es nicht mehr so rapide ist - steigt der Speicher des Java-Prozesses selbst noch immer stetig an. Ein gelegentliches *jcmd \$(pidof java) System.trim_native_heap* scheint da etwas Abhilfe zu schaffen.

In jedem Fall gehört diese Art des Systemmonitorings auch in produktiven Systemen zum soliden Handwerk. Ähnliche Beispiele haben die Studierenden in ihren Ausarbeitungen aufgegriffen und eigene Analysen durchgeführt, die am Ende des Semesters in Form einer Plakatausstellung präsentiert wurden.

4 Fazit

Das hier vorgestellte Beispiel des *Off-Heap Memory Leaks in Servlet-Containern* zeigt exemplarisch das Lernsetting in dem Modul *Grundlagen der Qualitätsmanagements* auf. Schritt für Schritt erarbeiten wir gemeinsam mit den Studierenden entlang einer konkreten Implementierung einer Java-Applikation wie eine Analyse eines Gesamtsystems aufgebaut werden kann und wie diese Erkenntnisse in die Verbesserung der Qualität des Gesamtsystems eingehen. In dem konkreten Beispiel konnte durch eine veränderte Konfiguration der Laufzeitumgebung - dem Tomcat - das Problem zumindest teilweise behoben werden ohne zusätzliche Ressourcen dem Container zuzuordnen. Es konnte gezeigt werden, dass eine Analyse des Gesamtsystems auf einfachen, sehr effektiven Werkzeugen des Betriebssystems aufgebaut werden konnte.

Mit weiteren Beispielen haben wir im Laufe der Veranstaltung verschiedene Facetten der Nachhaltigkeit in der Softwareentwicklung aufgegriffen und die Studierenden für das Zusammenspiel von Softwareentwicklung, Qualitätssicherung und den operativen Betrieb sensibilisiert. Durch die gemeinsame Erarbeitung der konkreten Umsetzung der Beispiele in Java-Applikationen hat es für die Studierenden greifbarer gemacht und zumindest teilweise Einzug in ihr Alltagshandeln in der Softwareentwicklung gefunden.

Literaturverzeichnis

- [ERV20] Erb, U.; Radfelder, O.; Vosseberg, K.: Nachhaltigkeitskultur in der Informatik. FifF Kommunikation, 2020(1/20):40–42, 2020.
- [GGP22] Gerstlacher, J.; Groher, I.; Plösch, R.: Green und Sustainable Software im Kontext von Software Qualitätsmodellen. HMD Praxis der Wirtschaft, 59(4):1149–1164, 2022.
- [GTB20] GTB - German Testing Board e.V.: Lehrplan *Certified Tester Foundation Level*, 2020 <https://www.german-testing-board.de> (letzter Aufruf am: 2022-10-20).
- [KS19] Kasteleiner, B.; Schwartz, A.: DevOps - Schnell, zuverlässig und sicher von der Idee zur Realisierung. Informatik Spektrum, 42(3):211–214, 2019.
- [SL19] Spillner, A.; Linz, T.: Basiswissen Softwaretest, 6. Auflage. dpunkt-Verlag, Heidelberg, 2019.
- [WVS21] Winter, M.; Vosseberg, K.; Simon, F.: Technischer Report: Umfrage 2020 - Softwaretest in Praxis und Forschung. 2021, <https://www.softwaretest-umfrage.de/pdf/Technischer-Report-Umfrage-2020-V11Linked.pdf>, (letzter Aufruf am: 2022-10-20).