Iris Groher, Thomas Vogel (Hrsg.): Software Engineering 2023 Workshops,
Digital Library, Gesellschaft für Informatik, 2023 78

Adaption des Software-Qualitätsmanagements im Automotive-Bereich für eine Nutzung von Fremdkomponenten

Joachim Schlosser[1], Alexander Mattausch[2], Moritz Neukirchner[3], and Rainer Holve[4]

Abstract: OEMs verlangen für alle zugekaufte Software die Einhaltung von Standards wie ASPICE und oft auch ihrer eigenen. In High Performance Controllern wird jedoch bereits in großem Umfang nicht-konforme Software eingesetzt. Softwarelieferungen werden eher aufgrund von Erfahrungswerten als aufgrund einer systematischen Risikobewertung akzeptiert. Viele der vorhandenen Open-Source-Software-Komponenten haben sich als zuverlässig erwiesen und läuft sogar auf kritischen Teilen der Internet-Infrastruktur. Keine dieser Komponenten ist nach automobilen Entwicklungsprinzipien entwickelt worden, dennoch handelt es sich um insgesamt robuste und qualitativ hochwertige Implementierungen. Wie kann das Qualitätsmanagement von diesen Softwareprojekten lernen und dadurch die Entwicklungseffizienz steigern, ohne dass die Codequalität darunter leidet?

Keywords: Qualitätssicherung, Softwarequalität ASPICE, Open Source, Legacy Software, Lizenzierung, Entwicklungsrisiko.

1 Einführung

Automobilhersteller verlangen gegenwärtig für alle gekaufte Software die Einhaltung ihrer eigenen Software-Qualitätsstandards sowie normativer Standards wie ASPICE. [1] In High Performance Controllern (HPC) wird jedoch bereits in großem Umfang nicht konforme Software eingesetzt, z. B.

- Linux und native Legacy-Userland-Anwendungen
- OSS-Kommunikationsstacks (z. B. MQTT)
- Weitere OSS (z. B. Krypto-Bibliotheken)

Softwarelieferungen werden eher auf der Grundlage von Erfahrungswerten als auf der Grundlage einer systematischen Risikobewertung akzeptiert.

Dies hat seinen Grund: Viele der vorhandenen Open-Source-Software-Komponenten (OSS) haben sich als zuverlässig erwiesen und betreiben sogar kritische Teile der Internet-Infrastruktur. Beispiele für solche Komponenten sind der Linux-Kernel, der GCC-Compiler, der Apache-Webserver oder *systemd*, die zu den Standardkomponenten

[1] Dr. Joachim Schlosser, Elektrobit Automotive GmbH, Georg-Brauchle-Ring 23, 80992 München, joachim.schlosser@elektrobit.com, https://orcid.org/0000-0002-8142-2648
[2] Dr. Alexander Mattausch, Elektrobit Automotive, Am Wolfsmantel 46, 91058 Erlangen
[3] Dr.-Ing. Moritz Neukirchner, Elektrobit Automotive GmbH, Hannoversche Str. 60 D, 38116 Braunschweig
[4] Dr. Rainer Holve, Elektrobit Automotive GmbH, Am Wolfsmantel 46, 91058 Erlangen

industrietauglicher Softwarelösungen gehören. Keine dieser Komponenten ist nach automobilen Entwicklungsprinzipien entwickelt worden, dennoch handelt es sich um insgesamt robuste, qualitativ hochwertige Implementierungen. Wie kann das Qualitätsmanagement von diesen Softwareprojekten lernen und dadurch die Entwicklungseffizienz steigern, ohne dass die Codequalität darunter leidet?

Der vorliegende Beitrag erörtert Lösungsansätze aus unserer industriellen Softwareentwicklungspraxis.

2 Verwandte Arbeiten

In der Forschung finden sich unter anderem Arbeiten zu Metamodellen der Softwarequalität von OSS wie in [2], oder Ontologie-basierte Ansätze wie [3], die eher ein lexikalischer Vergleich verschiedener Messungsmethoden für Software und Softwarequalität verfolgen.

Ein Vergleich verschiedener Frameworks und eine Analyse der Verwendung von OSS im SPICE-Kontext [4, 5] in [6, pp. 176-186] liefert zwar eine Tauglichkeit, lässt sich jedoch nur wenig über die Praktikabilität der Anwendung der durch die Norm geforderten Prozesse aus. Automatisierung, auch in der Empfehlung von Maßnahmen, wird oft propagiert [7] und ist aus technischer Sicht auch sinnvoll, lässt jedoch die Frage nach der Leistbarkeit im Kontext umfangreicher Drittanbieter-Bestandssoftware offen. SPICE selbst [4, 5] referenziert zwar in SWE.3 BP7 (Base Practice) und BP 8 auf Quellcode, bringt aber der tatsächlichen Code-Entwicklung noch nicht hinreichend Wertschätzung entgegen.

Da Softwareentwicklung im Unternehmenskontext ein fortlaufendes Unterfangen ist, und auch neue Entwicklungen beständig an Kunden ausgeliefert werden müssen, ist eine Vorgehensweise notwendig, die diese Ansätze zwar berücksichtigt, aber dennoch regelmäßig Software in erforderlicher und definierter Qualität ausliefert.

Eine gute Basis für eine Klassifizierung von Software bietet der Ansatz des „Tool Criticality Level“ aus der ISO 26262 [8], which we extended to on target software.

3 Codezentriertheit

Ein Schlüsselaspekt von OSS-Projekten ist ihre *Codezentriertheit*. In erfolgreichen Projekten sind die wichtigsten Kriterien für den Beitrag in der Regel:

1. Ist der Code korrekt?
2. Ist der Code verständlich und wartbar?
3. Passt er zur Softwarearchitektur?
4. Folgt er den Kodierungs- und Implementierungsrichtlinien des Projekts?

5. Ist der Code performant?

Im Allgemeinen entscheiden die *Projektexperten*, ob der Beitrag die notwendigen Qualitätskriterien erfüllt.

Alles, was in diesem Kapitel als „Code“ bezeichnet wurde, schließt automatisch auch Modellierungsartefakte ein. Ein Modell, das simuliert oder ausgeführt werden kann, ist ebenfalls als *Code* zu betrachten.

Im Linux-Kernel z. B. werden Beiträge auf der Linux-Kernel-Mailingliste (LKML) gründlich diskutiert und müssen von erfahrenen Kernel-Experten „abgesegnet“ werden, bis sie für die Aufnahme in den Hauptstrang des Kernels freigegeben werden. Selbst wenn sie genehmigt sind, durchlaufen sie mehrere Stufen, bis sie es in eine Linux-Version schaffen, siehe LKML, Linux-Kernel-Entwicklungsprozess [9, 10].

Als weiteres Beispiel weist die Mozilla Firefox-Entwicklung neuen Mitwirkenden erfahrenen Mitwirkende zu, die sie mit den Projektprozessen und -vorschriften vertraut machen, siehe Mozilla-Entwicklungsprozess [11].

All diese Entwicklungsmodelle haben zwei Dinge gemeinsam:

1. Starker Fokus auf Experten: Die erfahrenen Gutachter arbeiten oft *jahrelang* am jeweiligen Projekt, bis sie das Recht erhalten, Code-Beiträge (z.B. Linux-Kernel) anzunehmen.
2. Fokus auf Code und Implementierungsqualität: Alle Prozesse sind auf die Qualität der *Implementierung* ausgerichtet. Alles andere wird nicht als unnötig, sondern als weniger wichtig angesehen. Die Frage, die jeder Prozessschritt beantworten muss, lautet: „Wie verbessert das die Implementierung?“

Im Gegensatz dazu konzentrieren sich die Entwicklungsprozesse in der Automobilindustrie sehr stark auf *Anforderungen* und *Rückverfolgbarkeit*. Abgesehen von Sicherheitsaspekten, die solche Ansätze vorschreiben, um den Nachweis für Sicherheitsaussagen zu erbringen, ist ein Hauptgrund für dieses Modell die Möglichkeit, zwischen mehreren Anbietern zu wählen: Das Wissen über das System ist in den Anforderungen enthalten, daher muss der Lieferant den Nachweis erbringen, dass die erforderliche Funktionalität implementiert und getestet ist. Um Abkürzungen auf Kosten der Qualität zu vermeiden, werden den Lieferanten Standards wie ASPICE auferlegt, um ein bestimmtes Qualitätsniveau beizubehalten, obwohl Low-Cost-Entwickler in vielen Fällen nicht über die für die Komplexität der Projekte erforderlichen Kompetenzen verfügen.

Für die Komplexität der Projekte und Systeme, wie sie bisher üblich waren, funktionierte dieses Modell gut:

1. Durch die Vergabe der gleichen Anforderungen an mehrere Lieferanten wurde eine zweite Quelle geschaffen.

2. Ein anderer Lieferant kann z.B. für die nächste Generation nominiert werden, da die Qualität der Implementierung mit ASPICE überwacht werden kann. [1]

Für moderne hochkomplexe Projekte und Systeme wie das ICAS (In Car Application Server) von Volkswagen [12, 13, 14] stößt dieses Modell an seine Grenzen. Der Mehrwert verlagert sich von den Anforderungen auf die Implementierung, in manchen Bereichen ist es nicht einmal mehr möglich oder bezahlbar, das Systemverhalten vollständig in den Anforderungen auszudrücken. Stattdessen wird High-Level-Funktionalität spezifiziert, die auf etablierte Standards oder verfügbare Komponenten verweist.

Die Fokussierung der Anforderungen auf zentrale Aspekte und insbesondere das „Was“ bei gleichzeitigem Offenlassen des „Wie“ ermöglicht:

1. Generell kürzere Markteinführungszeiten und Iterationszyklen, da sich das „Wie“ im Laufe der Zeit weiterentwickeln kann,
2. Mehr Flexibilität und Freiheit in der Systemgestaltung,
3. Verbesserte Konsistenz der Prozess- und Qualitätsstandards für das Gesamtsystem durch Fokussierung auf die zentralen Elemente,
4. Überschaubaren Umfang der Spezifikation auch bei hochkomplexen Systemen durch Offenlassen der Details,
5. Integrierte und referenzierte etablierte externe Standards für spezifische Funktionalitäten.

Die Nachrüstung bestehender Software im ASPICE-Kontext ist mühsam bis kaum möglich, doch lässt sich offenkundig auch außerhalb von ASPICE Software in hoher Qualität entwickeln. Da in einem codezentrierten Modell der Code ein integraler Bestandteil des Gesamtsystems ist und nicht „nur“ die Umsetzung ausgefeilter Anforderungen, werden die einzelnen Prozessbestandteile „Anforderungen“, „Design“ und „Implementierung“ als eine Einheit gesehen und *unterstützen* sich gegenseitig. Dies steht im Gegensatz zu klassischen, eher wasserfallorientierten Implementierungsmodellen, bei denen die Implementierung das Design und die Anforderungen *verfeinert*. Das Modell der Codezentrierung steht jedoch im Einklang mit dem agilen Paradigma „funktionierende Software über umfassende Dokumentation“: [15] vernachlässige nicht die Anforderungen und das Design (trotz der Behauptungen einiger agiler Evangelisten), sondern *fokussiere* auf die Implementierung. Dies muss für integrierte oder zugekaufte Software von Fremdherstellern, vor allem aber für Eigenentwicklungen gelten. Aufgrund der Tiefe und Komplexität von Lieferketten wird die Kommunikation über Anforderungen tendenziell ineffizient. Die direkte Zusammenarbeit auf Quellcode-Ebene – die Quellcodezentriertheit – ermöglicht eine direktere Integration und Prüfung, was zu einer schnelleren Validierung führt. Die Spezifikation wird wieder das, was sie sein soll: Eine Unterstützung der Implementierung, statt umgekehrt. Statt eher Spezifikationen zu belassen und Implementierungen neu erstellen zu lassen, wie es bei neuen Fahrzeuggenerationen der Regelfall war, wird die existierende Implementierung weiterentwickelt. Dazu muss diese selbst eine hohe Qualität aufweisen,

was durch die Spezifikation sowohl im Design der Softwarearchitektur als auch beim Test unterstützt wird.

Bei immer komplexeren Systemen verlagert sich die Wertschöpfung von den Anforderungen hin zur Implementierung. Prozess und Wissen der Entwickler sollten sich daher auf die Qualität der Implementierung, d.h. der Softwarearchitektur und des Codes, konzentrieren. Ein wichtiger Aspekt ist daher der Übergang von der *Portabilität* der Implementierung (z. B. zu einem anderen Anbieter) zur *Wiederverwendbarkeit* der Implementierung. Während Portabilität eher auf den Grad der Geräteunabhängigkeit für einen Wechsel der Ablaufumgebung abzielt, hat die Wiederverwendbarkeit eher Einfachheit und Modularität für Verwendung von Softwaremodulen in weiteren Projekten im Sinn. [16, p. 15]

4 Risikobasierte Zuordnung von Prozess- und Qualitätsstandards

Ein weiterer Aspekt zur Bestimmung der erforderlichen Strenge für Prozesse und Qualitätsstandards ist die Frage: Welches *Risiko* wird durch eine bestimmte Softwarekomponente verursacht?

Ein Risiko kann nach der Schwere des Schadens kategorisiert werden, den eine Softwarekomponente verursachen könnte:

1. Gefährdung der Gesundheit und des Lebens der Fahrzeuginsassen,
2. Risiko eines finanziellen Verlusts aufgrund von Fehlfunktionen und anschließenden Reparaturkosten nach SOP,
3. Risiko erhöhter Entwicklungskosten aufgrund von fehlerhaften oder qualitativ minderwertigen Beiträgen.

Das erste und schwerwiegendste Risiko wird durch die geltenden Sicherheitsnormen abgedeckt, und diese sind unumgänglich. Für eine Softwarekomponente mit einer zugeordneten Sicherheitsanforderung muss der Nachweis erbracht werden, dass diese Sicherheitsanforderung korrekt analysiert und mit der durch den zugewiesenen Automotive Safety Integrity Level (ASIL) [17] spezifizierten Sicherheit umgesetzt wurde. Es müssen also die erforderlichen Sicherheitsmechanismen und alle notwendigen Eigenschaften der Implementierung vorhanden sein. Dies gilt sowohl für integrierten OSS-Code als auch für neu entwickelten Code.

Es gibt zwei verschiedene Arten von Risiken, die einzeln bewertet werden müssen:

- Risiken aus dem Einsatzbereich der Software
- Risiken, die durch die Softwarequalität selbst verursacht werden

4.1 Risiken durch den Einsatzbereich

Der Schaden, den ein Fehlverhalten einer Software anrichten kann, hängt wesentlich davon ab, wo die Software eingesetzt wird. So ist beispielsweise eine Fehlfunktion zur Laufzeit der Software, die im Steuergerät eines Autos ausgeführt wird, kritischer als das Fehlverhalten eines Software-Tools zur Entwicklungszeit. Ausnahmen können Werkzeuge sein, die mit sicherheitsrelevanter Software interagieren, für die eine explizite Kritikalitätsanalyse vorgeschrieben ist. Oder andersherum: Es kann besser sein, ein qualitativ schwächeres Entwicklungswerkzeug zu verwenden, das bei der Fehlersuche hilft, als sich auf eine qualitativ hochwertige Entwicklungsumgebung zu verlassen, der es jedoch an der notwendigen Funktionalität mangelt.

Kurz: Das geforderte Qualitätsniveau muss dem Zweck entsprechen.

Um diese Entsprechung für hunderte von Drittsoftware-Packages kategorisieren zu können, ob diese qualifiziert werden müssen, oder ob sie „as is“ mitgeliefert werden können, welche können wir einfach „as-is“ mitliefern, ohne irgendwelche Garantien? Auch, wenn die Einordnung an den ASIL [17] erinnert, besteht keine direkte methodische Verbindung.

Wir haben die folgenden Risikoklassen identifiziert, in die Automotive-Software eingeteilt werden kann:

1. On-Target, uneingeschränkte Exposition:
    - Laufzeitsoftware auf dem Steuergerät mit klar erkennbarer Funktionalität. Kategorie 1 oder 2 der obigen Liste.
2. Off-Target Kompilierungssoftware
    - Risiko eines Schadens: hoch, möglicherweise auch sicherheitsrelevant. Kategorie 1 oder 2.
3. On-Target, begrenztes Risiko
    - Laufzeitsoftware auf dem Steuergerät, die unkritisch ist und selten zur Anwendung kommt
    - Schadensrisiko: mittel bis hoch. Kategorie 2.
4. Build-Tooling zur Erstellung der On-Target-Software, On-Target-Entwicklungssoftware
    - Entwicklungswerkzeuge, die auf dem Target laufen, aber zur Produktion entfernt werden
    - Build-Tools, die an der Erstellung der Target-Images beteiligt sind
    - Schadensrisiko: mittel, meist Kategorie 3, manchmal 2.
5. Entwicklungswerkzeuge (off-target)
    - Alle anderen Entwicklungswerkzeuge wie IDEs, Test-Frameworks usw.
    - Schadensrisiko: gering bis mittel, könnte zu Verzögerungen bei der Entwicklung führen. Kategorie 3.

6. Debug-Tools (On- und Off-Target)
    - Werkzeuge zur Identifizierung von Fehlern im System
    - Schadensrisiko: meist gering, hilft bei der Identifizierung von Fehlern. Könnte meist zu Verzögerungen in der Entwicklung führen. Kategorie 3.

Dies ist eine auf unserer Erfahrung mit der Integration von Drittsoftware basierende „Daumenregel"-Klassifizierung und berücksichtigt, wie bereits erwähnt, keine sicherheitsgerichtete Software. Hierfür muss eine Sicherheitsanalyse (on-target) oder eine Kritikalitätsanalyse (off-target) durchgeführt werden, um die Risiken der Software für ein Sicherheitsziel gründlich zu bewerten.

Generell kann es sinnvoll sein, die von der Software geforderten Qualitätskriterien nach den Risiken zu klassifizieren, die von der Software ausgehen. Eine solche Klassifizierung zusammen mit einer Begründung kann auch die Akzeptanz von starren Qualitätsmaßnahmen durch die Entwickler verbessern, wenn ihnen klar ist, dass und warum eine Softwarekomponente als „hohes Risiko" gekennzeichnet ist.

Wie würde die Log4Shell-Sicherheitslücke [18] in dieses Bild passen? Der für dieses Problem der Informationssicherheit verantwortliche Code ist Teil einer Logging-Bibliothek, die die Entwicklung und das Debugging unterstützt und kaum zu den unternehmenskritischen Funktionen gehört. Wie lässt sich ein solches Szenario am besten entschärfen, da log4j wahrscheinlich nicht in die höchste Risikoklasse eingestuft worden wäre?

Erstens hindert die Risikokategorisierung nicht, eine angemessene Sicherheitsarchitektur (Security) zu schaffen, die die Exposition einer möglichen Schwachstelle durch Design reduziert: Eine Schwachstelle, die nicht zugänglich ist, kann nicht ausgenutzt werden. Zweitens war die inkriminierte Funktion, die die Schwachstelle verursachte, kein Implementierungsfehler, sondern ein dokumentiertes Feature: die Möglichkeit, log4j über das „Java Naming and Directory Interface" (JNDI) und eine lokale XML-Datei zu konfigurieren. Dabei wurde übersehen, dass JNDI eine ganze Reihe von Netzwerkprotokollen unterstützt, um Daten von überall im Internet abzurufen. [19, 20] Folglich verringert dieser Ansatz nicht den Aufwand für ein sorgfältiges Systemdesign und eine kritische Auswahl der verwendeten Funktionen einer Komponente eines Fremdherstellers. Jede verwendete Funktion muss bewertet, verstanden und auf mögliche unerwünschte Nebeneffekte hin analysiert werden.

ASPICE zielt darauf ab, das Risiko zu verringern, indem es nicht spezifizierte Funktionalität verbietet und damit implizit die Verwendung von Open-Source-Software ausschließt. Somit muss jeder ASPICE-konforme Entwicklungsprozess ein angemessenes Systems Engineering durchführen, um das Verhalten auf Systemebene zu spezifizieren. Die gleiche Art von Systems Engineering ist auch in jedem Projekt erforderlich, das Open-Source-Software verwendet, doch anstatt die Spezifikation aller enthaltenen Software vorzuschreiben, sollte der Schwerpunkt auf Maßnahmen zur Risikominimierung liegen, um mögliche Interferenzen und Nebeneffekte zu verhindern.

Die Analyse und Klassifizierung des Anwendungszweckes der Komponente bedeutet hierbei zwar zunächst Mehraufwand, der in unserer Erfahrung jedoch durch Reduktion und Fokussierung des späteren Aufwandes bei der Qualitätsanalyse deutlich mehr als wettgemacht wird.

4.2 Risiken in Bezug auf Softwarequalität

Die Risiken aus dem Einsatzbereich bestimmen die notwendige Stringenz, mit der eine Software entwickelt und bewertet werden muss. Die Risiken, die auf der Softwarequalität der beigesteuerten Softwarekomponente selbst beruhen, müssen ebenfalls bewertet werden, um beurteilen zu können, ob eine Komponente die Kriterien für das Nutzungsrisiko erfüllt.

Während bei selbst entwickelter Software die Qualitätskriterien entsprechend angepasst werden können, muss eingebrachte Software (OSS-Komponenten, 3rd Party Software) anders bewertet werden. Außerdem müssen insbesondere bei beigesteuerten OSS-Komponenten auch die rechtlichen Risiken berücksichtigt werden, was eine sorgfältige Prüfung der Lizenz(en) erfordert, unter denen die jeweilige Komponente veröffentlicht wurde.

Die Einbindung von OSS-Komponenten in die eigene Software erfolgt in drei Schritten:

1) Lizenzprüfung,
2) vorgelagerte Qualitätsprüfung,
3) Qualifizierung der Komponente.

Lizenzprüfung

Das Ziel der Lizenzprüfung ist es, zu überprüfen, ob die OSS-Komponente überhaupt eingebunden werden kann. Darf zum Beispiel der eigene Quellcode veröffentlicht werden, wenn wir auf GPLv2-lizenzierte Software verlinken? Ist es erlaubt, GPLv3-lizenzierte Komponenten in ein Gerät einzubinden, das mit einem Secure Boot-Schema versehen ist? Die Antworten hängen auch davon ab, wie und wo die Software eingesetzt werden soll:

- On-Target für den Produktionseinsatz
- Off-Target-Verwendung, z. B. auf einem Build-Host
- Ausschließliche Verwendung in der Entwicklung (off- und on-target), keine Verwendung in der Produktion

Bei der Lizenzprüfung müssen verschiedene Fragen berücksichtigt werden, zum Beispiel:

- Ist die Lizenz „infektiös“, d.h. wie sind die Regeln für die Einbindung in ein größeres System?
- Welche Rechte bestehen für die Weiterverbreitung, d. h. ist die Weitergabe des Quellcodes vorgeschrieben oder gibt es eine Klausel zur Bekanntmachung?

- Wie eindeutig ist die Herkunft der Dateien im Paket?

Dies ist keine erschöpfende Liste von Kriterien, die zu beachten sind. Darüber hinaus hängt die Frage, ob bestimmte Lizenzbedingungen akzeptabel sind oder nicht, vom jeweiligen Anwendungsfall ab.

Vorgelagerte Qualitätsbewertung

Wenn die Lizenzprüfung keine Bedenken ergeben hat, wird im nächsten Schritt die Qualität der vorgelagerten Prozesse bewertet. Für die Bewertung des vorgelagerten Entwicklungsprozesses können verschiedene Kriterien herangezogen werden:

1. Verbreitung: Wo und wie oft wird das Softwarepaket verwendet?
    a. Ist es als Paket in einer größeren Linux-Distribution enthalten?
    b. Wie oft wurde es installiert? (Siehe z. B. [21])
    c. Wie präsent ist die Komponente in der Architektur der wichtigsten Linux-Distributionen?
2. Releases: Gibt es einen Release-Prozess und wie funktioniert dieser?
    a. Gibt es einen Freigabeprozess? Wie ist die Qualität dieses Prozesses, z. B. im Hinblick auf Wiederveröffentlichungstests, Versionshinweise mit Änderungsprotokoll, Versionierung usw.?
    b. Wie viele Releases wurden bisher veröffentlicht?
    c. Gibt es einen Prozess zur Meldung von Fehlern, gibt es Fehlerlisten?
3. Entwicklung: Wie wird die Software entwickelt?
    a. Gibt es Kodierungsrichtlinien und werden diese eingehalten?
    b. Sind Tests vorhanden? Wie hoch ist deren Codeabdeckung?
    c. Gibt es Einschränkungen für Mitwirkende bei der Übergabe an den Hauptentwicklungszweig?
    d. Wie sehen die Kontrollmechanismen aus, z. B. die erforderliche Überprüfung und Freigabe?
    e. Gibt es ein Build-System?
    f. Wie sieht die Branching-Strategie aus?
4. Beitrag: Wie aktiv und offen ist die Gemeinschaft?
    a. Wie viele Mitwirkende können aufgelistet werden?
    b. Ist es möglich, Korrekturen oder Erweiterungen beizusteuern?
    c. Wie aktiv ist die Entwicklung, d. h. Beiträge im Laufe der Zeit, Art der Beiträge (Fehlerbehebungen, Erweiterungen), Anzahl der aktiven Entwickler usw.?
5. Sicherheit: Wie wird mit Sicherheitsfragen umgegangen?
    a. Wie werden Sicherheitsvorfälle der vorgelagerten Entwicklung behandelt?

b. Gibt die CVE-Historie (die von der Größe und Exposition der Software abhängt) einen Hinweis?

Für das Projekt, in das die Komponente integriert werden soll, können Kriterienlisten zusammen mit einem Bewertungsschema erstellt werden. Diese können, müssen aber nicht notwendigerweise in eine numerische, automatisiert auswertbare Einstufung münden, werden auf jeden Fall durch Expertenbegutachtung ausgewertet.

Generell ist zu beobachten, dass qualitativ hochwertige OSS-Komponenten sich oft strenger an Kodierungs- und Entwicklungsrichtlinien halten als viele kommerzielle Software. Dies kann folgende Gründe haben:

- die Beiträge zum Projekt sind öffentlich und identifizierbar,
- der Gruppendruck innerhalb der Entwicklungsgruppe,
- die Regeln für die Beiträge zum Projekt sind einfacher als bei vielen Prozessen in der Automobilindustrie und werden mit Begründungen versehen.

Qualifikation der Komponenten

Die zuvor beschriebene vorgelagerte Qualitätsbewertung ist eine erste Qualitätsanalyse, ob eine Komponente für den Einbau in ein Projekt geeignet ist. In einem zweiten Schritt muss die Komponente noch entsprechend ihres Verwendungszwecks qualifiziert werden. Da eine integrierte 3rd-Party-Komponente nicht zwangsläufig vollständig dem geforderten Funktionsumfang entspricht, d.h. die Komponente bietet entweder zu viel oder zu wenig Funktionalität, hat dies Auswirkungen auf die Qualifizierungsstrategie. Während ein zu geringer Funktionsumfang durch Hinzufügen der fehlenden Funktionen in die Integrationsumgebung behoben werden kann, kann ein zu hoher Funktionsumfang zu einem höheren Qualifizierungsaufwand als geplant führen. Daher empfiehlt es sich, bei der Qualifizierung zwischen einer *generischen* und einer *Use-Case-basierten* Qualifizierung zu unterscheiden.

Generische Qualifikation

Bei der generischen Qualifizierung wird der gesamte Funktionsumfang der Komponente betrachtet, unabhängig davon, wo und wie sie integriert wird. Dies bedeutet, dass jeder Aspekt der Software analysiert und verifiziert werden muss. Die Stringenz der Verifikation kann je nach Verwendungszweck variieren, wie in den vorangegangenen Abschnitten erläutert, aber im Allgemeinen ist es nicht möglich, den Funktionsumfang zu reduzieren, da nicht spezifiziert ist, wie die Komponente verwendet wird. Diese Qualifizierungsart ist die vollständigste, aber auch die mit dem höchsten Aufwand.

Use-Case-basierte Qualifizierung

Im Gegensatz dazu wird bei der Use-Case-basierten Qualifizierung berücksichtigt, wie und für welchen Zweck die Komponente eingesetzt werden soll. Die Use-Cases lassen sich mit Standardverfahren spezifizieren und verifizieren. Diese Anwendungsfälle können

auch auf Systemebene spezifiziert werden, wobei dann die Systemebene mit der erforderlichen Genauigkeit verifiziert wird.

Bei sorgfältiger Durchführung hat dieser Ansatz den Vorteil, dass der Aufwand für die Qualifizierung der integrierten Softwarekomponente reduziert wird, ohne die Qualität des Gesamtsystems wesentlich zu beeinträchtigen, insbesondere wenn die integrierte Komponente mit großer Sorgfalt ausgewählt wurde (siehe vorheriger Abschnitt). Sie vermeidet die Qualifizierung von Teilen der Komponente, die im System nicht verwendet werden, und konzentriert sich auf die Systemfunktionalität.

Nachteile bestehen ebenfalls: Die Qualifizierung ist nicht übertragbar, sie gilt nur für die Integration in eine bestimmte Umgebung. Auch wenn sich die Umgebung ändert, muss die Qualifikation neu bewertet werden, wenn sie überhaupt noch gültig ist. Sie ist für die Sicherheitsqualifizierung kaum geeignet und steht daher nur für QM und ergänzende Funktionalität im System zur Verfügung.

Dennoch ist dies ein gängiger Ansatz, um große Linux-Distributionen für bestimmte Anwendungsfälle zu qualifizieren, z. B. um große Datenbanken bestimmter Hersteller in Rechenzentren zu hosten. Bei einem solchen zertifizierten System werden Fehlfunktionen sofort als Fehler betrachtet und entsprechend behandelt. Fehlfunktionen, die nicht mit den zertifizierten Anwendungsfällen zusammenhängen, werden natürlich gesammelt, aber nicht unbedingt mit der gleichen Akribie behandelt.

5 Reifegrad der Organisation in Bezug auf Open Source

Ein wichtiger Aspekt, der bei der Einbindung von OSS-Software von Drittanbietern zu berücksichtigen ist, ist das Maß an organisatorischer Reife für dieses Vorhaben. Wie in den vorangegangenen Abschnitten gezeigt wurde, erfolgt die Nutzung von OSS nicht einfach durch die Einbindung einer OSS-Komponente in das eigene Projekt. Der Umgang mit diesen Themen kann mit organisatorischen Reifegraden angegangen werden.

In „Open Source Reifegrade - Leitfaden für gemeinsame Standards“ [22] hat der ZVEI fünf organisatorische Reifegrade für den Umgang mit Open Source Software zusammengefasst:

- **Stufe 1: kein allgemeiner Ansatz.** Die Nutzung von OSS wird von Fall zu Fall ohne allgemeine Richtlinien gehandhabt.
- **Stufe 2: Nicht-Nutzung.** Eine allgemeine Unternehmensregel verbietet die Verwendung von OSS-Komponenten in jedem Projekt.
- **Stufe 3: Aktive und defensive Nutzung.** Die Einbeziehung von OSS-Komponenten kann von Kunden gefordert oder in begrenzten Fällen für ein bestimmtes Projekt beschlossen werden. Ein organisierter Ansatz ist vorhanden.

- **Stufe 4: Beitrag.** OSS-Komponenten werden nicht nur genutzt, sondern es findet ein aktiver Beitrag in OSS-Projekten statt.
- **Stufe 5: Führende Rolle.** Das Unternehmen spielt eine führende Rolle bei der Entwicklung von OSS-Komponenten, die direkt oder indirekt zum Einkommensstrom des Unternehmens beitragen.

Im Allgemeinen wird von Stufe 1 für Automotive-Anwendungsfälle abgeraten, da kein Ansatz bedeutet, dass man sich nicht ausreichend kümmert und jeden Fall und jedes Projekt, das Artefakte aus einem früheren Projekt wiederverwendet, einem Risiko aussetzt. [22].

6 Zusammenfassung und Ausblick

Ein risikobasierter Ansatz kann verwendet werden, um die Qualifikationsaktivitäten auf die Komponenten mit den höchsten Risiken zu konzentrieren. Einerseits ergeben sich die Risiken bei der Verwendung einer Komponente eines Drittanbieters aus dem Anwendungsbereich, z. B., ob es sich um eine sicherheitsrelevante Zielkomponente oder ein Entwicklungswerkzeug handelt. Andererseits können die Risiken aus der Qualität der integrierten Software selbst resultieren, die bewertet werden muss und deren potenzielle Schwachstellen entschärft werden müssen. Beide Metriken bilden eine Matrix, die dann die Gesamtkritikalität der Software zeigt.

Der risikobasierte Ansatz ermöglicht es, die Qualifizierungsbemühungen auf kritische Komponenten zu konzentrieren. Anstatt alle Komponenten gleich zu behandeln und damit einen hohen Qualifizierungsaufwand für Software mit geringer Kritikalität zu betreiben, kann der Qualifizierungsaufwand durch die Identifizierung der hochkritischen Komponenten zu Beginn reduziert werden.

Die Verwendung von Softwarekomponenten von Drittanbietern, insbesondere von Open-Source-Bibliotheken und -Software, im Automobilkontext ist in der Praxis möglich, muss jedoch gut überlegt sein, um erfolgreich sein zu können. Eine blinde Anwendung etablierter Verfahren der automobilen Systementwicklung auf große Fremdkomponenten würde die Entwicklung zum Stillstand bringen. Dennoch ist es notwendig, die Verantwortung für ein qualitativ hochwertiges, automobilgerechtes Softwaresystem bewusst wahrzunehmen und umzusetzen, und es bedarf effektiver Ansätze zur Berücksichtigung und Einbeziehung solcher Softwarekomponenten.

Wir ermutigen Automobilhersteller und -zulieferer, sehr bewusst mit bestehenden Komponenten von Drittanbietern zu verfahren, insbesondere mit Open-Source-Komponenten, und erst recht im Linux-Kontext. Die Autoren stehen für weitergehende Diskussionen zur Verfügung.

7 References

[1] VDA QMC Working Group 13 / Automotive SIG, Automotive SPICE Process Reference Model / Process Assessment Model, 3.1 Hrsg., VDA Quality Management Center, 2017.

[2] N. Yılmaz und A. K. Tarhan, „Matching terms of quality models and meta-models: toward a unified meta-model of OSS quality," *Software Quality Journal*, 17 12 2022.

[3] M. P. Barcellos, R. d. A. Falbo und R. D. Moro, „A Well-Founded Software Measurement Ontology," in *Formal Ontology in Information Systems, Proceedings of the Sixth International Conference, FOIS 2010*, Toronto, Canada, 2010.

[4] International Standardization Organization, „ISO/IEC 15504-5:2012 Information technology - Process assessment - Part 5: An exemplar software life cycle process assessment model," 2012.

[5] International Standardization Organization, „ISO/IEC 33001:2015 Information technology — Process assessment — Concepts and terminology," 2015.

[6] S. Keßler, Anpassung von Open-Source-Software in Anwenderunternehmen, Wiesbaden: Springer Vieweg, 2013.

[7] S. Höhne, M. Meyer, R. Jochem und H. Pohlheim, „Automatisierte Maßnahmenempfehlung zur Erhöhung der Qualität in Softwareentwicklungsprozessen," in *Trends und Entwicklungstendenzen im Qualitätsmanagement: Bericht zur GQW-Jahrestagung 2021 in Cottbus*, Wiesbaden, Springer Vieweg, 2022, pp. 134-154.

[8] International Organization for Standardization, „ISO 26262-1:2018 Road vehicles — Functional safety — Part 8: Supporting processes," ISO Copyright Office, Geneva, 2018.

[9] J. Spaans, „LKML.org - The Linux Kernel Mailing List Archive," [Online]. Available: https://lkml.org/. [Zugriff am 04 08 2022].

[10] The kernel development community, „A guide to the Kernel Development Process," [Online]. Available: https://www.kernel.org/doc/html/latest/process/development-process.html. [Zugriff am 04 08 2022].

[11] Mozilla.org, „Mozilla Release Processes," [Online]. Available: https://mozilla.github.io/process-releases/. [Zugriff am 04 08 2022].

[12] S. Fillenberg, „Fahrzeugserver von Continental vernetzt VW ID. Elektrofahrzeuge," Continental AG, 19 11 2019. [Online]. Available: https://www.continental.com/de/presse/pressemitteilungen/2019-11-12-icas-vw.

[13] Elektrobit, „Revolutionized vehicle infrastructure architecture,“ 25 11 2021. [Online]. Available: https://www.elektrobit.com/success-stories/revolutionized-vehicle-infrastructure-architecture/.

[14] Automobilwoche, „Conti liefert zentralen Rechner für Elektroauto-Reihe,“ 12 11 2019. [Online]. Available: https://www.automobilwoche.de/nachrichten/conti-liefert-zentralen-rechner-fur-elektroauto-reihe.

[15] K. Beck, M. Beedle, A. v. Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland und D. Thomas, „Manifesto for Agile Software Development,“ 2001. [Online]. Available: https://agilemanifesto.org/.

[16] ISO/IEC JTC 1/SC 7 Software and systems engineering, "ISO/IEC 25010:2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models," International Organization for Standardization, Geneva, 2011.

[17] International Organization for Standardization, „ISO 26262-1:2018 Road vehicles — Functional safety — Part 1: Vocabulary,“ ISO Copyright Office, Geneva, 2018.

[18] The MITRE Corporation, „CVE - CVE-2021-44228,“ 26 11 2021. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2021-44228.

[19] M. Dölle, „Sicherheitslücke Log4Shell: Internet in Flammen,“ Heise Medien, 31 12 2021. [Online]. Available: https://www.heise.de/news/Sicherheitsluecke-Log4Shell-Internet-in-Flammen-6304730.html.

[20] D. Everson, L. Cheng und Z. Zhang, „Log4shell: Redefining the Web Attack Surface,“ in *Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb) 2022*, San Diego, CA, 2022.

[21] A. Pennarun, B. Allombert und P. Reinholdtsen, „Ubuntu Popularity Contest,“ Debian.org, [Online]. Available: https://popcon.ubuntu.com/. [Zugriff am 04 08 2022].

[22] A. Bühls, „Open Source Reifegrade - Leitfaden für gemeinsame Standards,“ ZVEI e.V., 06 2021. [Online]. Available: https://www.zvei.org/open-source-reifegrade-leitfaden-fuer-gemeinsame-standards.