Software-Engineering-Empfehlungen des DLR

Version 1.0.0

DLR

Änderungsstand

Version	Datum	Bemerkung
1.0.0	17.08.2018	Erste öffentliche Version.

Inhaltsverzeichnis

1 Einleitung

Dieses Dokument beschreibt die Software-Engineering-Empfehlungen des Deutschen Zentrums für Luft- und Raumfahrt e.V. (DLR). Die Zielgruppe der Empfehlungen sind Wissenschaftlerinnen und Wissenschaftler des DLR. Die Empfehlungen sollen sie unterstützen, ihre entwickelte Software in Bezug auf gute Software-Entwicklungs- und Dokumentationspraxis einzuschätzen und zu verbessern. Der Fokus der Empfehlungen liegt auf dem Wissenserhalt und der Förderung von nachhaltiger Software-Entwicklung in der Forschung.
Die Empfehlungen wurden in Zusammenarbeit mit den Mitgliedern des DLR Software-Engineering-Netzwerks entwickelt. Das Netzwerk ist das zentrale DLR-Austauschforum zum Thema Software-Engineering. Wir veröffentlichen die Empfehlungen, um die generelle Diskussion zum Thema gute wissenschaftliche Software-Entwicklungspraxis zu unterstützen.

1.1 Danksagung

Die Autoren möchten sich bei allen Beteiligten und insbesondere den Mitgliedern des DLR Software-Engineering-Netzwerks recht herzlich für ihre Beiträge bedanken. Zudem danken wir der zentralen IT-Abteilung des DLR für die kontinuierliche finanzielle Unterstützung für dieses wichtige Thema.

1.2 Weitergehende Informationen

Weitergehende Informationen zu den Empfehlungen und dem übergeordneten Konzept sind hier zu finden:

- T. Schlauch, C. Haupt, "Helping a friend out. Guidelines for better software", Second Conference of Research Software Engineers, September 2017. [Online]. Available: https://elib.dlr.de/114049/
- T. Schlauch, "Software engineering initiative of DLR: Supporting small development teams in science and engineering", ESA S/W Product Assurance and Engineering Workshop 2017, September 2017. [Online]. Available: https://elib.dlr.de/117717/
- C. Haupt, T. Schlauch, "The software engineering community at DLR: How we got where we are" in Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE5.1), N. C. Hong, S. Druskat, R. Haines, C. Jay, D. S. Katz, and S. Sufi, Eds., September 2017. [Online]. Available: https://elib.dlr.de/114050/
- C. Haupt, T. Schlauch, M. Meinel, "The software engineering initiative of DLR - overcome the obstacles and develop sustainable software" in 2018 ACM/IEEE International Workshop on Software Engineering for Science, June 2018. [Online]. Available: https://elib.dlr.de/120462/

1.3 Zitationshinweis

T. Schlauch, M. Meinel, C. Haupt, "Software-Engineering-Empfehlungen des DLR", Version 1.0.0, August 2018. [Online]. Available: https://doi.org/10.5281/zenodo.1344608

1.4 Lizenz

1.5 Englischsprachige Übersetzung

Die englischsprachige Übersetzung finden Sie unter: https://doi.org/10.5281/zenodo.1344612

2 Begriffe und Abkürzungen

2.1 Begriffe

Software	Unter dem Begriff Software versteht man i.Allg. Programme, die auf einem Computer oder ähnlichen Geräten ablaufen. Zur Software gehören neben dem Programm beispielsweise der Quelltext, die Nutzerdokumentation, Testdaten und das Architekturmodell.
Software mit Produktcharakter	Software, die in einem produktiven Kontext genutzt wird und funktionieren muss. Eventuell ist diese wesentlicher Bestandteil einer Kooperation mit anderen Organisationen.
Software-Verantwortlicher	Der bzw. die Software-Verantwortliche besitzt den technischen und fachlichen Überblick über eine Software. Bei Software mit geringem Umfang ist das i.d.R. der aktuelle Hauptentwicklerin bzw. Hauptentwickler.
an der Entwicklung Beteiligte	In diesem Dokument werden mit der Wortgruppe "an der Entwicklung Beteiligte" alle Personen bezeichnet, die direkt zur Entwicklung der Software beitragen. Das sind beispielsweise Software-Entwickler oder Tester. Alternativ wird auch der Begriff "Entwicklungsteam" verwendet.
SoftwareEngineering.Wiki	Das SoftwareEngineering.Wiki ist der zentrale DLR-interne Wiki-Bereich, um Informationen und Wissen zum Thema Software-Engineering auszutauschen.

2.2 Abkürzungen

AK	Anwendungsklasse
EQA	Empfehlung "Qualifizierung"
EAM	Empfehlung "Anforderungsmanagement"
ESA	Empfehlung "Software-Architektur"
EÄM	Empfehlung "Änderungsmanagement"
EDI	Empfehlung "Design und Implementierung"
EST	Empfehlung "Software-Test"
ERM	Empfehlung "Release-Management"
EAA	Empfehlung "Automatisierung und Abhängigkeitsmanagement"

3 Anwendungsklassen

Die Anwendungsklassen (AK) helfen, geeignete Maßnahmen hinsichtlich der Software-Qualität festzulegen. Sie erlauben, Aktivitäten und Werkzeugeinsatz bedarfsgerecht zu gestalten, und strukturieren die Kommunikation der Beteiligten zum Thema Software-Qualität.
Die Anwendungsklassen definieren aufeinander aufbauend Empfehlungen, um eine angemessene Engineering-Praxis und Software-Qualität sicherzustellen. Sie adressieren primär den Investitionsschutz, die Minderung von Risiken und den Wissenserhalt. Die Maßnahmen, die dazu ergriffen werden, müssen sich an den Ansprüchen der Anwendungsklasse orientieren.
Die Anwendungsklassen unterstützen primär die Entwicklung individueller Software in der Einrichtung. Zusätzlich können sie als Basis für Vorgaben an extern beauftragte Unternehmen verwendet werden, um die Qualität der Entwicklung sicherzustellen. Dies ist insbesondere zu empfehlen, wenn die extern erstellte Software durch die Einrichtung später gepflegt oder weiterentwickelt werden soll.

3.1 Anwendungsklasse 0

Bei Software dieser Klasse steht der persönliche Gebrauch in Verbindung mit einem geringen Funktionsumfang im Vordergrund. Die Weitergabe der Software innerhalb und außerhalb des DLR ist nicht vorgesehen.
Software dieser Anwendungsklasse entsteht häufig bei der Lösung von Detailproblemen im Forschungsumfeld. Die jeweilige Einrichtung legt für diese Anwendungsklasse selbst die erforderlichen Maßnahmen fest, die beispielsweise zur Einhaltung der guten wissenschaftlichen Praxis erforderlich sind. Beispiele für eine mögliche Zuordnung zur Anwendungsklasse 0 sind:

- Skripte, um die Daten für eine Publikation aufzubereiten.
- Einfache administrative Skripte, um bestimmte Aufgaben zu automatisieren.
- Software, die nur bestimmte Funktionen demonstriert bzw. zum Austesten dieser entwickelt wird.

3.2 Anwendungsklasse 1

Bei Software dieser Klasse soll es möglich sein, dass an der Entwicklung Unbeteiligte diese im festgelegten Umfang nutzen und ihre Entwicklung fortsetzen können. Dies ist das anzustrebende Grundniveau, falls die Software über den persönlichen Gebrauch hinaus weiterentwickelt und genutzt werden soll.
Dazu muss der vorhandene Stand nachvollziehbar und reproduzierbar sein. Es ist notwendig, dass die grundlegenden Anforderungen und Randbedingungen, der zur Verfügung stehende Funktionsumfang sowie bekannte Probleme der Software ersichtlich sind.
Diese Anwendungsklasse ist zu empfehlen, wenn die Software kein breites Spektrum an Funktionen bietet oder die Einrichtung diese nur in einem engbegrenzten Rahmen weiterentwickelt. Beispiele für eine mögliche Zuordnung zur Anwendungsklasse 1 sind:

- Software, die Studierende in Studien-, Bachelor- oder Masterarbeiten entwickeln.
- Software aus Dissertationen, bei der die längerfristige Weiterentwicklung keine Rolle spielt.
- Software aus Drittmittelprojekten, wobei die Software vorerst reinen Demonstrationscharakter besitzt, ohne dass eine längerfristige Weiterentwicklung vorgesehen ist.

3.3 Anwendungsklasse 2

Bei Software dieser Klasse soll eine langfristige Weiterentwicklung und Wartbarkeit sichergestellt werden. Dies ist die Basis für einen Übergang in den Produktstatus.
Dazu ist der strukturierte Umgang mit den jeweiligen Anforderungen erforderlich. Insbesondere sind die Randbedingungen und qualitativen Anforderungen in einer angemessenen Software-Architektur zu adressieren. Diese beschreibt die technischen Konzepte und den Aufbau der Software, sichert das Entwicklungs-Know-how und erlaubt, die Eignung für neue Nutzungsszenarien einzuschätzen. Weiterhin sind in diesem Zusammenhang ein definierter Entwicklungsablauf, Regeln für Design und Kodierung sowie der Einsatz von Testautomatisierung unerlässlich.
Diese Anwendungsklasse ist zu empfehlen, wenn die Software ein breites Spektrum an Funktionen bietet und die Einrichtung diese längerfristig weiterentwickelt. Beispiele für eine mögliche Zuordnung zur Anwendungsklasse 2 sind:

- Software aus Dissertationen, bei der Wartbarkeit und längerfristige Verwendbarkeit eine wichtige Rolle spielen.
- Software aus Drittmittelprojekten, bei der Wartbarkeit und längerfristige Verwendbarkeit über das Projekt hinaus eine wesentliche Rolle spielen.
- Umfangreiche Forschungs-Frameworks, die ein Großteil einer Abteilung mit entwickelt (ohne Produktcharakter).

3.4 Anwendungsklasse 3

Bei Software dieser Klasse ist es essentiell, Fehler zu vermeiden und Risiken zu mindern. Dies betrifft insbesondere kritische Software und solche mit Produktcharakter.
Dazu ist ein aktives Risikomanagement durchzuführen. D.h., Risiken der technischen Lösung sind von Beginn an aktiv zu identifizieren und in der Software-Architektur zu adressieren. Zudem sollen durch den weiteren Ausbau der Testautomatisierung und strukturierte Reviews Fehler frühzeitig erkannt werden, um ihren Eingang in eine Produktivversion möglichst zu verhindern. Weiterhin ist eine Nachvollziehbarkeit von Änderungen sicherzustellen.
Diese Anwendungsklasse ist zu empfehlen, wenn für die Einrichtung hohe Risiken hinsichtlich der Entwicklung vorliegen. Diese können beispielsweise aus Produkthaftungs-, Zertifizierungsgründen, externen Vorgaben oder aus der Bedeutung der Software für die wertschöpfenden Aktivitäten entstehen. Beispiele für eine mögliche Zuordnung zur Anwendungsklasse 3 sind:

- Missionskritische Software z.B. im Kontext von Fluggeräten, autonomen Fahrzeugen oder Raumfahrtmissionen.
- Software, für die die Einrichtung eine Gewährleistung innerhalb oder außerhalb des DLR (z.B. über eine externe Firma) übernimmt.

- Software, die einen wesentlichen Beitrag zur Generierung von Drittmitteln und Forschungsergebnissen für die Einrichtung leistet und deshalb zuverlässig funktionieren muss.

3.5 Festlegung der angestrebten Anwendungsklasse

Zu Beginn der Entwicklung legt der Software-Verantwortliche ggf. zusammen mit weiteren fachlich Beteiligten die angestrebte Anwendungsklasse fest. Zudem wird regelmäßig geprüft, ob die Zuordnung zu der angestrebten Anwendungsklasse anzupassen ist.
Die Entscheidungskriterien zur Zuordnung der Anwendungsklasse orientieren sich direkt an den Zielen, die die jeweilige Anwendungsklasse verfolgt. Aus den Kriterien ergibt sich ein Entscheidungsbaum (vgl. Abbildung 1). Dieser stellt lediglich eine Empfehlung dar, von der begründet abgewichen werden kann.
Im Folgenden wird die Zuordnung zu einer Anwendungsklasse anhand der Kriterien erläutert:

1. **Risiken für die Einrichtung:** Dies stellt das erste und wesentliche Entscheidungskriterium dar. Hohe Risiken für die Einrichtung können beispielsweise aus Produkthaftungs-, Zertifizierungsgründen, externen Vorgaben oder aus der Bedeutung der Software für die wertschöpfenden Aktivitäten entstehen. Ein "Ausfall" der Software könnte somit empfindliche Einschnitte für die Einrichtung oder einen Teil davon zur Folge haben. Deshalb ist in Fällen mit hohem Risiko, unabhängig von den weiteren Kriterien, eine Zuordnung zur Anwendungsklasse 3 anzustreben.
2. **Umfang:** Das nächste Kriterium ist der erwartete Umfang. Bei einem geringen Umfang ist Anwendungsklasse 0 bzw. 1 ausreichend. Dies gilt auch, wenn eine längere Nutzung und Weiterentwicklung der Software vorgesehen ist. Eine objektive Einschätzung eines geringen Umfangs ist schwierig. Metriken wie "Anzahl der Quelltextzeilen" lassen sich nur bedingt mit dem Umfang korrelieren. Daher ist zu empfehlen, den Gesamtaufwand der Entwicklung zu begrenzen. Bei einem geringen Umfang sollte der Aufwand zur Implementierung der Software (inklusive der Umsetzung der Empfehlungen der Anwendungsklasse 1) ein Personenjahr nicht überschreiten.
3. **Weitergabe der Software:** Dieses Kriterium bezieht sich auf eine Weitergabe der Software innerhalb und außerhalb des DLR. Insbesondere wenn die Software an Dritte außerhalb des DLR weitergegeben wird, sind i.d.R. Lizensierungsaspekte zu beachten. Deshalb ist in diesen Fällen mindestens die Anwendungsklasse 1 zu wählen. Wenn die Software nicht durch andere Kollegen genutzt wird, ist die Zuordnung zur Anwendungsklasse 0 ausreichend.
4. **Zeitraum der Weiterentwicklung:** Dieses Kriterium bezieht sich auf den erwarteten Zeitraum, in dem die Einrichtung die Software weiterentwickelt und pflegt. Falls ein hoher Funktionsumfang vorliegt und über einen längeren Zeitraum die Weiterentwicklung sichergestellt sein muss, ist die Anwendungsklasse 2 anzustreben. Es besteht in diesem Fall ein erhöhter Bedarf, dem Know-how-Verlust entgegen zu wirken. Ein längerer Weiterentwicklungszeitraum liegt i.d.R. vor, wenn die Weiterentwicklung auch nach dem möglichen Ausscheiden wichtiger Know-how-Träger (> 2 Jahre) notwendig ist.

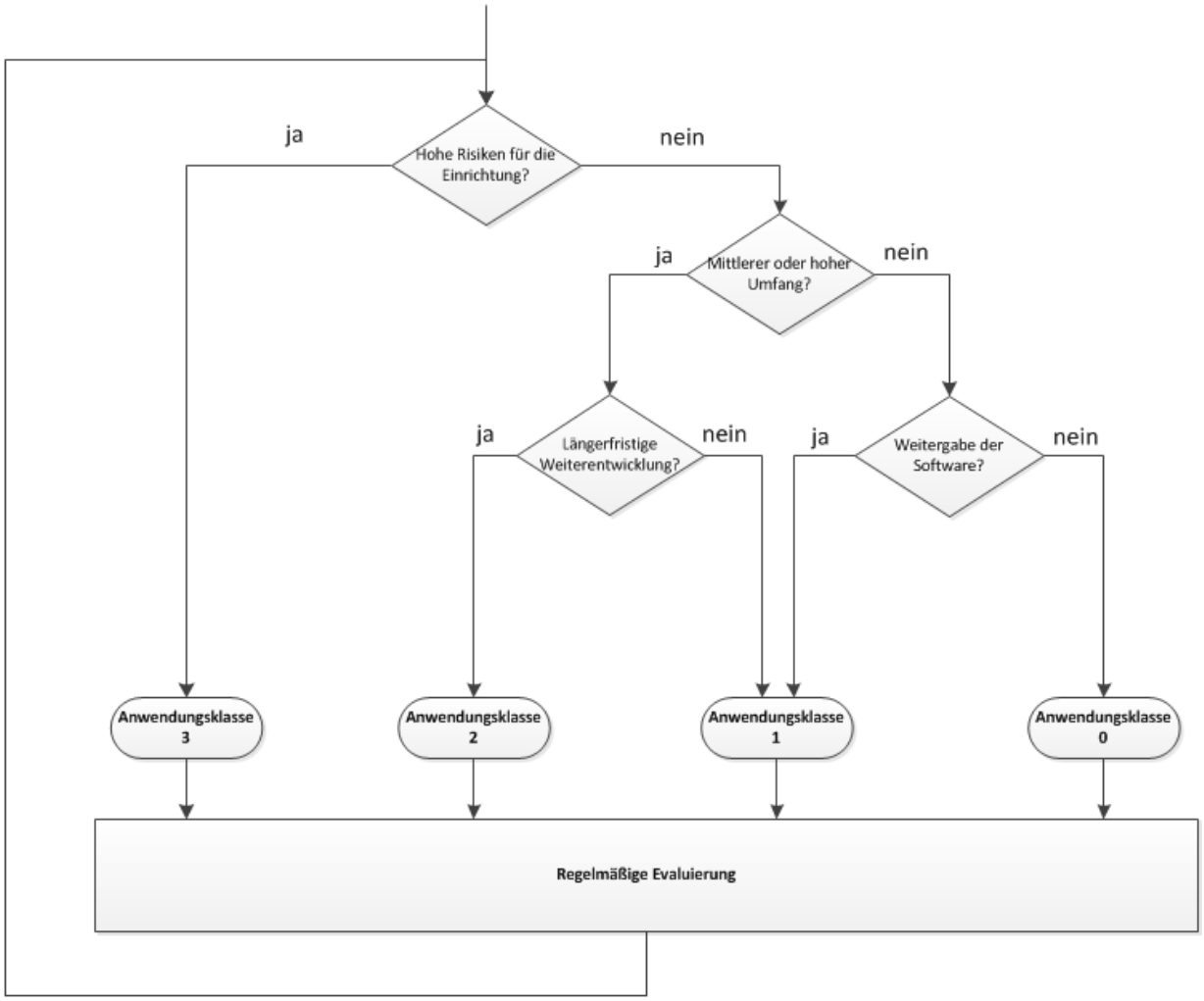

Abbildung 1: Entscheidungsbaum zur Bestimmung der angestrebten Anwendungsklasse

3.6 Prüfung der erreichten Anwendungsklasse

Auf Basis der angestrebten Anwendungsklasse schätzt der Software-Verantwortliche die erreichte Anwendungsklasse regelmäßig ein. Dazu ist festzustellen inwieweit die Empfehlungen der jeweiligen Anwendungsklasse umgesetzt werden. Zu diesem Zweck werden ergänzend zu diesem Dokument Checklisten für die Anwendungsklassen 1 - 3 in verschiedenen Formaten bereitgestellt. Sie listen alle für eine Anwendungsklasse relevanten Empfehlungen auf. Es kann aber sinnvoll sein, bereits Empfehlungen einer höheren Anwendungsklasse – zumindest in abgeschwächter Form – umzusetzen. Dies vereinfacht den Übergang zu einer höheren Anwendungsklasse. Der nächste Abschnitt bietet einen detaillierten Überblick zu allen Empfehlungen inklusive Erläuterungen und weiterführenden Informationen.
Die Empfehlungen sind im Kontext der jeweiligen Einrichtung zu interpretieren und zu bewerten. Falls die angestrebte Anwendungsklasse nicht erreicht wird, legt der Software-Verantwortliche ggf. zusammen mit weiteren fachlich Beteiligten geeignete Maßnahmen für die weitere Entwick-

lung fest. In diesem Zusammenhang muss das Kosten-Nutzen-Verhältnis in Anbetracht der verbleibenden Entwicklungszeit und -ressourcen realistisch betrachtet werden.

4 Übersicht der Empfehlungen

Dieses Kapitel beschreibt die Empfehlungen für verschiedene Bereiche der Software-Entwicklung. Zu Beginn jedes Bereichs werden die Inhalte zusammenfassend dargestellt und die wesentlichen Begriffe eingeführt. Anschließend folgen die Empfehlungen inklusive einer Erläuterung. Zu jeder Empfehlung ist vermerkt, ab welcher Anwendungsklasse diese gilt (vgl. Abschnitt Anwendungsklassen). Die Sortierung der Empfehlungen orientiert sich an dem Aufbau des einleitenden Abschnitttextes.
Die Umsetzung der einzelnen Empfehlungen ist bewusst offen gehalten, um eine möglichst optimale Entscheidung in Abhängigkeit vom jeweiligen Entwicklungskontext zu erlauben. Es finden sich jedoch i.d.R. erste Hinweise dazu im Erläuterungstext. Das *SoftwareEngineering.Wiki* liefert darüber hinaus weiterführende Informationen, Werkzeug- und Literaturempfehlungen sowie konkrete Beispiele zu den verschiedenen Software-Engineering-Themen.

4.1 Qualifizierung

Die folgenden Empfehlungen sollen sicherstellen, dass die an der Entwicklung Beteiligten das notwendige Wissen und Training besitzen. Es ist zu empfehlen, bestehende Lücken durch Schulungen zu schließen. Dies ist u.a. bei der Erstellung der individuellen Schulungspläne zu berücksichtigen.

Empfehlung	ab AK	Erläuterung
EQA.1: Der Software-Verantwortliche kennt die verschiedenen Anwendungsklassen und weiß, welche für seine Software anzustreben ist.	1	Dieses Wissen ist die Voraussetzung, um die im DLR empfohlenen Maßnahmen zur Sicherstellung guter Engineering-Praxis und der Software-Qualität umsetzen zu können.
EQA.2: Der Software-Verantwortliche weiß, wie er gezielt Unterstützung zu Beginn und im Verlauf der Entwicklung anfordern und sich mit anderen Kollegen zum Thema Software-Entwicklung austauschen kann.	1	Die Kenntnis weiterer Ansprechpartner zum Thema Software-Engineering ist wichtig, um Probleme zu Beginn und während der Entwicklung leichter beheben zu können.
EQA.3: Die an der Entwicklung Beteiligten ermitteln den Qualifikationsbedarf in Bezug auf ihre Rolle und die angestrebte Anwendungsklasse. Sie kommunizieren diesen Bedarf an den Vorgesetzten.	1	Auf Ebene der Anwendungsklasse 1 sind neben dem fachlichen Wissen zumindest die Kenntnis der Programmiersprache und eines Versionskontrollsystems erforderlich. Auf den höheren Anwendungsklassen sind ggf. weitere Fähigkeiten auf Ebene des Teams notwendig. Diese müssen zielgerichtet aufgebaut bzw. geschult werden.

EQA.4: Den an der Entwicklung Beteiligten stehen die für ihre Aufgaben benötigten Werkzeuge zur Verfügung und sie sind geschult in deren Benutzung.	1	Insbesondere müssen sie den Umgang mit den Werkzeugen im jeweiligen Nutzungsumfang beherrschen. Anderenfalls sind unnötige Aufwände und Nacharbeiten durch Missverständnisse wahrscheinlich.

4.2 Anforderungsmanagement

Zentraler Einstiegspunkt in das Anforderungsmanagement bildet die **Aufgabenstellung**. Sie beschreibt in knapper und verständlicher Form die Ziele und den Zweck der Software. Zudem fasst sie die wesentlichen Anforderungen zusammen. Sie legt somit "das Warum" und "das Was" fest und dient bei Entscheidungen als Orientierungshilfe.
Eine **Anforderung** beschreibt eine zu erfüllende Eigenschaft in der Software. Es existieren verschiedene Anforderungstypen:

- **Funktionale Anforderungen** beschreiben gewünschte Funktionen der Software.
- **Qualitätsanforderungen** beschreiben erwartete qualitative Eigenschaften der Software (z.B. Nutzbarkeit, Sicherheit, Effizienz, vgl. ISO/IEC 25010).
- **Randbedingungen** beschreiben Beschränkungen, die bei der Entwicklung und Auslegung der Software zu beachten sind.

Anforderungen geben die Richtung der Software-Entwicklung vor. Insbesondere Qualitätsanforderungen prägen den resultierenden Lösungsansatz und beinhalten häufig Risiken. Deshalb sollten Qualitätsanforderungen möglichst frühzeitig abgestimmt und analysiert werden. Die gewünschte **Produktqualität der Software** muss stets explizit konstruiert werden und ist für jede Software individuell ausgeprägt.

Empfehlung	ab AK	Erläuterung
EAM.1: Die Aufgabenstellung ist mit allen Beteiligten abgestimmt und dokumentiert. Sie beschreibt in knapper, verständlicher Form die Ziele, den Zweck der Software, die wesentlichen Anforderungen und die angestrebte Anwendungsklasse.	1	Es ist wichtig, dass die Aufgabenstellung zwischen den Beteiligten möglichst früh abgestimmt ist, um Missverständnissen und Fehlentwicklungen vorzubeugen. Auch für eine spätere Nutzung und Weiterentwicklung liefert die Aufgabenstellung wichtige Hinweise.
EAM.2: Funktionale Anforderungen sind zumindest mit eindeutiger Kennung, Beschreibung, Priorität, Ursprung und Ansprechpartner erfasst.	2	Anforderungen müssen eindeutig identifizierbar sein, um sie während der Entwicklung referenzieren und an der Software durchgeführte Änderungen (vgl. Abschnitt Änderungsmanagement) auf sie zurückführen zu können. Zudem hilft eine Priorisierung, die Reihenfolge der Abarbeitung festzulegen. Schließlich sind Informationen zum Ansprechpartner und Ursprung essentiell bei Rück-

		fragen.
EAM.3: Die Randbedingungen sind erfasst.	1	Die zu beachtenden Randbedingungen (z.B. die zu verwendenden Programmiersprachen und Frameworks, die zu unterstützende Betriebsumgebung, rechtliche Aspekte) sind frühzeitig abzustimmen, um Missverständnisse und Fehlentwicklungen zu vermeiden. Zudem erlauben sie begründet Entscheidungen zu treffen, da sie helfen Optionen auszuschließen. Im Fall von Software mit geringem Umfang ist zu empfehlen, die Randbedingungen innerhalb der Aufgabenstellung festzuhalten (vgl. Empfehlung EAM.1).
EAM.4: Die Qualitätsanforderungen sind erfasst und priorisiert.	2	Die Produktqualität jeder Software ist individuell zu betrachten. Die relevanten Qualitätsanforderungen sind möglichst frühzeitig abzustimmen und festzulegen. Insbesondere ist eine Priorisierung notwendig, da Qualitätseigenschaften z.T. entgegen wirken. Ein gutes Verständnis der Qualitätsanforderungen ist essentiell, um Missverständnisse und Fehlentwicklungen zu vermeiden. Vergessene und fehlende Qualitätsaspekte haben häufig umfangreiche Änderungen der Software zur Folge. In der Praxis ist es hilfreich, Qualitätsanforderungen durch Szenarien zu konkretisieren. Ein solches Qualitätsszenario beschreibt ein typisches Nutzungsszenario der Software, wobei der Fokus auf einem Qualitätsmerkmal liegt (z.B. "Das System zeigt nach spätestes einer Sekunde erste Suchergebnisse an."). Dadurch werden geforderte Qualitätseigenschaften verständlicher, diskutierbar und überprüfbar.
EAM.5: Nutzergruppen und deren Aufgaben sind im jeweiligen Nutzungskontext erfasst.	2	Diese Analyse ist essentiell, um Verständnis für die Anwender der Software aufzubauen und um eine angemessene Lösung zu erstellen. Ohne diese Analyse ist eine mangelhafte Akzeptanz der Software wahrscheinlich bzw. es wird Aufwand in Funktionen gesteckt, die in der realisierten Form nicht genutzt werden.

EAM.6: Es wird aktives Risikomanagement betrieben. Die sich aus der Entwicklung ergebenden Risiken sind mit der Eintrittswahrscheinlichkeit und den erwarteten Auswirkungen erfasst.	3	Risiken entstehen beispielsweise aus unklaren, unverstandenen Anforderungen und damit verbundenen späten Änderungen. Zudem können fehlendes Know-how bzgl. einer einzusetzenden Technologie oder die Technologie selbst Risiken bergen. In der Folge können Terminverzögerungen und zusätzliche Aufwände entstehen. Deshalb ist es wichtig, Risiken aktiv zu ermitteln, zu verfolgen und geeignete Gegenmaßnahmen (z.B. Erstellung von Prototypen, Know-how-Aufbau) zu ergreifen. Insbesondere ist auf Risiken in Bezug auf die Software-Architektur (vgl. Abschnitt Software-Architektur) zu achten, da diese potentiell hohen Schaden verursachen können.
EAM.7: Vorgaben zur Formulierung und Dokumentation von Anforderungen sind festgelegt und werden konsequent angewendet.	3	Diese Vorgaben stellen sicher, dass alle wesentlichen Informationen konsistent erfasst werden, und tragen zur Fehlervermeidung bei.
EAM.8: Es existiert ein Glossar, der die wesentlichen Begrifflichkeiten und Definitionen beschreibt.	2	Das Glossar legt ein gemeinsames Vokabular fest. Es dient zur Vermeidung von Missverständnissen und Fehlern, die auf unterschiedlichen Begrifflichkeiten und Definitionen beruhen.
EAM.9: Die Anforderungsliste wird regelmäßig abgestimmt, aktualisiert, analysiert und geprüft. Die daraus resultierenden Änderungen lassen sich nachvollziehen.	2	Anforderungsbezogene Tätigkeiten finden wiederholt – insbesondere vor dem Beginn einer neuen Entwicklungsetappe – statt. In diesem Zusammenhang ist es wichtig, dass die Beteiligten sich abstimmen, um ein gemeinsames Verständnis für die nächsten Schritte zu entwickeln und um die Anforderungen weiter zu verfeinern. Dadurch lassen sich Missverständnisse und Fehlentwicklungen vermeiden, die auf nicht berücksichtige Anforderungen oder Teilen von diesen beruhen. Zudem erhält man eine in sich konsistente Anforderungsliste und kann Widersprüche in den Anforderungen erkennen bzw. auflösen.
EAM.10: Zu jeder Anforderung lassen sich die durchgeführten Änderungen an allen Bestandteilen der Software (z.B. Quelltext, Testfälle) nachvollziehen (Traceability).	3	Durch diese Maßnahme können die Auswirkungen einer Anforderung nachvollzogen werden. Es ist beispielsweise möglich zu erkennen, dass zu einer Anforderung die erforderliche Umsetzung inklusive Testfällen existiert. Weiterhin können

		auftretende Fehler leichter eingegrenzt und zugeordnet werden.

4.3 Software-Architektur

Die **Software-Architektur** vermittelt eine Idee der Kernbestandteile der Software, auf denen der Rest der Software aufbaut. Eine Änderung der Kernbestandteile ist teuer und qualitätsgefährdend.
Dazu das folgende Beispiel: Es ist ein verteiltes System zu realisieren. Aus den Qualitätsanforderungen geht hervor, dass die Komponenten verschlüsselt kommunizieren müssen (Sicherheit). Zudem muss sichergestellt werden, dass keine Nachrichten verlorengehen (Zuverlässigkeit). Weiterhin schränken die Randbedingungen die Lizenzwahl für Fremdsoftware ein. In diesem Zusammenhang ist die einzusetzende Kommunikationstechnologie ein solcher Kernbestandteil. Eine falsche Entscheidung kann dazu führen, dass die Anforderungen bzgl. Sicherheit und Zuverlässigkeit nicht erreicht werden. Zudem bringt eine nachträgliche Änderung der Technologie einen potentiell hohen Anpassungsaufwand mit sich. Der Entscheidungsprozess kann deshalb aufwendig ausfallen. Verschiedene Optionen sind zu ermitteln, es ist Wissen zu den Lösungsalternativen aufzubauen und es sind ggf. Prototypen zu erstellen. Randbedingungen wie die Einschränkung der Lizenzauswahl helfen dabei, bestimmte Optionen ausschließen zu können.
Es ist zu erkennen, dass die Software-Architektur sich systematisch aus den Anforderungen ableiten lässt. Insbesondere aus den Qualitätsanforderungen ergeben sich häufig architektonische Fragen. Dabei erleichtern Randbedingungen das Treffen angemessener Entscheidungen. Daher ist der konkrete Architekturumfang und -aufwand je nach Entwicklungskontext unterschiedlich. Sie hängen maßgeblich von den geforderten qualitativen Eigenschaften und der Erfahrung der Beteiligten ab.
Bei umfangreicher Software, die über einen längeren Zeitraum gepflegt und weiterentwickelt wird, lohnt sich der Aufbau einer **Architekturdokumentation**. Typischerweise gibt sie Aufschluss über die Struktur der Software, das Zusammenspiel der Teile, die übergreifenden Konzepte und die wesentlichen Entscheidungen. Die konkreten Bestandteile und die Detailtiefe hängen von den relevanten Zielgruppen ab. Die Architekturdokumentation enthält wesentliches, konzeptionelles Entwicklungswissen, das nicht ohne weiteres bzw. gar nicht aus dem Quelltext ersichtlich ist. Dieses Wissen ist essentiell, um die Software langfristig effizient und zielgerichtet weiterentwickeln zu können.
Die folgenden Empfehlungen sollen sicherstellen, dass wesentliche, übergreifende Entscheidungen hinsichtlich der Software beschrieben und strukturiert erarbeitet werden.

Empfehlung	ab AK	Erläuterung
ESA.1: Die Architekturdokumentation ist verständlich für die relevanten Zielgruppen aufbereitet.	2	Die Software-Architektur ist ein wichtiger Einstiegspunkt in die Software für verschiedene Beteiligte. Beispielsweise liefert sie wichtige Anhaltspunkte für die Entwickler in Bezug auf die Struktur, Schnittstellen und einzuhaltenden Architek-

		turkonzepte. Der Auftraggeber erhält einen Überblick, wie sich die Software in die übrige Systemlandschaft eingliedert und wie die Umsetzung zentraler qualitativer Eigenschaften sichergestellt wird. Deshalb ist es wichtig, für den konkreten Fall die relevanten Zielgruppen zu ermitteln und die Informationen für diese verständlich aufzubereiten.
ESA.2: Wesentliche Architekturkonzepte und damit zusammenhängende Entscheidungen sind zumindest in knapper Form dokumentiert.	1	Das sind insbesondere die Konzepte und Entscheidungen, die nicht ohne weiteres aus dem Quelltext hervorgehen (z.B. "Was sind die fachlichen Komponenten und wie wirken sie zusammen?", "Wie funktioniert das übergeordnete Parallelisierungskonzept?", "Warum wird eine bestimmte Bibliothek zur Anbindung eines Fremdsystems genutzt?"). Das Wissen darüber ist wichtig, um die Software effizient weiterentwickeln zu können. Insbesondere wenn der ursprüngliche Hauptentwickler nicht mehr verfügbar ist.
ESA.3: Die Testbarkeit der Software ist angemessen auf Ebene der Software-Architektur adressiert.	2	Damit ein Testfall ausgeführt werden kann, muss die Software (bzw. ein Teil von ihr) in einen festgelegten Zustand gebracht werden. Zudem muss es möglich sein, die relevanten Auswirkungen beobachten zu können. Diese Testbarkeitseigenschaften müssen in angemessener Form in der Software-Architektur adressiert werden. Daher ist es notwendig, die Teststrategie mit der Software-Architektur abzustimmen (vgl. Abschnitt Software-Test). Es sind beispielsweise die Realisierung von Testschnittstellen und -infrastruktur konzeptuell vorzusehen und Designprinzipien (vgl. Abschnitt Design und Implementierung) zur Förderung einer testbaren Softwarestruktur vorzugeben.
ESA.4: Die Software-Architektur ist mit den relevanten Zielgruppen abgestimmt. Änderungen werden aktiv kommuniziert und sind nachvollziehbar.	2	Primär soll dadurch ein gemeinsames Verständnis der zentralen Lösungskonzepte gefördert werden, so dass Entscheidungen durch alle Beteiligte verstanden und auch umgesetzt werden können. Zudem wird verhindert, dass wichtige Aspekte übersehen werden. Somit wird die Weiterentwicklung und Verfeinerung der Lösungskonzepte für

		alle Beteiligte sichtbar und nachvollziehbar.
ESA.5: Die Architekturdokumentation überlappt möglichst gering mit der Implementierung.	2	Die Architekturdokumentation muss mit fortschreitender Entwicklung aktualisiert werden. Daher ist zu empfehlen, implementierungsnahe Aspekte (z.B. interner Aufbau einer Komponente) nicht darin aufzunehmen. Insbesondere ist zu vermeiden, dass Informationen dupliziert werden. Diese können auf Dauer nicht synchron gehalten werden.
ESA.6: Die Architekturdokumentation verwendet durchgehend die Begrifflichkeiten der Anforderungen.	2	In erster Linie erleichtert es den einen Einstieg in die Lösungskonzepte zu finden. Zudem vermeidet man Missverständnisse und Fehler, die auf die unterschiedliche Interpretation von Begrifflichkeiten zurückzuführen sind (vgl. Abschnitt Anforderungsmanagement, Empfehlung EAM.8).
ESA.7: Architekturkonzepte und -entscheidungen lassen sich auf Anforderungen zurückführen.	2	Insbesondere die einzuhaltenden qualitativen Eigenschaften beeinflussen viele Aspekte der Software-Architektur stark. Zusätzlich helfen Randbedingungen aus den verschiedenen Lösungsansätzen, die angemessenen Optionen auszuwählen. Schließlich wird die Software-Architektur durch die anforderungsbezogene Vorgehensweise nachvollziehbar und begründbar.
ESA.8: Wesentliche Architekturkonzepte werden durch geeignete Maßnahmen bezüglich ihrer Tauglichkeit überprüft.	2	Ungünstige Entscheidungen auf Architekturebene führen häufig zu erheblichen Mehraufwänden. Deshalb sind die wesentlichen Konzepte praktisch – beispielsweise durch eine prototypische Implementierung – zu erproben.
ESA.9: Die Architekturdokumentation wird regelmäßig aktualisiert.	2	Die Architekturdokumentation muss konsistent zur bestehenden Implementierung der Software sein. Eine veraltete oder teilweise falsche Architekturdokumentation reduziert stark deren Nutzen und ist mitunter kritischer zu bewerten als eine nicht vorhandene.
ESA.10: Es erfolgt regelmäßig ein systematisches Review der Software, um Abweichungen von der Software-Architektur zu erkennen.	3	Dadurch werden ungünstige Entscheidungen auf Architektur- und Implementierungsebene sichtbar. Daraus resultierende Verbesserungspotentiale sind zu bewerten und über den Änderungsprozess priorisiert umzusetzen. Zur Unterstützung ist der Einsatz von Code-Analysewerkzeugen zu empfeh-

		len.
ESA.11: Es erfolgt regelmäßig ein systematisches Review der Software-Architektur, ob sie geeignet ist, die gestellten Anforderungen zu erfüllen.	3	Es wird sichergestellt, dass in der Software-Architektur die gestellten Anforderungen angemessen adressiert werden. Somit lassen sich ungünstige Architekturentscheidungen erkennen. Daraus resultierende Verbesserungspotentiale sind zu bewerten und über den Änderungsprozess priorisiert umzusetzen.

4.4 Änderungsmanagement

Gegenstand des Änderungsmanagements ist, systematisch und nachvollziehbar Änderungen an der Software durchzuführen. Ursachen für Änderungen sind beispielsweise Anforderungen, Fehler oder Optimierungen. Das Änderungsmanagement unterstützt dabei, den Überblick über den Entwicklungsstand zu behalten und die verschiedenen Entwicklungsaufgaben zu koordinieren.
In diesem Zusammenhang beschreibt der **Änderungsprozess**, wie **Änderungswünsche** (z.B. Anforderungen, Fehler, Optimierungen) prinzipiell auf Entwicklerseite abgearbeitet werden und anschließend ggf. in Form einer neuen Software-Version zur Verfügung stehen. Dieser Prozess ist im Detail in jedem Entwicklungskontext unterschiedlich. Daher ist es wichtig, diesen im Entwicklungsteam abzustimmen und kontinuierlich zu verbessern. In der Praxis ist darauf zu achten, dass sich die Abläufe effizient umsetzen lassen. Daher ist auf angemessenen Einsatz von Werkzeugen und Automatisierung zu achten.
Um Änderungswünsche zentral zu erfassen, werden häufig Web-basierte **Ticketsysteme** (z.B. MantisBT, Jira) – auch als "Bug Tracker" oder "Issue Tracker" bezeichnet – eingesetzt. Damit lässt sich der Überblick über alle zu erledigenden Aufgaben behalten. Ticketsysteme erlauben es, die Änderungswünsche bestimmten Software-Versionen zuzuordnen. Auf dieser Basis stellen sie Planungsübersichten (**Roadmap**) und detaillierte Änderungshistorien (**Changelog**) zur Verfügung. Schließlich erlauben es Ticketsysteme, oftmals die Abarbeitung eines Änderungswunsches an den eigenen Änderungsprozess anzupassen. Der Einsatz eines Ticketsystems lohnt sich insbesondere bei der längerfristigen Entwicklung umfangreicher Software und wenn verteilte Entwicklungsteams zusammenarbeiten sollen.
Eine weitere wichtige Aufgabe des Änderungsmanagements besteht darin, die Ergebnisse der Entwicklungsarbeit sicher und nachvollziehbar zu verwalten. Bei den Ergebnissen handelt es sich beispielsweise um den Quelltext, die Testprozeduren inklusive notwendiger Testdaten oder die Benutzerdokumentation. Diese Ergebnisse werden typischerweise in einem Projektarchiv (**Repository**) abgelegt.
Im Repository werden möglichst alle Artefakte verwaltet, die zum Erstellen einer lauffähigen Version der Software und deren Test erforderlich sind. Zur Verwaltung der Verzeichnisse und Dateien eines Repository wird ein **Versionskontrollsystem** (z.B. Git, Subversion) eingesetzt. Es stellt sicher, dass jede Änderung des Repository (**Commit**) mit einer Beschreibung protokolliert (**Commit Message**) und über eine Versionshistorie einsehbar ist. Auf dieser Basis bietet es entscheidende

Vorteile für die Entwicklung. Beispielsweise lassen sich alte bzw. bereits gelöschte Versionsstände wiederherstellen. Bei Fehlern kann man mit Hilfe der Änderungshistorie die Ursache einfacher eingrenzen. Wichtige Zwischenstände können festgehalten werden (**Tag**) und sind somit schnell auffindbar. Parallele Änderungen derselben Dateien werden erkannt und die betreffenden Entwickler bei der Behebung dieses Konflikts unterstützt. Schließlich können verschiedene Entwicklergruppen voneinander losgelöst mit Hilfe von parallelen Entwicklungszweigen (**Branch**) arbeiten. Der zusätzliche Aufwand, ein Versionskontrollsystem zu erlernen und zu nutzen, rentiert sich in der Praxis schnell.
Die folgenden Empfehlungen sollen einen strukturierten Umgang mit Änderungen der Software und deren Nachvollziehbarkeit sicherstellen.

Empfehlung	ab AK	Erläuterung
EÄM.1: Der Änderungsprozess ist im Entwicklungsteam abgestimmt und dokumentiert.	2	Der Änderungsprozess beschreibt die grundlegenden praktischen Entwicklungsabläufe und verankert die wesentlichen Testaktivitäten (vgl. Abschnitt Software-Test). Primär dient dieser Prozess der Unterstützung der Entwickler, um die Zusammenarbeit im Entwicklungsteam zu verbessern und Fehler zu vermeiden. In diesem Zusammenhang ist auf Praktikabilität zu achten. D.h., die Entwicklungsabläufe sollten gut durch Werkzeuge und die Automatisierung von Routineaufgaben unterstützt werden (vgl. Abschnitt Automatisierung und Abhängigkeitsmanagement). Es ist empfehlenswert, den Änderungsprozess in regelmäßigen Abständen einem Review zu unterziehen.
EÄM.2: Die wichtigsten Informationen, um zur Entwicklung beitragen zu können, sind an einer zentralen Stelle abgelegt.	1	Diese Informationen sind essentiell für neue Entwickler oder falls nach längerer Pause die Entwicklung wiederaufgenommen werden soll. Sie umfassen die grundlegenden Schritte, um mit der Entwicklung beginnen zu können (z.B. "Was wird benötigt, um die lauffähige Software zu erstellen?", vgl. Abschnitt Automatisierung und Abhängigkeitsmanagement, Empfehlungen EAA.1 und EAA.2). Häufig befinden sich diese Informationen direkt im Repository. Typischerweise legt man diese in der Datei "README" oder "CONTRIBUTING" ab. Alternativ bietet es sich an, eine Webseite für den Einstieg zu erstellen.

EÄM.3: Änderungswünsche werden zumindest mit einer eindeutigen Kennung, einer Kurzbeschreibung und den Kontaktdaten des Urhebers zentral erfasst. Sie sind langfristig und durchsuchbar gespeichert. Im Fall von Fehlerberichten sind zusätzlich Angaben zur Reproduzierbarkeit, zum Schweregrad und zur betroffenen Software-Version zu erfassen.	2	Dadurch sind alle Software-bezogenen Aufgaben an einer Stelle auffindbar. Zudem ist sichergestellt, dass ausreichend Informationen zu deren Bearbeitung vorliegen. Auf dieser Basis lassen sich alle Aufgaben überblicken und sinnvoll priorisieren. Die eindeutige Kennung erlaubt es, die Aufgaben aus einem anderen Kontext bzw. Werkzeug heraus zu referenzieren. Dies ist die Grundlage, um Änderungen vom Ursprung bis hin zu den Auswirkungen nachvollziehen zu können (Traceability).
EÄM.4: Es existiert eine Planungsübersicht (Roadmap), die beschreibt, welche Software-Versionen mit welchen Ergebnissen wann erreicht werden sollen.	2	Die Roadmap erhöht die Übersichtlichkeit der ggf. umfangreichen Liste von Änderungswünschen. Es wird ersichtlich, welche Aufgaben im Fokus der aktuellen Entwicklungsphase liegen. Engpässe und inhaltliche Überschneidungen lassen sich einfacher feststellen. In der Praxis unterstützt die Roadmap die Diskussion des Entwicklungsfortschritts und stellt ein effizientes Mittel zur Release-Planung dar (vgl. Abschnitt Release-Management).
EÄM.5: Bekannte Fehler, wichtige ausstehende Aufgaben und Ideen sind zumindest stichpunktartig in einer Liste festgehalten und zentral abgelegt.	1	Diese Informationen vereinfachen die spätere Weiterentwicklung bzw. sind auch für Nutzer der Software interessant. Im einfachsten Fall können sie als Teil der "README"-Datei im Repository abgelegt werden.
EÄM.6: Es existiert eine detaillierte Änderungshistorie (Changelog), aus der hervorgeht, welche Funktionen und Fehlerbeseitigungen in welcher Software-Version enthalten sind.	2	Das Changelog erhöht die Übersichtlichkeit der ggf. umfangreichen Liste von Änderungswünschen. Es lässt sich nachvollziehen, in welcher konkreten Software-Version eine Funktionalität oder Fehlerbeseitigung enthalten ist. Dies ist hilfreich, um die Ursache von Fehlern einzugrenzen. Schließlich lassen sich auf Basis des Changelog einfach Release Notes generieren (vgl. Abschnitt Release-Management).
EÄM.7: Ein Repository ist in einem Versionskontrollsystem eingerichtet. Das Repository ist angemessen strukturiert und enthält möglichst alle Artefakte, die zum Erstellen einer nutzbaren Version der Soft-	1	Das Repository ist der zentrale Einstiegspunkt in die Entwicklung. Dadurch sind alle wesentlichen Artefakte sicher gespeichert und an einer Stelle auffindbar. Einzelne Änderungen können nachvollzogen und dem jeweiligen Urheber zugeordnet werden. Darüber hinaus stellt das Versionskon-

ware und deren Test erforderlich sind.		trollsystem die Konsistenz aller Änderungen sicher. Die Verzeichnisstruktur des Repository sollte man anhand bestehender Konventionen ausrichten. Quellen dafür sind typischerweise das Versionskontrollsystem, das Build-Werkzeug (vgl. Abschnitt Automatisierung und Abhängigkeitsmanagement) oder die Community der eingesetzten Programmiersprache bzw. des verwendeten Frameworks. Dazu zwei Beispiele: 1. Der Hauptentwicklungszweig heißt im Fall des Versionskontrollsystems Subversion "trunk". Im Fall von Git heißt dieser "master". 2. Das Build-Werkzeug Maven (vgl. Abschnitt Automatisierung und Abhängigkeitsmanagement) standardisiert zum großen Teil die Verzeichnisstruktur unterhalb des Hauptentwicklungszweigs. Beispielsweise ist der Java-Quelltext im Verzeichnis "src/main/java" abzulegen. Tests befinden sich im Ordner "src/test". Insbesondere ist zu empfehlen, die Verzeichnisstruktur unterhalb des Hauptentwicklungszweigs stabil zu halten. Build-Werkzeuge zum automatischen Erstellen und Testen der Software bauen darauf auf. Um Zwischenstände der Software reproduzieren zu können, ist es erforderlich, dass alle dazu notwendigen Artefakte im Repository vorhanden sind. Dazu gehören neben dem Quelltext i.d.R. auch Testskripte und -daten (vgl. Abschnitt Software-Test) sowie Abhängigkeiten, Konfigurationseinstellungen und Skripte zum Erstellen der Software (vgl. Abschnitt Automatisierung und Abhängigkeitsmanagement). In der Praxis gibt es jedoch Grenzen. Beispielsweise wenn die Artefakte sehr groß bzw. stark betriebssystemabhängig sind. In diesen Fällen sind zumindest auseichend Informationen zu hinterlegen, um bei Bedarf auf diese Artefakte zugreifen zu können.
EÄM.8: Jede Änderung des Repository dient möglichst einem spezifi-	1	Versionskontrollsysteme helfen Versionsstände wiederherzustellen, Fehler einzugrenzen und Än-

schen Zweck, enthält eine verständliche Beschreibung und hinterlässt die Software möglichst in einem konsistenten, funktionierenden Zustand.		derungen nachvollziehen zu können. Damit man diese Funktionen effektiv nutzen kann, ist folgende Arbeitsweise zu empfehlen: Eine Änderung des Repository dient möglichst genau einem Zweck. Beispielsweise behebt sie einen Fehler oder fügt eine neue Funktionalität hinzu. Wenn ein Commit viele verschiedene Änderungen mischt, lässt sich beispielsweise eine darin enthaltene Fehlerbehebung schwerer in andere Entwicklungszweige integrieren. Zudem lassen sich Fehler, die durch solch einen Commit verursacht werden, schwerer auf ihre Ursache zurückführen. Die Commit Message beschreibt den Zweck der Änderung und fasst die wichtigsten Details knapp zusammen. Informationen, die direkt aus dem Versionskontrollsystem hervorgehen, sollten nicht in der Commit Message dupliziert werden. Beispielsweise zeigt ein Versionskontrollsystem alle inhaltlichen Änderungen eines Commit übersichtlich an. Es ist zu empfehlen, die Commit Message mit einem aussagekräftigen, kurzen Satz zu beginnen. Durch eine Leerzeile abgesetzt können ggf. weitere Details folgen. Dadurch erhöht sich die Übersichtlichkeit bei der Arbeit mit der Versionshistorie. Nach dem Commit liegt die Software in einem funktionierenden Zustand vor. Dies vermeidet, dass die übrigen Entwickler in ihrer Arbeit behindert werden. Weiterhin fällt es leichter, eine Änderung in andere Entwicklungszweige zu integrieren.
EÄM.9: Falls mehrere gemeinsame Entwicklungszweige existieren, lässt sich deren Zweck einfach erschließen.	2	Dies erhöht die Übersichtlichkeit im Repository. Typischerweise findet sich auf dem Hauptentwicklungszweig (z.B. "trunk" oder "master" genannt) die neuste Version der Software. Daneben können weitere, aktive Entwicklungszweige existieren, beispielsweise um eine bestimmte Funktionalität umzusetzen oder einen Stand der Software vor der Veröffentlichung zu stabilisieren. In diesem Zusammenhang ist zu empfehlen, dass nur die

		Entwicklungszweige sichtbar sind, an denen auch tatsächlich aktiv gearbeitet wird. Weiterhin sollte man das Format und die Bedeutung der verwendeten Entwicklungszweignamen festlegen. Beispielsweise dient der Branch mit dem Namen "RB-1.0.0" zur Stabilisierung der Produktivversion 1.0.0 (vgl. Abschnitt Release-Management).
EÄM.10: Zu jedem Änderungswunsch lassen sich die Änderungen im Repository nachvollziehen (Traceability).	3	Somit lässt sich beispielsweise einfach prüfen, ob alle notwendigen Änderungen auch tatsächlich durchgeführt worden sind. Zudem sind die Auswirkungen eines Änderungswunsches direkt sichtbar. In Kombination mit dem Changelog kann die Suche nach der Ursache von Fehlern vereinfacht werden. Praktisch lässt sich dies durch die Integration des Ticketsystems mit dem Versionskontrollsystem realisieren. Typischerweise enthält jede Commit Message einen Verweis auf einen Änderungswunsch. Durch diese Maßnahme kann man im Ticketsystem direkt die Auswirkungen von Änderungswünschen auf das Repository betrachten. Auch auf Basis der Versionshistorie des Repository lässt sich auf den Grund einer Änderung schließen.

4.5 Design und Implementierung

Zu Beginn der Entwicklung entsteht häufig eine erste Idee der groben Softwarestruktur. Diese beschreibt eine geeignete Zerlegung der Software nach fachlichen und technischen Gesichtspunkten. Im Folgenden nehmen wir eine beispielhafte Zerlegung der Software in fachliche **Komponenten** an. Auf dieser Basis sind Schritt für Schritt die Bestandteile der Komponenten – hier die **Module** – zu entwerfen und zu implementieren. Dabei ist häufig gewünscht, dass die Softwarestruktur verständlich, leicht änderbar und erweiterbar ist. Zudem sind die konzeptionellen Randbedingungen der gewählten Software-Architektur (vgl. Abschnitt Software-Architektur) einzuhalten. Beispielsweise ist eine Suche von Schlüsselwörtern in einem Text umzusetzen. Die Auswahl des Suchalgorithmus und weitere Designentscheidungen hängen dabei stark von der geforderten Antwortzeit und der Größe des zu durchsuchenden Texts ab. Daher ist es notwendig, diese Randparameter zu kennen, um ein angemessenes Suchmodul zu implementieren.
Design und Implementierung sind eng miteinander verknüpft und laufen in kleinen, sich wiederholenden Schritten ab. Kontinuierliches Refactoring und das konsequente Testen auf Modulebene unterstützen diese iterative Vorgehensweise. **Refactoring** bezeichnet eine Verbesserung der

Struktur der Software unter Beibehaltung ihres sichtbaren Verhaltens. Es ist essentiell, um langfristig die Qualität der Softwarestruktur zu erhalten. **Modultests** (vgl. Abschnitt Software-Test) bilden die Basis für effizientes Refactoring. Sie erlauben es, schnell Änderungen der Software zu überprüfen.
Konkrete Anhaltspunkte für eine angemessene Umsetzung liefern Designprinzipien und Entwurfsmuster. **Designprinzipien** sind Heuristiken bzgl. Design und Implementierung. Bei deren konsequenter Anwendung hat sich gezeigt, dass diese sich positiv auf die Qualität der Softwarestruktur auswirken. Beispielsweise empfiehlt das Don`t-Repeat-Yourself-Prinzip, Dopplungen von Informationen möglichst zu vermeiden. Solche treten z.B. häufig im Quelltext und der Dokumentation auf und begünstigen Inkonsistenzen und Fehler. **Entwurfsmuster** beschreiben bewährte Lösungen für typische Designprobleme. Beispielsweise erläutert das Model-View-Controller-Muster, wie sich eine interaktive Schnittstelle möglichst wiederverwendbar als Softwarestruktur realisieren lässt. Schließlich helfen gemeinsame Regeln bzgl. des **Programmierstils** eine konsistente Umsetzung in Form von Quelltext zu erzielen.
Die folgenden Empfehlungen sollen den Einsatz üblicher Designprinzipien und Implementierungstechniken sicherstellen.

Empfehlung	ab AK	Erläuterung
EDI.1: Es werden die üblichen Konstrukte und Lösungsansätze der gewählten Programmiersprache eingesetzt sowie ein Regelsatz hinsichtlich des Programmierstils konsequent angewendet. Der Regelsatz bezieht sich zumindest auf die Formatierung und Kommentierung.	1	In Programmiersprachen existieren häufig zu bevorzugende Ansätze, um konkrete Probleme zu lösen. Diese gilt es anzuwenden, um eine effiziente, verständliche Implementierung zu erstellen und Fehler zu vermeiden. Zudem unterstützen Regeln zum Programmierstil, einen möglichst konsistenten Quelltext zu erhalten. Dadurch erhöht sich dessen Verständlichkeit, wodurch die Software leichter gewartet werden kann. Im Folgenden werden einige Hinweise zum Programmierstil gegeben. Es ist zu empfehlen, sich an einem bestehenden Regelwerk zu orientieren: Der Quelltext sollte über ein aufgeräumtes und konsistentes Layout verfügen. Dies erleichtert dessen Verständnis und die Arbeit damit. Insbesondere lässt sich der Quelltext schneller überblicken und relevante Informationen leichter finden. Dazu sollten beispielsweise Kommentarblöcke immer an der erwarteten Stelle zu finden sein (z.B. vor einer Funktionsdefinition). Weiterhin sollte man darauf achten, funktional ähnlichen Quelltext auch ähnlich zu formatieren. Wichtige Informationen sollten bereits über den

		Namen der Quelltextelemente (z.B. Variablen, Funktionen) deutlich werden. Dazu sind spezifische Wörter zu verwenden. Beispielsweise suggeriert "DownloadPage", dass eine Netzwerkoperation zum Zugriff auf die konkrete Webseite erforderlich ist. Im Fall von "GetPage" fehlt diese Zusatzinformation. Dementsprechend sollten Füllwörter und generische Namen wie z.B. "i, j ,k, tmp" eher vermieden werden. Im Fall von Variablen mit kurzer Gültigkeitsdauer ist der Einsatz dieser generischen Namen jedoch sinnvoll. Die wesentlichen Komponenten und Module sind angemessenen zu kommentieren. Mit Hilfe der Kommentare soll der Leser möglichst wichtige Zusatzinformationen (z.B. Invarianten, Einschränkungen, Fallstricke) zu einem Modul bzw. einer Komponente erhalten. Daher sollte man vermeiden offensichtliche Aspekte zu kommentieren. Insbesondere im Fall von Komponenten ist es hilfreich, deren Zweck inklusive der Einordnung in die übrige Softwarestruktur zu beschreiben. Als Zielgruppe sind i.d.R. Entwickler anzunehmen, die die genutzte Programmiersprache beherrschen und die fachliche Domäne kennen. Generell ist darauf zu achten, dass sich die Kommentierung mit den übrigen Dokumentationsquellen ergänzt und Informationen nicht dupliziert.
EDI.2: Die Software ist möglichst modular strukturiert. Die Module sind lose gekoppelt, d.h., ein einzelnes Modul hängt möglichst gering von anderen Modulen ab.	1	Dazu ist es erforderlich, dass jedes Modul möglichst einem konkreten Zweck dient. Dadurch verringert sich dessen Komplexität auf der Ebene der Implementierung und der Schnittstelle. In der Konsequenz sind Module verständlicher, einfacher zu testen sowie leichter wiederverwendbar. Schließlich trägt dieser Ansatz dazu bei, Änderungen eines Moduls möglichst lokal zu begrenzen.
EDI.3: Zu jedem Modul gibt es möglichst durchgängig Modultests. Die Modultests zeigen deren typische Verwendung und Einschränkungen auf.	2	Modultests unterstützen die Entwicklung und sind ein wichtiges Mittel, um langfristig eine effiziente Weiterentwicklung der Software sicherzustellen. Wesentliche Vorteile von Modultests umfassen: Modultests liefern konkrete Quelltextbeispiele, die

		zeigen wie ein Modul zu nutzen ist und Fehlersituationen zu behandeln sind. Somit stellen sie einen wichtigen Teil der technischen Dokumentation dar. Modultests geben Hinweise zur Qualität des Designs. Komplizierte, umfangreiche Modultests weisen darauf hin, dass das Modul eventuell zu komplex oder zu stark mit den übrigen Modulen gekoppelt ist. Dadurch lässt sich frühzeitig verhindern, dass ungünstige Designentscheidungen sich auf weitere Bereiche der Software auswirken. Die einfache Automatisierbarkeit und die kurze Ausführungsdauer der Modultests erlauben es, Regressionen (vgl. Abschnitt Software-Test) bereits während der Entwicklung zu erkennen und zu beheben. Damit liefern Modultests eine wichtige Voraussetzung, dass Verbesserungen der Softwarestruktur effizient durchführbar sind (Refactoring). Idealerweise erstellt der Entwickler diese Funktionstests parallel zum eigentlichen Modul (vgl. Empfehlung EST.2). Je nach Softwaretyp ist es jedoch nicht immer möglich bzw. sinnvoll, alle Module durch Modultests abzudecken. Insbesondere im Fall von grafischen Oberflächen ist die Testbarkeit auf dieser Ebene ggf. "schwierig" herzustellen.
EDI.4: Die Implementierung spiegelt die Software-Architektur wieder.	2	D.h., es lassen sich die in der Softwarestruktur festgelegten Komponenten auf Quelltextebene wieder finden. Beispielsweise existiert zu einer technischen Komponente "persistence" ein konkretes Java-Paket mit gleichem Namen. Durch dieses Prinzip erhöht sich die Übersichtlichkeit, da man anhand der Strukturdiagramme und unter bekannten Begriffen den Einstiegspunkt auf Quelltextebene findet. Zudem lassen sich beiläufige Änderungen der Softwarestruktur auf Architekturebene vermeiden und ein ggf. bestehender Verbesserungsbedarf auf Architekturebene einfacher erkennen.

		Generell ist an dieser Stelle zu empfehlen, die Begriffe auf Ebene der Anforderungen (vgl. Abschnitt Anforderungsmanagement) und Software-Architektur (vgl. Abschnitt Software-Architektur) konsequent in der Implementierung zu nutzen, um Missverständnissen und Fehlern vorzubeugen.
EDI.5: Während der Entwicklungsaktivitäten wird kontinuierlich auf Verbesserungspotential geachtet. Erforderliche Anpassungen (Refactoring) werden ggf. direkt oder über den Änderungsprozess priorisiert umgesetzt.	2	Falls Verbesserungen der Softwarestruktur nicht kontinuierlich durchgeführt werden, verschlechtert sich die Qualität der Softwarestruktur. In der Konsequenz ist die Software immer schlechter anpassbar und erweiterbar. Es fällt beispielsweise auf, dass eine Funktion aufzuteilen ist, bevor eine Erweiterung sinnvoll umgesetzt werden kann. Es ist zu empfehlen, solch kleinere Anpassungen konsequent direkt durchzuführen. Ein gutes Sicherheitsnetz bestehend aus Tests (insbesondere Modultests) hilft diese Änderung schnell und sicher umzusetzen. Im Fall von größeren Anpassungen ist zu empfehlen, die Auswirkungen genauer zu analysieren und priorisiert über den Änderungsprozess (vgl. Abschnitt Änderungsmanagement) abzuarbeiten.
EDI.6: Die Eignung der Regeln bzgl. des Programmierstils wird regelmäßig geprüft. Ggf. werden zu bevorzugende Lösungsansätze und Entwurfsmuster, zu beachtende Designprinzipien und -regeln sowie Regeln bzgl. erlaubter und nicht erlaubter Sprachelemente ergänzt.	2	Mit voranschreitender Entwicklung stellen sich bevorzugte Lösungsansätze heraus und Erfahrungswerte zu bestehenden Regeln ein. Deshalb ist zu empfehlen, den Programmierstil um diese Erkenntnisse zu ergänzen und somit Fehlern vorzubeugen. Beispielsweise hat sich gezeigt, dass Mehrfachvererbung sich nur im Fall von Schnittstellenklassen bewährt hat. Daher soll Mehrfachvererbung zukünftig nur noch zu diesem Zweck eingesetzt werden.
EDI.7: Die Einhaltung einfacher Regeln des Programmierstils wird automatisiert geprüft bzw. sichergestellt.	2	Neben der Verabredung bestimmter Regeln, ist auch darauf zu achten, dass diese umgesetzt werden. Im Fall von einfachen Regeln (z.B. Namensmuster für Variablennamen) existieren häufig Werkzeuge, die Inkonsistenzen zum festgelegten Programmierstil feststellen (Style Checker) oder direkt beheben (Source Formatter) können. Je

		nach Werkzeugtyp ist es sinnvoll, den Einsatz beispielsweise über eine gemeinsame Entwicklungsumgebung oder über das Build-Skript (vgl. Abschnitt Automatisierung und Abhängigkeitsmanagement) sicherzustellen.
EDI.8: Wesentliche Designprinzipien sind festgelegt und kommuniziert.	2	Designprinzipien fördern bestimmte Arbeitsstile bei der Entwicklung, die die Qualität der Softwarestruktur verbessern. Beispielsweise fördert die Pfadfinderregel, dass kleinere "Unschönheiten" im Quelltext (auch Code Smell genannt) direkt bei der Arbeit an einem Modul behoben werden. Durch konsequente Anwendung dieses Prinzips beugt man aktiv Fehlern vor. Es ist zu empfehlen, die wesentlichen Designprinzipien bewusst auszuwählen und im Entwicklungsteam zu kommunizieren. Sie lassen sich über den definierten Programmierstil sinnvoll verankern.
EDI.9: Im Quelltext und in den Kommentaren sind möglichst wenig duplizierte Informationen enthalten. ("Don`t repeat yourself.")	1	Die betroffenen Stellen können auf Dauer nicht synchron gehalten werden. Daher sind im Laufe der Entwicklung Inkonsistenzen und Fehler zu erwarten.
EDI.10: Es werden einfache, verständliche Lösungen bevorzugt eingesetzt. ("Keep it simple and stupid.").	1	Dieses Prinzip hat ein einfaches, verständliches Design zum Ziel. Denn unnötig komplexe Lösungen erhöhen ohne Nutzen den Aufwand, die Software zu verstehen und zu erweitern. D.h. nicht, dass generische, komplexe Lösungen per se verboten sind. Man sollte sich dafür bewusst entscheiden bzw. die Lösung dahingehend schrittweise "wachsen" lassen. Dies gilt auch für den Einsatz von Entwurfsmustern. Anstatt beispielsweise direkt mit dem Abstract-Factory-Muster zu beginnen, kann es zunächst sinnvoll sein, auf dieses Muster zu verzichten bzw. das leichtgewichtige Factory-Method-Muster einzusetzen. Später kann unter Umständen diese Lösung hin zum Abstract-Factory-Muster weiterentwickelt werden (Refactoring). Wichtig ist jedoch bewusst und begründet anhand der Anforderungen (vgl. Abschnitt Anforderungsmanagement) zu entscheiden.

EDI.11: Die Angemessenheit der Lösung, die Einhaltung der vereinbarten Regeln bzgl. des Programmierstils und einzuhaltende Randbedingungen bzgl. der Software-Architektur werden systematisch durch Code-Reviews überprüft.	3	Einige Aspekte lassen sich nicht oder nur aufwendig automatisiert prüfen. Dazu zählen die Verständlichkeit und Angemessenheit der implementierten Lösung. Dafür stellen Code-Reviews eine effiziente Alternative dar. Verfügbare Werkzeuge lassen sich mittlerweile sehr effizient in den Entwicklungsablauf integrieren. Typischerweise prüfen ein bis zwei erfahrene Entwickler eine Änderung bevor diese in den gemeinsamen Entwicklungszweig eingeht. Dadurch lassen sich eine Vielzahl von Fehlern frühzeitig erkennen und beheben. Zudem unterstützen Code-Reviews den Lernprozess im Entwicklungsteam. Es ist zu empfehlen, Code-Reviews über den Änderungsprozess zu verankern (vgl. Abschnitt Änderungsmanagement).

4.6 Software-Test

Beim **Testen** führt man die Software aus und analysiert diese, um Fehler zu finden. Ein **Fehler** ist eine Abweichung des tatsächlichen vom erforderlichen Zustand. Beispielsweise berechnet die Software nicht das Produkt einer Zahlenreihe sondern deren Summe (Abweichung gegenüber funktionaler Anforderung). In einem anderen Fall liefert die Software das Ergebnis nach einer Sekunde und nicht wie gefordert nach einer Millisekunde (Abweichung gegenüber Qualitätsanforderung). Mit Hilfe von Tests kann kein Korrektheitsnachweis erbracht werden. Vielmehr schafft das Testen Vertrauen in die Software, indem es aufzeigt, wie gut sie die gewünschten Eigenschaften erfüllt.
Die konkreten Testaktivitäten hängen stark von der jeweiligen Software ab. Da ein vollständiger Korrektheitsnachweis praktisch nicht geführt werden kann, sind alle kritischen Fehler möglichst auszuschließen. Daher ist festzulegen, welche Aspekte der Software mit welchen Methoden und Techniken zu testen sind (**Teststrategie**). Die Umsetzung der Teststrategie erfordert nicht zu vernachlässigenden Aufwand. Beispielsweise können separate Testinfrastruktur oder spezielle Schnittstellen benötigt werden. Diese Anforderungen sind frühzeitig zu identifizieren (vgl. Abschnitt Anforderungsmanagement) und benötigen ggf. Entscheidungen und Konzepte auf Architekturebene (vgl. Abschnitt Software-Architektur). Damit sollen Situationen wie folgt vermieden werden: "Die zum Test erforderlichen Schnittstellen sind nicht vorhanden. Es ist zu spät oder zu aufwendig, diese zu implementieren. In der Konsequenz können die Tests nicht durchgeführt werden. Das Risiko für Fehler im Betrieb steigt."
Es gibt verschiedene Möglichkeiten, einen konkreten Test bzw. Testfall zu klassifizieren. Nach der **Teststufe** lassen sich die folgenden Testarten unterscheiden:

- **Modultests** (auch als Unit Tests bzw. Komponententests bezeichnet) zeigen, wie ein konkretes Modul funktioniert, welche Einschränkungen bestehen und welche Randbedingungen zu beachten sind.
- **Integrationstests** konzentrieren sich auf das Zusammenspiel bestimmter Module und Komponenten. Sie erlauben es, Fehler auf der Ebene der Schnittstelle zu finden.
- **Systemtests** sollen sicherstellen, dass die Software als Ganzes die spezifizierten Anforderungen erfüllt. Diese Tests werden typischerweise gegen die in einer Testumgebung installierte Software ausgeführt. Vielfach ist die Einhaltung von Qualitätsanforderungen nur auf dieser Teststufe überprüfbar.
- **Abnahmetests** prüfen, ob die Software aus Sicht des Auftraggebers die gestellten Anforderungen erfüllt. Diese Tests werden unter Beteiligung des Auftraggebers und auf Basis der in der Zielumgebung installierten Software durchgeführt. Das Bestehen dieser Teststufe ist häufig die Voraussetzung für eine Übernahme der Software durch den Auftraggeber.

In der Praxis ist es wichtig, auf eine angemessene **Testautomatisierung** zu achten und die verschiedenen Teststufen effektiv miteinander zu kombinieren. Das Konzept der Testpyramide liefert dafür einen praktikablen Ansatz. Die Grundidee ist, den Fokus auf Modultests zu legen. Diese Tests haben den Vorteil, dass sie sich gut automatisieren lassen, schnell verlässliche Ergebnisse liefern, keine komplizierte Testumgebung benötigen und überschaubaren Wartungsaufwand erfordern. Damit unterstützen Modultests direkt die Entwicklung und finden bereits eine Vielzahl von Fehlern. Komplementär dazu folgen Tests auf Integrations- und Systemtestebene. Diese Tests sind unerlässlich, da nur diese Fehler im Zusammenspiel von Modulen und Komponenten erkennen lassen.

Ein weiterer Aspekt des Testens besteht darin, einen Einblick in die Qualität der Software zu erhalten. Diese lässt sich mit Hilfe von Metriken quantifizieren. Eine **Metrik** bildet eine Eigenschaft der Software auf eine Zahl ab. In der Praxis ist es damit möglich, Trends zu erkennen und Fehlern entgegenzuwirken. Dazu ist es wichtig, Metriken gezielt auszuwählen und systematisch auszuwerten. Beispielsweise zeigt die **Testabdeckung** an, zu welchem Grad der Quelltext durch Tests überprüft wird. Dadurch lässt sich die Effektivität der Testfälle beurteilen.

Die folgenden Empfehlungen sollen den Einsatz angemessener Methoden zur frühzeitigen Erkennung und Vermeidung von Fehlern sicherstellen.

Empfehlung	ab AK	Erläuterung
EST.1: Eine übergeordnete Teststrategie ist abgestimmt und festgelegt. Sie wird regelmäßig auf ihre Angemessenheit hin geprüft.	2	Die Teststrategie legt fest, wie der Testprozess für eine konkrete Software prinzipiell ausgestaltet wird. Dabei ist u.a. zu überlegen, welche Teststufen relevant sind, in welcher Intensität und zu welchem Zeitpunkt bestimmte Tests durchzuführen sind sowie welche Testumgebung und -infrastruktur dazu benötigt wird. Dabei ist es wichtig, sich auf die für den Betrieb kritischen Aspekte zu konzentrieren.

		Eine wichtige Basis für die Teststrategie bilden Qualitätsanforderungen und Randbedingungen. Soll die Software beispielsweise unter den Betriebssystemen Linux und Windows genutzt werden, müssen die Tests unter beiden Betriebssystemen durchgeführt werden. Daher ist zu empfehlen, die Teststrategie Stück für Stück zu entwickeln und die erforderliche Testinfrastruktur entwicklungsbegleitend aufzubauen. Ggf. sind zusätzliche Schnittstellen für den Test konzeptionell in der Software-Architektur (vgl. Abschnitt Software-Architektur) zu berücksichtigen. Die damit verbundenen Aufgaben sind priorisiert über den Änderungsprozess (vgl. Abschnitt Änderungsmanagement) durchzuführen. Schließlich ist es empfehlenswert, die resultierenden Testaktivitäten im Änderungsprozess (vgl. Abschnitt Änderungsmanagement) selbst zu verankern. Dadurch lässt sich sicherstellen, dass diese systematisch ausgeführt werden und für alle an der Entwicklung Beteiligte sichtbar sind. Manuelle Schritte sind möglichst zu minimieren, um die Praktikabilität zu gewährleisten (vgl. Abschnitt Automatisierung und Abhängigkeitsmanagement).
EST.2: Funktionstests werden systematisch erstellt und ausgeführt.	2	Funktionstests helfen, Fehler auf Ebene der funktionalen Anforderungen frühzeitig zu erkennen. Dazu ist das Zusammenspiel von Testfällen auf verschiedenen Teststufen erforderlich. Beispielsweise prüft ein Systemtest, ob ein Nutzer eine Datei importieren kann. In diesem Zusammenhang stellen Integrations- und Modultests sicher, dass Fehler beim Import erkannt und richtig behandelt werden. Die Fehlerbehandlung ist auf Systemtestebene häufig gar nicht oder nur schwer zu überprüfen. Funktionstests werden sehr häufig ausgeführt, um bei voranschreitender Entwicklung Fehler (auch Regression genannt) zu erkennen. Dies ist ohne angemessene Automatisierung effizient nicht

		möglich. Dazu existieren häufig bereits passende Testwerkzeuge. Beispielsweise stehen für viele Programmiersprachen xUnit-Frameworks zur Verfügung, die es erlauben, effizient Modul- und Integrationstest zu erstellen. Auf der Ebene von Systemtests ist es ggf. erforderlich eigene Test-Infrastruktur zu erstellen.
EST.3: Die Einhaltung der qualitativen Eigenschaften wird systematisch überprüft.	3	Neben der Funktionalität spielen häufig auch Aspekte wie Effizienz oder Benutzbarkeit eine wichtige Rolle. Diese qualitativen Eigenschaften sind individuell in jeder Software ausgeprägt (vgl. Abschnitt Anforderungsmanagement) und tragen entscheidend zur ihrer Akzeptanz bei. Daher sind die wesentlichen qualitativen Eigenschaften konsequent zu überprüfen. Konkrete Testfälle lassen sich anhand von Qualitätsszenarien bestimmen (vgl. Abschnitt Anforderungsmanagement, Empfehlung EAM.4). Die Automatisierbarkeit der Testfälle hängt stark von dem jeweiligen Qualitätsmerkmal ab. Beispielsweise lassen sich Effizienz und Zuverlässigkeit relativ gut automatisiert testen. Benutzerbarkeit und Änderbarkeit hingegen eher schlecht. In diesen Fällen muss auf manuelle Methoden wie z.B. Reviews zurückgegriffen werden.
EST.4: Die grundlegenden Funktionen und Eigenschaften der Software werden in einer möglichst betriebsnahen Umgebung getestet.	1	Damit wird sichergestellt, dass sich die Software wie gefordert in der Betriebsumgebung verhält. Im Fall von Software mit geringem Umfang ist es prinzipiell ausreichend, die Hauptfunktionen manuell zu testen. Oft ist es jedoch sinnvoll zumindest bestimmte Teilaspekte des Tests zu automatisieren. Das hängt davon ab, wie hoch der Aufwand für die Automatisierung ist und wie häufig die Hauptfunktionen überprüft werden. I.d.R. sind diese Tests vor der Freigabe eines Releases durchzuführen (vgl. Abschnitt Release-Management). Bei wissenschaftlicher Software ist darauf zu achten, die Ergebnisse auch inhaltlich zu validieren. Dies kann durch Vergleich mit bekannten Lösungen oder der Diskussion mit Kollegen erfolgen.

		Häufig sind Fehler z.B. in einem Simulationsergebnis nicht offensichtlich.
EST.5: Für jeden nicht-trivialen Fehler existiert ein Test.	3	Dadurch wird sichergestellt, dass behobene Fehler nicht nochmals auftreten (Regression). Es ist zu empfehlen, den Fehler mit Hilfe eines Testfalls auf einer möglichst niedrigen Teststufe (Modul- bzw. Integrationstest) zu provozieren und ihn anschließend zu beheben.
EST.6: Es existieren möglichst keine nicht-deterministischen Funktionstests.	3	Nicht-deterministische Funktionstests treten häufig auf der Teststufe "Systemtest" auf. Dabei läuft ein Testfall nicht vorhersagbar durch oder schlägt fehl, obwohl Testobjekt, Testprozeduren und Testdaten zwischen den Testläufen nicht verändert wurden. Nicht-deterministische Tests deuten ggf. auf einen Fehler hin. Daher ist die Ursache des Problems zu ermitteln und zu beheben. Beispielsweise nutzen mehrere Testfälle einen produktiven Mailserver. Aus verschiedenen Gründen ist der Mailserver zeitweise nicht erreichbar, woraufhin diese Testfälle häufig fehlschlagen. In diesem Fall ist es sinnvoll, die Nutzung des Produktivservers im Test zumindest teilweise durch ein Testsystem zu ersetzen. In einem anderen Fall wird eine Berechnung im Zusammenspiel von verschiedenen Threads durchgeführt. Hier deuten nicht vorhersagbare Testergebnisse eher auf einen Fehler im Zusammenspiel der Threads hin.
EST.7: Geeignete Metriken werden zielgerichtet festgelegt und erfasst. Der Trend der gewählten Metriken wird regelmäßig analysiert und Verbesserungspotential identifiziert.	3	Für die Software-Entwicklung existiert eine Vielzahl von Metriken. Um damit Einblicke in die Software-Qualität zu erhalten, muss man diese regelmäßig evaluieren. Daher sind Metriken bewusst und zielgerichtet auszuwählen. Der Goal-Question-Metric-Ansatz stellt dazu eine praktikable Methode bereit. Insbesondere ist zu beachten, dass Metriken nur aussagekräftig für eine bestimmte Software sind. Zudem ist nicht die konkrete Maßzahl relevant, sondern der Trend ihrer Entwicklung. Nur so lassen sich Auswirkungen von Maßnahmen (z.B. Verstärkung der Testaktivitäten) einschätzen.

EST.8: Der Trend der Testergebnisse, der Testabdeckung, der Verletzungen des Programmierstils und der durch Codeanalysewerkzeuge ermittelten Fehler wird regelmäßig auf Verbesserungspotential hin untersucht.	2	Anhand des **Trends der Testergebnisse**, lässt sich beispielsweise erkennen, dass bestimmte Testfälle regelmäßig oder zeitweise fehlschlagen. Das kann ein Zeichen für einen verdeckten Fehler oder eine unzuverlässige Testumgebung sein (vgl. Empfehlung EST.6). Durch die Analyse der Testergebnisse wird das Problem ersichtlich und kann behoben werden. Die **Testabdeckung** zeigt an, welche Bereiche des Quelltexts durch Testfälle überprüft werden. Praktisch relevant sind insbesondere die Anweisungs- und Zweigüberdeckung. Auf dieser Basis lassen sich Aussagen über die Qualität der vorhandenen Testfälle treffen und Verbesserungsbedarf ableiten. Eine zunehmende Anzahl der **Programmierstilverletzungen** weist drauf hin, dass sich die Konsistenz des Quelltexts verschlechtert. Dadurch verringert sich dessen Verständlichkeit und das Auftreten von Fehlern wird begünstigt. Auf Basis der Trendanalyse lässt sich eine solche Tendenz erkennen und man kann bewusst gegensteuern. **Codeanalysewerkzeuge** analysieren den Quelltext hinsichtlich typischer Fehlermuster (z.B. Vergleich einer Zeichenkette in Java mit `"=="` anstatt mit `"equals"`). Dadurch lassen sich (potentielle) Programmierfehler finden. I.d.R. kategorisieren die Werkzeuge die Ergebnisse nach ihrem Schweregrad (z.B. Hinweis, Warnung, Fehler). Je nach Werkzeug und Programmiersprache treten auch falsch positive Ergebnisse auf. Daher sollte darauf geachtet werden, die Ergebnisliste auf die praktisch relevanten Fälle zu reduzieren und das Werkzeug dementsprechend zu konfigurieren. Damit diese Metriken regelmäßig evaluiert werden können, sind passende Werkzeuge zu ihrer Ermittlung auszuwählen und deren Ausführung mit Hilfe des Build-Werkzeuges zu automatisieren (vgl. Abschnitt Automatisierung und Abhängigkeitsmanagement). Schließlich ist zu empfehlen, den Zeit-

		punkt der Evaluierung der Metriken über den Änderungsprozess festzulegen (vgl. Abschnitt Änderungsmanagement).
EST.9: Der Trend neuer Fehler wird regelmäßig untersucht.	3	Dazu sind Fehler systematisch zu erfassen (vgl. Abschnitt Änderungsmanagement). Relevant für die Trendanalyse sind Fehler, die sich auf einen stabilen Versionsstand beziehen. Damit lässt sich erkennen, wie viele Fehler trotz aller Testaktivitäten in eine stabile Software-Version gelangt sind. Durch eine Zuordnung der Fehler auf bestimmte Komponenten der Software, lassen sich besonders fehleranfällige Bereiche erkennen. Auf dieser Basis lassen sich Testaktivitäten besser steuern und deren Auswirkungen feststellen.
EST.10: Das Repository enthält möglichst alle für den Test der Software erforderlichen Artefakte.	1	Dazu zählen beispielsweise Testprozeduren, Testdaten und Parameter von Testumgebungen. Auch diese Artefakte sind Teil der Software und sind nachvollziehbar aufzubewahren. In der Praxis gibt es jedoch Grenzen. Beispielsweise können Testdatensätze zu groß sein, um diese effizient im Repository zu verwalten. In diesen Fällen sollte zumindest ein Verweis auf den Testdatensatz im Repository abgelegt sein. Zusätzlich ist sicherzustellen, dass der Testdatensatz auf einem Speichermedium sicher gespeichert ist.

4.7 Release-Management

Bei einem **Release** handelt es sich um eine stabile Version der Software, die an die Nutzer (z.B. externe Projektpartner, Kollegen) weitergegeben wird. Die **Release-Nummer** stellt sicher, dass das Release und der damit verbundene Inhalt eindeutig gekennzeichnet sind. Das **Release-Paket** enthält neben dem ausführbaren Programm weitere Dateien und Informationen. Dazu gehören typischerweise Installations- und Nutzungshinweise, Kontaktinformationen, eine Übersicht zu den Neuerungen (**Release Notes**) und die Lizenzbestimmungen.
Über die **Release-Planung** legt man den Veröffentlichungszeitpunkt und den Umfang von Releases fest. Zur Unterstützung bietet es sich an, ein Ticketsystem (vgl. Abschnitt Änderungsmanagement) einzusetzen. Es verknüpft die Release-Planung mit dem Änderungsmanagement. Dadurch kann der Änderungsumfang eines Releases bis auf die Ebene des Repository leicht nachvollzogen werden.
Bis das Release veröffentlicht werden kann, sind je nach Software und Entwicklungskontext verschiedene Schritte durchzuführen. Es ist zu empfehlen, diesen Prozess (**Release-Durchführung**)

inklusive der Freigabekriterien festzulegen und essentielle Aspekte zu automatisieren. Damit wird sichergestellt, dass das Release die angestrebte Qualität besitzt. Dazu ein Beispiel: Zu Beginn der Release-Durchführung ist ein separater Entwicklungszweig anzulegen, um den Softwarestand zu stabilisieren. Daher dürfen dort nur Änderungen durchgeführt werden, die Fehler beseitigen oder die Dokumentation betreffen. Sobald die erforderliche Release-Dokumentation vorhanden ist und der Softwarestand alle vorgesehenen Tests besteht, sind die Freigabekriterien für das Release erfüllt. Anschließend wird das Release-Paket erstellt und den Projektpartnern über die Projektseite zur Verfügung gestellt. Zum Abschluss ist der dem Release zugrunde liegende Softwarestand im Repository festzuhalten und im Ticketsystem das Release als "abgeschlossen" zu kennzeichnen.
Abschließend ein wichtiger Hinweis bzgl. der Weitergabe des Release-Pakets. Bevor das Release-Paket an Dritte außerhalb des DLR (z.B. externe Partner oder Organisationen) weitergeben wird, sind folgende Aspekte unbedingt zu beachten:

1. Es müssen die Lizenzbedingungen, unter denen die Software weitergegeben wird, festgelegt sein und dem Release-Paket beigelegt werden. In diesem Zusammenhang ist insbesondere darauf zu achten, dass die Pflichten und Beschränkungen genutzter Fremdsoftware eingehalten werden, um rechtliche Konsequenzen für das DLR zu vermeiden.
2. Bestimmte Software unterliegt der Exportkontrolle (z.B. Verschlüsselungsverfahren). Es ist sicherzustellen, dass die Weitergabe des Release-Pakets bestehende Ausfuhrbeschränkungen nicht verletzt, um rechtliche Konsequenzen für das DLR zu vermeiden.

Die beschriebenen Punkte sind nicht nur für den Spezialfall des Release-Pakets zu beachten. Sie sind generell einzuhalten, wenn ein Softwarepaket an Dritte außerhalb des DLR weitergegeben wird.
Die folgenden Empfehlungen sollen sicherstellen, dass veröffentlichte Versionen der Software alle notwendigen Informationen enthalten und vor deren Freigabe grundlegende Sachverhalte geprüft werden.

Empfehlung	ab AK	Erläuterung
ERM.1: Jedes Release besitzt eine eindeutige Release-Nummer. Anhand der Release-Nummer lässt sich der zugrunde liegende Softwarestand im Repository ermitteln.	1	Zweck der Release-Nummer ist, das Release und den damit verbundenen Inhalt identifizieren zu können. Auf dieser Basis lassen sich Fehlerberichte eindeutig dem Softwarestand im Repository zuordnen. Damit vereinfachen sich Fehlersuche und -behebung. Ein Beispiel für ein häufig eingesetztes Release-Nummernformat ist: X.Y.Z (z.B. 1.0.1). Die Erhöhung einer bestimmten Stelle der Release-Nummer impliziert Aussagen zu Art und Umfang des Releases: Die Erhöhung der Hauptrelease-Nummer (X) sagt aus, dass wesentliche Neuerungen durch das Release bereitgestellt werden. Zudem ist eine Aktua-

		lisierung der Vorgängerversion ggf. nicht trivial bzw. die Änderungen sind dazu inkompatibel. Die Erhöhung der Wartungsrelease-Nummer (Y) sagt aus, dass eine Reihe neuer Funktionen und Fehlerbehebungen durch das Release bereitgestellt werden. Eine Aktualisierung der Vorgängerversion ist ohne größere Schwierigkeiten möglich. Die Erhöhung der Patchrelease-Nummer (Z) sagt aus, dass eine Reihe dringender Fehlerbehebungen durch das Release bereitgestellt werden. Eine Aktualisierung der Vorgängerversion ist ohne größere Schwierigkeiten möglich und wird dringend empfohlen. Um anhand einer Release-Nummer auf den betreffenden Softwarestand schließen zu können, ist zu empfehlen, im Versionskontrollsystem jedes Release durch einen Tag zu kennzeichnen (vgl. Abschnitt Änderungsmanagement). Der Tag-Name sollte der Release-Nummer entsprechen bzw. direkt aus ihr hervorgehen.
ERM.2: Das Release-Paket enthält oder verweist auf die Nutzer-Dokumentation. Sie besteht zumindest aus Installations-, Nutzungs- und Kontaktinformationen sowie den Release Notes. Im Fall der Weitergabe des Release-Pakets an Dritte außerhalb des DLR, sind die Lizenzbedingungen unbedingt beizulegen.	1	Durch diese Maßnahme ist sichergestellt, dass dem Nutzer ausreichend Informationen zum Betrieb der Software vorliegen. Zudem ist im Fall von Fragen oder Problemen geklärt, wie der Nutzer die Entwickler kontaktieren kann. Die Release Notes geben einen Überblick zu den wesentlichen Neuerungen und Verbesserungen des Releases. Schließlich legen die Lizenzangaben fest, unter welchen Bedingungen das DLR die Software zur Verfügung stellt. Die Nutzer-Dokumentation und die Lizenzbedingungen sind Teil der Software und sind daher im Repository abzulegen (vgl. Abschnitt Änderungsmanagement).
ERM.3: Releases werden in regelmäßigen, kurzen Abständen veröffentlicht.	2	Solange eine Software aktiv gepflegt und erweitert wird, werden idealerweise regelmäßig und in kurzen Abständen Releases (z.B. alle 3 Monate) veröffentlicht. Dadurch besteht die Chance, stetig Rückmeldung von Nutzern zu erhalten, die die Software produktiv nutzen. Die Richtung der wei-

		teren Entwicklung lässt sich somit besser steuern.
ERM.4: Die notwendigen Schritte zur Erstellung und Freigabe eines Releases sind abgestimmt und dokumentiert. Die Release-Durchführung ist weitestgehend automatisiert.	2	Dadurch sind alle erforderlichen Schritte, Verantwortlichkeiten und insbesondere die Kriterien für die Freigabe festgelegt. Die Schritte zur Erstellung eines Releases und die Freigabekriterien sind Teil des Änderungsprozesses (vgl. Abschnitt Änderungsmanagement). Daher sollten sie in dessen Rahmen beschrieben werden. Der Prozess der Release-Durchführung kann schnell komplex werden. Daher ist eine angemessene Automatisierung aller wesentlichen Schritte sicherzustellen, um Fehler zu vermeiden und den Aufwand zu reduzieren (vgl. Abschnitt Automatisierung und Abhängigkeitsmanagement).
ERM.5: Die notwendigen Schritte zur Erstellung und kurzfristigen Freigabe eines Releases für kritische Fehlerbeseitigungen sind abgestimmt und dokumentiert.	3	Wenn kritische Fehler gemeldet werden (z.B. Sicherheitslücken), kann es erforderlich sein, dass schnell und unplanmäßig ein Release zu veröffentlichen ist. Daher ist zu überlegen, wie das übliche Verfahren für diese Fälle verkürzt werden kann, um den Nutzern möglichst schnell die Fehlerbeseitigung zur Verfügung stellen zu können. Die abweichenden Schritte und Freigabekriterien sollten im Rahmen des Änderungsprozesses (vgl. Abschnitt Änderungsmanagement) beschrieben werden.
ERM.6: Während der Release-Durchführung werden alle vorgesehenen Testaktivitäten ausgeführt.	1	Je nach Software und Entwicklungskontext sind bestimmte Testaktivitäten vorgesehen, um Fehler auf Seiten der Nutzer möglichst auszuschließen (vgl. Abschnitt Software-Test). Daher ist sicherzustellen, dass vor der Freigabe des Releases diese durchgeführt worden sind. Bestehende Fehler und Probleme werden vor der Freigabe des Releases entweder behoben oder zumindest in den Release Notes dokumentiert.
ERM.7: Vor der Freigabe des Releases wurden alle vorgesehenen Tests bestanden.	2	Ein Release soll keine bekannten Fehler enthalten. Falls während der Release-Durchführung Fehler gefunden werden, sind diese zu beheben und deren Beseitigung durch nochmalige Ausführung der Tests zu überprüfen. Es ist empfehlenswert, die durchgeführten Tests mit Angaben zur Be-

		triebsumgebung, zum Ergebnis und zum Zeitpunkt festzuhalten.
ERM.8: Vor der Freigabe des Releases wurde jede Fehlerbeseitigung eines kritischen Fehlers explizit durch ein unabhängiges Review geprüft.	3	Dadurch verringert sich die Chance, dass bei der Fehlerbeseitigung bestimmte Aspekte übersehen wurden und der Fehler ggf. nur teilweise (z.B. nur in einer Betriebsumgebung) beseitigt wurde. Im Fall von Fehlern, die kritische Auswirkungen für die Nutzer haben, rechtfertigt sich dieser zusätzliche Aufwand.
ERM.9: Vor der Weitergabe des Release-Pakets an Dritte außerhalb des DLR ist sicherzustellen, dass eine Lizenz festgelegt ist, die Lizenzbestimmungen verwendeter Fremdsoftware eingehalten werden und alle erforderlichen Lizenzinformationen dem Release-Paket beigelegt sind.	1	Fast jede Software nutzt kommerzielle oder unter einer Open-Source-Lizenz stehende Fremdsoftware. Durch die Weitergabe der Fremdsoftware im Rahmen der eigenen Software, sind je nach Lizenz und Art der Nutzung bestimmte Bedingungen einzuhalten. Möglicherweise wird dadurch die eigene Lizenzwahl eingeschränkt. Damit rechtliche Konsequenzen durch die Verletzung von Lizenzbestimmungen für das DLR vermieden werden, sind die genannten Sachverhalte zu prüfen. Es ist zu empfehlen, frühzeitig festzulegen, unter welcher Lizenz (kommerziell, Open Source) die eigene Software weitergegeben werden soll. Dadurch lässt sich die Lizenzkompatibilität bereits bei der Auswahl konkreter Fremdsoftware berücksichtigen. In der Konsequenz fällt der Aufwand für die Lizenzprüfung nur punktuell an und man vermeidet eventuelle umfangreichere Nacharbeiten. Weitere Informationen und Ansprechpartner zum Thema Open-Source-Nutzung können der Broschüre "Nutzung von Open Source Software im DLR" entnommen werden.
ERM.10: Vor der Weitergabe des Release-Pakets an Dritte außerhalb des DLR ist sicherzustellen, dass die Regelungen zur Exportkontrolle eingehalten werden.	1	Bestimmte Software unterliegt der Exportkontrolle (z.B. Verschlüsselungsverfahren). Daher ist vor der Weitergabe der Software an Partner oder externe Organisationen zu prüfen, ob diese durch eine gültige Sanktionsliste erfasst oder in einem Land beheimatet sind, das einer Genehmigungspflicht unterliegt. Falls dies zutrifft, ist zu prüfen, ob die Software für die Exportkontrolle relevante Bestandteile enthält. Bei positivem Ergebnis sind die

		sich daraus ergebenden Pflichten und Verbote konsequent einzuhalten, um rechtliche Konsequenzen für das DLR zu vermeiden.
ERM.11: Jeder Schritt der Release-Durchführung und dessen Ergebnis ist protokolliert. Alle wesentlichen Artefakte (z.B. Release-Paket, Testprotokolle) werden langfristig und sicher gespeichert.	3	Dadurch sind langfristig ausreichend Informationen vorhanden, um die Release-Durchführung nachzuvollziehen und um das Release ggf. reproduzieren zu können. Zudem lassen sich Ursachen für ggf. vorhandene Fehler und Probleme des Releases ermitteln. Praktisch lässt sich diese Empfehlung durch einen hohen Automatisierungsgrad der Release-Durchführung sicherstellen (vgl. Abschnitt Automatisierung und Abhängigkeitsmanagement).

4.8 Automatisierung und Abhängigkeitsmanagement

Software-Entwicklung ist komplex. Die zu erstellende Software allein ist i.d.R. bereits sehr umfangreich. Zusätzlich benötigt man verschiedene Softwarepakete (**Abhängigkeiten**) in der richtigen Version, um die Software zu erstellen. Weiterhin sind diverse weitere Programme (**Entwicklungsumgebung**) erforderlich, damit Entwickler neue Funktionen effizient erstellen und deren Qualität sicherstellen können. Schließlich muss die Software unter verschiedenen **Betriebsumgebungen** funktionieren. Daher ist es erforderlich, dies für jede unterstützte Betriebsumgebung explizit durch Tests sicherzustellen. Benötigte **Testumgebungen** sind bereitzustellen und zu pflegen.

Ohne die Automatisierung wiederkehrender Abläufe ist die beschriebene Komplexität nicht zu beherrschen. Insbesondere die einzelnen Schritte des Build-Prozesses sind konsequent zu automatisieren. Der **Build-Prozess** bezeichnet das Verfahren, welches aus dem Quelltext und den Abhängigkeiten das lauffähige Programm erstellt (**einfacher Build-Prozess**). Im erweiterten Sinn gehören dazu auch die Ausführung von Tests (vgl. Abschnitt Software-Test) und das Erstellen des Release-Pakets (vgl. Abschnitt Release-Management). Im Folgenden wird dafür der Begriff "**erweiterter Build-Prozess**" verwendet.

Zur Automatisierung des Build-Prozesses existieren je nach Programmiersprache spezialisierte **Build-Werkzeuge** (z.B. Maven, CMake). Auf dieser Basis erstellt man das **Build-Skript**, welches den Build-Prozess einer konkreten Software automatisiert. Vielfach erlauben **integrierte Entwicklungsumgebungen** (z.B. Eclipse) den Build-Prozess "per Knopfdruck" automatisiert durchzuführen. Dazu bauen diese entweder auf bestehenden Build-Werkzeugen auf oder nutzen eine eigene Implementierung.

Zusätzlich unterscheidet man verschiedene **Build-Varianten**, die sich in Bezug auf den Zweck, die durchzuführenden Build-Schritte und die erforderliche Ablaufumgebung unterscheiden:

- Der **Entwickler-Build** (auch Private Build genannt) erstellt und prüft die Software in der lokalen Entwicklungsumgebung. Der Entwickler nutzt diese Build-Variante, um die Auswirkungen seiner/ihrer Änderungen zu überprüfen.
- Der **Integrations-Build** prüft die Änderungen aller beitragenden Entwickler in einer neutralen Testumgebung. Es werden ggf. weitere Build-Schritte (z.B. spezielle Tests, Erstellung des potentiellen Release-Pakets) durchgeführt, die aus Effizienzgründen im Entwickler-Build entfallen. Der Integrations-Build wird meist zeit- oder ereignisgesteuert ausgelöst. Daher muss der Build-Prozess ohne manuelle Einwirkung ausgeführt werden können. Ein automatisierter Build-Prozess auf Basis einer integrierten Entwicklungsumgebung scheidet daher meist aus.
- Der **Release-Build** erstellt auf der Basis eines freigegebenen Softwarestands das ausliefer-bare und mit der Release-Nummer gekennzeichnete Release-Paket. Er stellt eine Erweiterung des Integrations-Builds dar. Es werden ggf. weitere Build-Schritte durchgeführt, die die Release-Erstellung nachbereiten (z.B. die Verteilung des Release-Pakets an die Nutzer).

Viele Schritte der Software-Entwicklung lassen sich erst durch den automatisierten Build-Prozess effizient durchführen. Zudem vermeidet dieser Fehler, da die Beteiligten von Routineaufgaben entlastet werden. Schließlich liefert der Build-Prozess die Basis dafür, dass erreichte Entwicklungsstände reproduziert werden können.

Die folgenden Empfehlungen sollen einen angemessenen Einsatz von Automatisierungstechniken zur Effizienzsteigerung und einen strukturierten Umgang mit den Abhängigkeiten sicherstellen.

Empfehlung	ab AK	Erläuterung
EAA.1: Der einfache Build-Prozess läuft grundlegend automatisiert ab und notwendige manuelle Schritte sind beschrieben. Zudem sind ausreichend Informationen zur Betriebs- und Entwicklungsumgebung vorhanden.	1	Ein automatisierter Build-Prozess hilft den Entwicklern, effizienter neue Funktionen zu erstellen. Der Build-Prozess lässt sich i.d.R. durch einen einfachen Skriptaufruf oder über eine integrierte Entwicklungsumgebung starten. Durch diese Maßnahme verringert sich die Komplexität, da nicht jeder Entwickler alle Details der verwendeten Programme und deren Einstellungen kennen muss. Ergänzend benötigt man i.d.R. einige Zusatzinformationen. Beispielsweise müssen zur Verwendung des Build-Skripts weitere Abhängigkeiten manuell installiert und deren Installationsverzeichnis per Parameter übergeben werden. Daher ist es empfehlenswert, den grundlegenden Build-Ablauf und insbesondere detailliertere Informationen zur Entwicklungsumgebung als Teil der Einstiegsdokumentation für Entwickler (vgl. Abschnitt Änderungsmanagement) zu beschreiben. Schließlich ist auch die notwendige Betriebsumgebung zu dokumentieren (z.B. als Teil der Installationsanlei-

		tung, vgl. Empfehlung ERM.2).
EAA.2: Die Abhängigkeiten zum Erstellen der Software sind zumindest mit dem Namen, der Versionsnummer, dem Zweck, den Lizenzbestimmungen und der Bezugsquelle beschrieben.	1	Insbesondere die Lizenzinformationen liefern die Grundlage, um die Pflichten und Beschränkungen im Fall der Weitergabe der Software an Dritte außerhalb des DLR einschätzen zu können (vgl. Empfehlung ERM.9). Diese Dokumentationspflicht ist auch in der Broschüre "Nutzung von Open Source Software im DLR" aufgeführt.
EAA.3: Neue Abhängigkeiten werden auf Kompatibilität zur angestrebten Lizenz überprüft.	2	Die Lizenzbestimmungen genutzter Fremdsoftware können u.a. die eigene Lizenzwahl beschränken. Daher ist bei der Auswahl von Fremdsoftware zu beachten, dass deren Lizenzbestimmungen kompatibel zur angestrebten Lizenz sind. Falls bisher keine eigene Lizenz festgelegt ist, sollte zumindest darauf geachtet werden, dass die Lizenzbestimmungen möglichst wenige Pflichten und Einschränkungen auferlegen (vgl. Empfehlung ERM.9).
EAA.4: Die Abhängigkeiten werden langfristig und sicher gespeichert.	3	Teilweise greifen Build-Werkzeuge (z.B. Maven) auf öffentliche Repository zurück, um benötigte Fremdsoftware lokal zu installieren und darauf aufbauend die Software zu erstellen. Dadurch kann im Laufe der Zeit das Problem entstehen, dass ein bestimmter Stand der Software nicht mehr reproduziert werden kann, da eine Abhängigkeit in der erforderlichen Version nicht mehr zur Verfügung steht. In manchen Fällen kann es auch erforderlich sein, dass dazu Teile der Betriebsumgebung vorzuhalten sind. Daher ist insbesondere für Releases zu empfehlen, zumindest die Abhängigkeiten langfristig sicher zu speichern, um ggf. bestehenden Gewährleistungspflichten nachkommen zu können.
EAA.5: Im Build-Prozess laufen die Ausführung von Tests, die Ermittlung von Metriken, die Erstellung des Release-Pakets und ggf. weitere Schritte automatisiert ab.	2	Die Automatisierung des erweiterten Build-Prozesses bildet eine wichtige Grundlage für die effiziente Entwicklung (vgl. Empfehlung EDI.3) und das systematische Testen (vgl. Abschnitt Software-Test). Insbesondere schafft er die Voraussetzung, dass sich wesentliche Aspekte des Änderungsprozesses (vgl. Abschnitt Änderungs-

		management) und der Release-Durchführung (vgl. Abschnitt Release-Management) praktikabel umsetzen lassen. Es ist zu empfehlen, das Standardverhalten des Build-Prozesses auf den Entwickler-Build auszulegen, da Entwickler die Hauptzielgruppe darstellen. Beispielsweise können Entwickler auf dieser Basis leicht feststellen, dass zumindest in der lokalen Entwicklungsumgebung keine Regressionen auftreten (vgl. Empfehlung EST.2). Zudem lassen sich die Einhaltung von Vereinbarungen wie z.B. einfache Regeln des Programmierstils (vgl. Empfehlung EDI.1) effizient prüfen. Der automatisierte Build-Prozess stellt in diesem Zusammenhang eine einfach zu nutzende Schnittstelle bereit. Die Entwickler benötigen kein Detailwissen über zusätzliche Testwerkzeuge oder wie auf Testdaten zuzugreifen ist.
EAA.6: Der Build-Prozess protokolliert alle wesentlichen Schritte und lässt insbesondere die zur Erstellung verwendeten Abhängigkeiten inklusive derer Versionen nachvollziehen.	3	Dies ist eine wichtige Voraussetzung, um gezielt einen bestimmten Softwarestand reproduzieren zu können (vgl. Abschnitt Release-Management, Empfehlung ERM.11). Beispielsweise lassen sich damit in der Entwicklungsumgebung die Ursachen für vorhandene Fehler gezielt untersuchen.
EAA.7: Erforderliche Testumgebungen können automatisiert bereitgestellt werden.	3	Testumgebungen können schnell recht komplex werden. Die zentrale Bereitstellung, Pflege und Nutzung kann sich einerseits zu einem Ressourcenengpass entwickeln, andererseits können unbemerkt Änderungen der Testumgebung Fehler verursachen. Daher ist es sinnvoll, die Bereitstellung der Testumgebung weitestgehend zu automatisieren. Eine wesentliche Basis bilden Techniken zur Virtualisierung (z.B. Docker-Container) und Systemkonfigurationswerkzeuge (z.B. Ansible). Letztere erlauben es, automatisiert mehrere Systeme zu konfigurieren. Somit können die Konfigurationsparameter der Testumgebung im Repository (vgl. Empfehlung EAA.10) abgelegt und auf dieser Basis die Testumgebung reproduziert werden.

EAA.8: Ein Integrations-Build ist eingerichtet.	2	Der Integrations-Build erlaubt es die Änderungen aller Entwickler regelmäßig im Zusammenspiel zu überprüfen. Dadurch lassen sich Integrationsfehler frühzeitig erkennen und beheben. Der Aufwand für die Fehlerbeseitigung fällt dadurch i.d.R. geringer aus. Eine späte Integration führt häufig dazu, dass Meilensteine nicht wie geplant erreicht werden können. Dazu müssen alle Entwickler regelmäßig ihre Änderungen ins Repository übertragen. Auf dieser Basis kann der Integrations-Build die Software überprüfen und liefert das Ergebnis an die Entwickler zurück. Falls Probleme durch den Integrations-Build sichtbar werden, sind diese direkt zu beheben. Es ist zu empfehlen, diese Arbeitsweise im Änderungsprozess zu verankern (vgl. Abschnitt Änderungsmanagement). Voraussetzung für einen effizienten Integrations-Build ist die Automatisierung des erweiterten Build-Prozesses. Dabei müssen alle Schritte ohne manuellen Eingriff ausgeführt werden können und die relevanten Testumgebungen zur Verfügung stehen. Zur technischen Realisierung wird häufig auf Web-basierte Werkzeuge für die kontinuierliche Integration zurückgegriffen (z.B. Jenkins). Diese erlauben es, den Build- und Teststatus übersichtlich anzuzeigen, und bieten auch Funktionen zur Trendanalyse von Metriken (vgl. Abschnitt Software-Test, Empfehlungen EST.7 und EST.8).
EAA.9: Ein Release-Build ist eingerichtet.	3	Der Release-Build automatisiert konsequent alle wesentlichen Schritte der Release-Durchführung (vgl. Abschnitt Release-Management), um Fehler weitestgehend ausschließen zu können. Dies umfasst beispielsweise die Erstellung des Release-Pakets, die Installation des Releases in den relevanten Testumgebungen, die Prüfung des Releases und ggf. auch die Verteilung und Installation des Releases in der Betriebsumgebung (vgl. Continuous Delivery). Je nach Entwicklungskontext ist zu überlegen, welche dieser Schritte sich tatsäch-

		lich sinnvoll automatisieren lassen. Voraussetzungen für einen effizienten Release-Build sind die Automatisierung des erweiterten Build-Prozesses und die Verfügbarkeit der relevanten Testumgebungen. Häufig wird als Ausgangspunkt der Integrations-Build genutzt und um zusätzliche Schritte ergänzt.
EAA.10: Das Repository enthält möglichst alle Bestandteile, um den Build-Prozess durchführen zu können.	1	Dazu zählen beispielsweise Informationen zu den grundlegenden Build-Schritten und den Abhängigkeiten, das Build-Skript sowie Konfigurationsdateien der integrierten Entwicklungsumgebung und Testwerkzeuge. Auf dieser Basis ist es möglich, erreichte Zwischenstände wiederherstellen zu können. Zudem lassen sich einfacher Fehler erkennen, die auf geänderte Einstellungen des Build-Prozesses beruhen.