Varianten und Herausforderungen der Zusammenarbeit in der Software-Entwicklung - Ein Vergleich verschiedener Herangehensweisen und Werkzeuge

Ingmar Bauckhage[1] Vsevolod Pypenko[2]

Abstract: Zusammenarbeit ist bei der Entwicklung von Software-Projekten von großer Bedeutung. Dieser Beitrag vergleicht vier Vorgehensmodelle in Bezug auf die Zusammenarbeit: Wasserfall-Modell, V-Modell, Rational Unified Process und Scrum. Es werden Unterschiede herausgearbeitet sowie Vor- und Nachteile benannt. Die Zusammenarbeit während der Implementierung und des Qualitätsmanagements wird näher betrachtet.
Herausforderungen sind z. B. die gemeinsame Arbeit am Quellcode. Hier existieren verschiedene Workflows für die Werkzeuge Git und Github, die veranschaulicht und bewertet werden.
Im Qualitätsmanagement ist vor allem die Zusammenarbeit von verschiedenen Teams notwendig. Dies kann über regelmäßigen Austausch mithilfe von formalen Dokumenten und Meetings sichergestellt werden. Darüber hinaus wird beschrieben, welche verschiedenen Möglichkeiten der Verifikation und des Testens es gibt und für welche Projekte diese notwendig sind.

Keywords: Collaboration; Zusammenarbeit; Qualitätsmanagement; Quellcode-Verwaltung; Wasserfall-Modell; V-Modell; RUP; Scrum; Git; Branching

1 Einführung

Die Entwicklung von großen Software-Projekten kann nur durch Zusammenarbeit gelingen. Dies ist vor allem durch die steigende Komplexität moderner Software bedingt [Wi22]. Um die steigende Komplexität zu beherrschen haben sich eine Vielzahl an Vorgehensmodellen, Strategien und Werkzeugen entwickelt. Welche Unterschiede bestehen dabei zwischen den einzelnen Modellen in Bezug auf die Zusammenarbeit und welche Empfehlungen können für die Wahl ausgesprochen werden?

1.1 Motivation

Wie soll der Begriff *groß* im Rahmen dieser Arbeit verstanden werden? Ein *großes* Software-Projekt soll hier insbesondere durch die Anzahl an Entwicklerinnen und Entwickler gekennzeichnet sein. Ein konkreter Vorschlag wäre, ein Software-Projekt *groß* zu

[1] DHBW Stuttgart Campus Horb, TINF2020, Florianstraße 15, 72160 Horb am Neckar, Deutschland i20003@hb.dhbw-stuttgart.de
[2] DHBW Stuttgart Campus Horb, TINF2020, Florianstraße 15, 72160 Horb am Neckar, Deutschland i20028@hb.dhbw-stuttgart.de

nennen, wenn mehr als drei Entwicklerinnen und Entwickler zusammenarbeiten. Dies ist eine recht willkürliche Festlegung, entscheidend soll aber insbesondere die Abgrenzung zur Einzel-Entwicklung sein.
Auch in kleineren und mittleren Unternehmen steigen die Anforderungen an Software, während die Anzahl der Entwicklerinnen und Entwickler meist nicht so schnell gesteigert werden kann. Dadurch fehlen dedizierte Mitarbeiterinnen oder Mitarbeiter für das Management der Zusammenarbeit, während in größeren Unternehmen oft komplette Abteilungen alleine hierfür zuständig sind. In der Folge kommt es zu Problemen in der Zusammenarbeit und daraus resultierend einer geringeren Produktivität oder Fehler in der Software.
Die Zusammenarbeit im Team während eines Projekts ist entscheidend für den Ausgang des Projektes. Dabei spielt die Kommunikation zwischen den Teammitgliedern eine enorme Rolle, vor allem je größer das Projekt und somit die Anzahl der Beteiligten ist [Ve00]. Um diese Herausforderung zu bewältigen haben sich verschiedene Modelle entwickelt, auf die im Folgenden kurz eingegangen wird. Außerdem kommt es bei der gemeinsamen Bearbeitung von Quellcode oft zu Konflikten, die aufwändig gelöst werden müssen. Hierfür sollen in dieser Arbeit Strategien vorgestellt werden, um diese Konflikte zu minimieren.

1.2 Vorgehensmodelle in der Software-Entwicklung

Vorgehensmodelle bieten die Möglichkeit ein Rahmenwerk mit festen Strukturen und Prozessen festzulegen, um häufig wiederkehrende Aufgaben bei der Software-Entwicklung zu erleichtern und zu standardisieren. Sie stellen im Grunde einen Plan dar, welcher den Entwicklungsprozess in überschaubare, zeitlich und inhaltlich begrenzte Phasen aufteilt und festlegt, wie die Übergänge zwischen den Phasen zu gestalten sind und welche Werkzeuge oder Werkzeug-Prototypen verwendet werden sollen. Auch die Arten der Zusammenarbeit sind hier mehr oder weniger festgelegt.
Dabei haben sich eine Vielzahl an konkurrierenden Modellen entwickelt, die von ihren jeweiligen Schöpfern angepriesen werden. Wenn eine grobe Aufteilung vorgenommen werden soll, kann unterschieden werden in die *klassischen* Vorgehensmodelle, die oft aus akademischer oder institutioneller Umgebung stammen und agile Methoden, die sich aufgrund von Problemen mit den bereits vorhandenen Vorgehensmodellen entwickelt haben. Die Klassifizierung von Vorgehensmodellen lässt sich noch weiter verfeinern. Für diese Arbeit reicht aber die übergeordnete Unterteilung und gegebenenfalls wird bei Bedarf direkt darauf eingegangen.

1.3 Zielsetzung

Diese Arbeit soll einen Überblick über die Zusammenarbeit in verschiedenen Vorgehensmodellen der Software-Entwicklung geben und Unterschiede herausarbeiten. Dabei sollen Vor- und Nachteile für verschiedene Einsatzzwecke aufgezeigt werden. Für die Phasen der Implementierung und der Qualitätssicherung soll außerdem eine nähere Betrachtung

erfolgen und Werkzeuge und Methoden vorgestellt und bewertet werden, die bei der Lösung der genannten Probleme helfen können.

2 Zusammenarbeit in ausgewählten Vorgehensmodellen

Im Folgenden werden die vier Vorgehensmodelle *Wasserfallmodell*, *V-Modell*, *Rational Unified Process* und *Scrum* in Bezug auf die Zusammenarbeit kurz vorgestellt und miteinander verglichen.

2.1 Wasserfall-Modell

Das Wasserfallmodell wurde im Jahr 1970 von Dr. Winston W.Royce in einem Paper vorgestellt. Somit ist dieses Modell auch einer der ältesten und bekannten Vorgehensmodellen in der Softwareentwicklung. Trotz des Alters findet das Modell immer noch Einsatz in vielen großen Unternehmen [AA15c][AA15b].
Das Vorgehensmodell stellt die Projektphasen in einem „Wasserfall“ dar. Dabei werden die einzelnen Phasen top-down nacheinander abgearbeitet und sind in sich abgeschlossen. Das heißt, dass bevor eine Phase komplett abgearbeitet wird, darf die Nächste nicht angefangen werden. Darüber hinaus lässt das reine Wasserfall-Modell keine Rückführung zu den vorherigen, abgeschlossenen Phasen, zu [Go11, S. 84].
Als Grundlage für den gesamten Entwicklungsprozess dient die Spezifikation. So werden auch die Anforderungen für Anwender von vornherein ermittelt und festgeschrieben. Die Ergebnisse einer Phase fallen in die nächste, um dort weiterverarbeitet zu werden. Zur Kommunikation zwischen den Entwicklern und Anwendern werden umfangreiche Dokumente über das Softwaresystem erstellt [Go11, S. 83]. Die wichtigsten Phasen des Modells lassen sich folgendermaßen definieren[AA15c, S. 85][Go11]:

1. Anforderungsdefinition: in Zusammenarbeit mit dem Kunden werden die Spezifikationen, Ziele und Anforderungen an den Softwareprodukt definiert. Anschließend wird eine umfassende Dokumentation erstellt, die die Grundlage der weiteren Arbeit bildet.
2. Software- und System-Entwurf: die Anforderungen werden bearbeitet und es werden abstrakte Softwarekomponenten sowie die grundlegende Softwarearchitektur modelliert
3. Implementierung: in dieser Phase werden die modellierten Software-Komponenten programmiert.
4. Integration und Testen: die einzelnen Elemente des Software-Systems werden zusammengeführt und es werden Tests durchgeführt.
5. Betrieb und Wartung: das erstellte Produkt wird zum Gebrauch freigegeben.

Das Wasserfallmodell, bereits in der einfachen Ausführung, bietet eine gute Möglichkeit, die Anwenderteams bei kleineren Projekten wie z.B. Prototypen, zu organisieren. Darüber hinaus lässt sich das Modell durch weitere Funktionalitäten erweitern. Zum Beispiel, um die Qualität der Software zu verbessern und eine höhere Flexibilität in die Projekte zu schaffen, wurde das sogenannte „Wasserfallmodell mit Rückführschleifen“ entwickelt. Dabei finden Validierungen und Verifikationen der vorherigen Phase beim Übergang in die nächste Phase statt und es lässt sich ein „Rückschritt“ über mehrere Phasen implementieren [Go11, S. 86, 87]. Die graphische Abbildung des oben beschriebenen Modells wird in Abbildung 1 dargestellt. Dabei wird das einfache Wasserfallmodel mit den roten Pfeilen und das erweiterte mit blauen dargestellt.

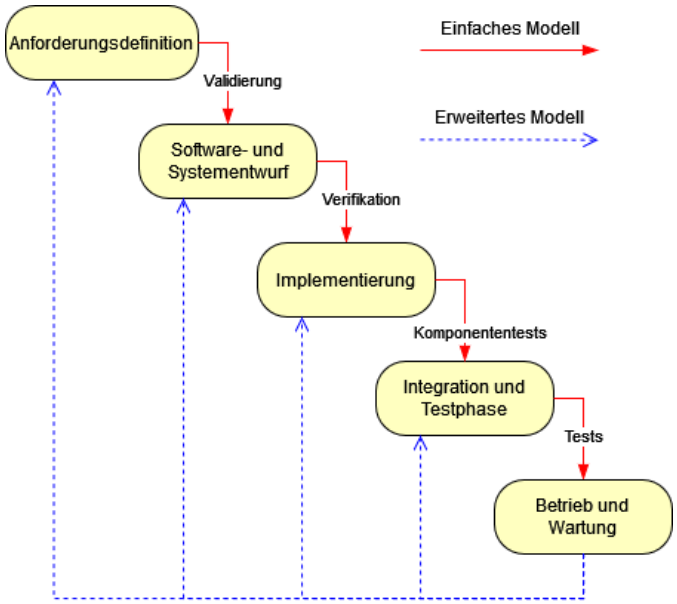

Abb. 1: Das einfache Wasserfallmodell

Die Vorteile des Wasserfallmodells liegen in der klaren, einfachen Struktur des Modells. Die Abläufe werden klar und verständlich in Phasen aufgeteilt wodurch sich die Ergebnisse kontrollieren lassen. Dadurch können die Aufgaben während einer Phase klar auf die Entwicklerteams aufgeteilt und verwaltet werden und jede Phase kann fehlerarm ablaufen. Darüber hinaus lässt sich das Modell mithilfe verschiedener Ansätze erweitern wodurch verschiedene Aspekte der Verwaltung und Softwareentwicklung, z.B. Qualitätssicherung verbessert werden [Go11, S. 84–87] [Ky19, S. 14].
Die Nachteile des Modells basieren zum einen auf der Starrheit und zum anderen auf dem viel zu großem Anteil der Dokumentation in dem Modell. Aufgrund der Starrheit lässt sich nicht so schnell auf die Veränderungen der Anforderungen und Spezifikation reagieren und dadurch können erhebliche Zeit- und Geldkosten anfallen. Dieses Problem

wird zwar teilweise durch das erweiterte Wasserfall Modell gelöst, jedoch genügt es nicht ganz, wenn die Anforderungen oft verändert werden müssen. Die Dokumentation, die umfänglich während des Projektablaufs geführt wird, kann auch unter Umständen einen viel zu großen Stellenwert im Projekt übernehmen, wodurch die eigentlichen Ziele nicht vollständig erreicht werden [Ky19, S. 14] [AA15c].
Bezogen auf die Zusammenarbeit, verfügt das Modell über zwei Ansätze: es findet ein Austausch zwischen Entwicklern innerhalb eines Teams während einer Phase und eine Kommunikation mithilfe von Dokumenten zwischen verschiedenen Phasen. Durch diese Ansätze kann zwar eine verständliche, fest vorgegebene Kommunikation geführt und eine Kontrolle angesetzt werden, aber bei größeren Projekten, kann dies zu einer langen Projektlaufzeit führen. Die Starrheit des Modells wird dabei nicht aufgelöst, was zu großen Problemen bei Anforderungsänderungen und anderen ungeplanten Geschehen führen kann.

2.2 V-Modell

Ein weiteres sequenzielles Modell, welches heutzutage häufig in deutschen Behörden und großen Unternehmen verwendet wird, ist das V-Modell [Hö08]. Wie der Name bereits bezeichnet, werden die einzelnen Projektphasen im Modell in Form eines „V“ dargestellt. Das V-Modell wurde als Weiterentwicklung des Wasserfallmodels entwickelt und liegt dabei den Fokus vor allem auf der Qualitätssicherung. Diese wird durch Einführung der Verifikation und Validierung mithilfe von Tests nach jeder Projektphase ermöglicht [AA15a][Va14].
Das V-Modell entstand bereits in 80er Jahren und stellte als Modell-Prototyp den Grundgerüst der modernen Vorgehensmodelle für die Bundeswehr in Deutschland her [DW99, S. 1]. Dieser Prototyp wurde im Laufe der Jahre bearbeitet und im Jahr 1992 für alle Bundesbehörden in Deutschland in einer verbindlichen Fassung als Rahmenregelung empfohlen [DW99, S. 3]. Diese Fassung beschrieb den Arbeitsprozess „statisch, als Abfolge einzelner Arbeitsschritte“ [DW99](S 3) wodurch dynamische Aspekte des Entwicklungsprozesses wie z.B. Kombinierung der verschiedenen Phasen miteinander, unbeschrieben wurden. Dieses und andere Probleme des Modells führten dazu, dass das Modell überarbeitet und weiterentwickelt werden musste. Dies führte zur Standardisierung des V-Modells 97 im Jahre 1997 [DW99, S. 3, 4]. Die neuen Projekte und Entwicklungen in Methodik und Technologie wurden in diesem Modell dennoch unzureichend beschrieben und konnten somit nicht dem Stand der Technik entsprechen. Als Antwort auf diese Probleme wurde im Jahre 2004 das V-Modell XT vorgestellt. XT steht dabei für „extreme Tailoring“ („extremes Maßschneidern“) und drückt die angestrebte Flexibilität des Modells aus [Hö08, S. 3]. Das V-Modell XT ist die neueste Version des V-Modells und wird fortlaufend aktualisiert. Diese Version wird im Folgenden näher beschrieben.

Das V-Modell XT beschreibt die Erstellung eines IT-Systems als eine Folge von Aktivitäten, bei denen Produkte erstellt werden [DW99, S. 4]. Nach dem Erstellen eines Produktes

wird dieser gegen die gestellten Anforderungen geprüft (**Verifikation**)[Fr09, S. 107]. Zum anderen, ist auch zu prüfen, ob das erstellte Produkt für die vorgesehene Aufgabe geeignet ist, also ob z.B. das richtige System in Auftrag gegeben wurde. (**Validierung**) In der Software-Entwicklung erfolgt die **Verifikation** durch die ausgebildeten Prüfer, welche bei der Übergabe der Produkte, die in Form von formalen Dokumenten bestehen, mitwirken [Hö08, S. 36, 37] [Fr09, S. 107]. Die **Validierung** erfolgt dabei oft mithilfe des „Test-first" Ansatzes. Dabei werden, angefangen bereits bei der Anforderungsdefinition bis hin zur Implementierungsphase, Tests und Abnahmekriterien für die Projektbausteine definiert [Va14, S. 2].
Insgesamt lässt sich das Modell wie in Abbildung 2 beschreiben. Dabei findet in dem Modell von Oben nach Unten eine schrittweise Verfeinerung des Gesamtsystems. Am Anfang findet eine Anforderungsdefinition, die dann bis zum Modulentwurf und deren Implementierung zerlegt wird. Nachdem man dann im Modell unten „angekommen ist", wird das System wieder rekursiv aufgebaut und dabei die vorher geschriebenen Tests bis zur Abnahme durchgeführt [Go11, S. 93][LL07]. Das V-Modell XT ist insgesamt in 4 Submodelle gegliedert. Das im

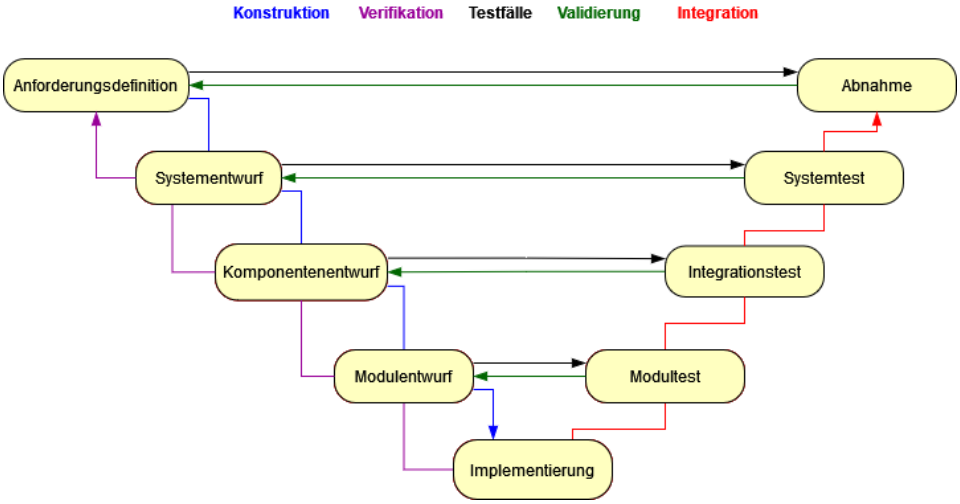

Abb. 2: Vereinfachte Darstellung des V-Modells XT

Bild 2 dargestelltes Modell beschreibt dabei das Submodell „Softwareentwicklung". Zu den weiteren Tätigkeiten gehören: Qualitätssicherung, Konfigurationsmanagement und das Projektmanagement. Die Vier Modelle sind miteinander verbunden, und sorgen zusammen für einen kontrollierten, sicheren Projektablauf [Fr09, S. 6] [DW99, S. 4].
Der Kernunterschied des V-Modells XT zu ihrem Vorgänger, dem V-Modell XT liegt in der Anpassbarkeit und Flexibilität. Dabei wurde das Modell so entwickelt, dass es für verschiedenen Projekten angewendet werden kann. Zu den Maßnahmen gehören zum Beispiel, die anpassbare Größe der Dokumentation, damit „so viel . . . wie nötig, aber so wenig wie möglich" erzeugt wird [Fr09, S. 3]. Darüber hinaus lässt sich in dem Modell die Anzahl von Phasen oder Vorgehensbausteinen und die Zuordnung von Mitarbeiter-Rollen

jeweils spezifisch für das Projekt zuordnen. Diese Methode wird als „Tailoring“ bezeichnet. [Fr09, S. 6]
Insgesamt lässt sich sagen, dass das Modell gut für verschiedene Projekttypen und -Größen einsetzbar ist [Fr09, S. 3]. Wie bereits oben beschrieben, findet das V-Modell insbesondere in der Industrie und bei IT-Projekten der Bundesbehörden Einsatz. Darüber hinaus, lassen sich verschiedene Projektdurchführungsstrategien innerhalb des V-Modells auswählen, z.B. iterativ oder evolutionär, welche für unterschiedliche Arten von Projekten anwendbar sind [Hö08, S. 7]. Insbesondere kann das Modell in Projekten verwendet werden, die feste, verständliche und wenig veränderbare Anforderungen haben [Va14]. Die Größe dieser Projekte sollte auch nicht zu gering sein, weil ein großer Aufwand in der Planung eingesetzt werden muss.
Die Vorteile des Modells liegen in der Qualitätssicherung und strikten Kontrolle innerhalb des Modells. Die häufige Verwendung des „Test First“ Ansatzes und die Maßnahmen zur Verifikation und Validierung ermöglichen einen fehlerarmen Ablauf des Projekts und die Erstellung eines qualitativ hohen Endprodukts. Durch das neu eingeführte „Tailoring“ im V-Modell XT ist das Modell auch flexibler geworden und kann somit für verschiedene Projekte benutzt werden.
Die Nachteile des Modells leiten sich vor allem aus der Starrheit des Modells heraus. Da das Modell linear ist, können die einzelnen Projektphasen nach dem das Projekt angefangen ist, nicht einfach umstrukturiert werden. Darüber hinaus müssen bei dem Modell die Anforderungen strikt formuliert werden, da die Änderungen dieser schwer in der Mitte des Projekts umgesetzt werden. [AA15a] [BK08]
Wie bereits oben beschrieben, ist das V-Modell in 4 Submodelle gegliedert. Dadurch lassen sich die Rollen der Qualitätsmanager, der Softwarearchitekten und anderen Mitarbeiter voneinander trennen und jeder Mitarbeiter befindet sich in seinem Zuständigkeitsgebiet. Die Kommunikation wird dabei wie bei dem Wasserfallmodell mithilfe von Dokumenten geregelt. Darüber hinaus lassen sich unterschiedliche Arten von Austausch in Unternehmen regeln.

2.3 Rational Unified Process

Der *Rational Unified Process (RUP)* ist zum einen ein Vorgehensmodell, zum anderen auch die zugehörigen Entwicklungswerkzeuge, die von IBM vertrieben werden. Entstanden ist RUP als Kombination der Prozesse der Firmen *Rational* und *Objectory*. Kombiniert werden dabei ein iterativer, inkrementeller Softwareprozess mit architekturzentrierter Vorgehensweise und die objektorientierte Darstellung, meist in Form von Use-Case-Diagrammen. [Hu15, S. 49, 50]
Im Gegensatz zu z. B. dem Wasserfallmodell überschneiden sich die einzelnen Phasen bei RUP und unterscheiden sich vor allem in ihrer Intensität voneinander in zeitlicher Hinsicht. Abbildung 3 veranschaulicht hierbei die zwei Dimensionen von RUP. Die vier Phasen sind dabei *Konzept*, *Entwurf*, *Implementierung* und *Produktübergabe*. Zu erkennen ist, dass die Implementierung bereits während der Konzeptphase beginnt und die Analyse und Design

auch während der Implementierung noch aktiv ist. Dadurch kann auf Anforderungsänderungen im Projektverlauf eingegangen werden. Zudem findet eine deutlich ausgeprägtere Kommunikation zwischen den verschiedenen Teams statt [Ve00, S. 6, 7].
Im Unterschied zu z. B. Scrum ist RUP ein kommerzielles Produkt. Es kann ohne Anpassungen direkt verwendet werden, bietet aber auch die Möglichkeit der Erweiterung und Anpassung [Kr99, S. 31].

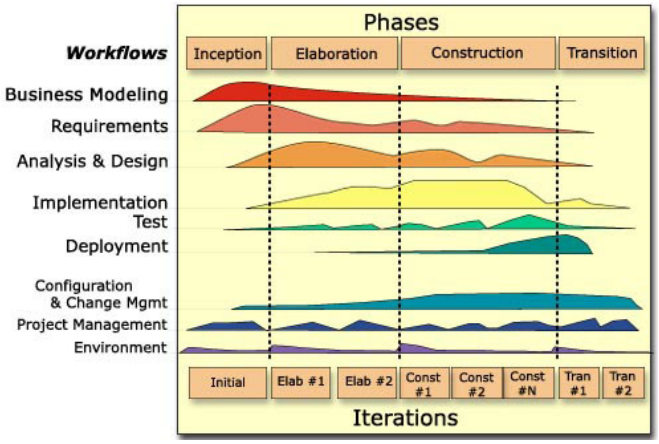

Abb. 3: Dimensionen von RUP. Übernommen von [Kr00]

2.4 Scrum

Scrum ist ein, vor allem von *Ken Schwaber* und *Jeff Sutherland* entwickeltes Vorgehensmodell, das der agilen Software-Entwicklung zugerechnet werden kann. Es wird seit den 1990er Jahren entwickelt und findet seitdem eine steigende Verbreitung [SS20, S. 1] [re18].
Einer der Kerngedanken von Scrum ist die selbstständige Arbeit der Mitarbeiterinnen und Mitarbeitern und das regelmäßige Reflektieren und Anpassen von Methoden und Vorgehensweisen um möglichst immer optimale Resultate zu erzielen. Um das zu erreichen teilt Scrum komplexe Probleme in kleinere Probleme auf, die für sich erarbeitet und bewertet werden können [SS20, S. 3].
Dabei gibt Scrum keine detaillierten Anweisungen vor, wie etwas zu tun ist, sondern lediglich einen Rahmen, der dafür essentiell ist. Innerhalb dieser Freiheit soll die Expertise der Mitarbeiterinnen und Mitarbeitern dafür sorgen, dass die nötigen Dinge getan werden. [SS20]
Die enge Zusammenarbeit, sowohl innerhalb des Teams als auch mit Kundinnen und Kunden ist hier stark ausgeprägt. Während sich z. B. das V-Modell stärker auf die Übergabe von Dokumenten verlässt, steht hier ein häufiger Kontakt im Fokus [Sh19]. Im wesentlichen gibt es nur drei Artefakte bei Scrum, *Product Backlog*, *Sprint Backlog* und *Increment* [SS20, S. 10–12].

2.5 Vergleich der vier Vorgehensmodelle

Bei einem Vergleich der vier Modelle lässt sich eine Einteilung in zwei Dimensionen vornehmen. Wie Abbildung 4 zeigt kann ein Modell einerseits als formell oder informell eingeordnet werden, andererseits kann in sequentielle und evolutionäre Modelle unterschieden werden. Sowohl das Wasserfallmodell als auch das darauf aufbauende V-Modell sind stark formell aufgebaut und stützen sich auf eine extensive Dokumentation, die bei der sequentiellen Abarbeitung des Entstehungsprozesses den Rahmen bildet. Die Zusammenarbeit findet hier primär über diese Artefakte statt, z. B. auch beim Übergang von Designphase zu Implementierungsphase.
Bei Scrum stehen deutlich weniger formelle Artefakte im Vordergrund, sondern die direkte Zusammenarbeit verschiedener Spezialistinnen und Spezialisten. Auch bei der Zusammenarbeit mit Kundinnen oder Kunden ist diese stärker ausgeprägt und es werden im Sinne einer evolutionären Entwicklung auch unvollständige Produktstände zum Testen an diese weitergegeben. RUP stellt einen Mittelweg dar, da zwar ebenfalls evolutionär vorgegangen wird, aber trotzdem noch ein stärkerer formeller Fokus als bei Scrum im Vordergrund steht.

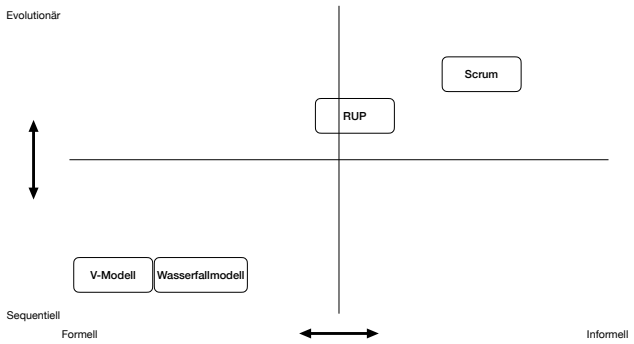

Abb. 4: Einordnung von Vorgehensmodellen. Angepasst nach [Sh19]

Ein weiterer wichtiger Unterschied zwischen den vier Vorgehensmodellen besteht darin, in welchen Bereichen das jeweilige Modell am geeignetsten ist. Das V-Modell und Wasserfallmodell eignen sich durch ihren Schwerpunkt auf eine präzise, gut dokumentierte Vorgehensweise z. B. für Anwendungsbereiche, in denen ebenfalls ein präzises Vorgehen unabdingbar ist. Shiklo [Sh19] nennt an dieser Stelle Softwareunternehmen im Bereich der Medizin oder dem Luftverkehr sowie staatliche Projekte, da hier vorhersehbare Projektpläne und Budgets mit einer strengen Kontrolle erforderlich sind.
Evolutionäre Vorgehensmodelle, wie etwa RUP können optimal für größere Projekte eingesetzt werden, bieten im Vergleich zu den sequentiellen Modellen aber bessere Möglichkeiten bei Anforderungsänderungen im Projektverlauf. Dabei ist RUP erst ab einer gewissen Projektgröße sinnvoll einzusetzen, da eine Vielzahl an Rollen erfüllt werden müssen. Scholz [Sc05, S. 150] nennt eine Teamgröße mit mehr als zehn Personen, mehr als 30 Rollen und über 100 verschiedene Artefakttypen.

Im Gegensatz dazu ist Scrum, als rein agiles Modell, vor allem für kleine Teams geeignet. Dies wird unterstützt durch eine Untersuchung, dass kleinere Teams bessere Leistung erbringen [HV70]. Das kann z. B. bei Start-Up-Unternehmen der Fall sein, bei denen die Kundeneinbeziehung auch stärker im Vordergrund steht. Aber auch größere Projekte können mithilfe von Scrum bewältigt werden, wenn diese in kleinere Teilprojekte aufgeteilt werden können. Hier hat sich der Begriff des *Scrum-of-Scrums* entwickelt, wobei die einzelnen Scrum-Teams einen Vertreter wählen, der wiederum Teil der übergeordneten Scrum-Teams ist [Da19, S. 227].
Allen genannten Modellen gemein ist, dass sie in Bezug auf die konkrete Zusammenarbeit relativ abstrakt bleiben. Welche Strategien und Werkzeuge können z. B. in der Phase der Implementierung helfen, Probleme zu vermeiden? Darauf geht das folgende Kapitel näher ein.

3 Zusammenarbeit während der Implementierung

Nach der Betrachtung der Zusammenarbeit in verschiedenen Vorgehensmodellen soll vertieft auf die gemeinsame Arbeit während der Implementierung eingegangen werden. Die Implementierung ist hier die Phase, in der auf Grundlage des Designs die nutzbare Software entsteht. Diese Phase kann streng zwischen der Designphase und der Testphase liegen, oder wie in agilen Vorgehensmodellen iterativ eingebettet sein. Unabhängig davon können im wesentlichen drei Bereiche der Zusammenarbeit genannt werden. Als erstes ist das die Übergabe von der Designphase zur Implementierung. Hauptsächlich ist dann die gemeinsame Arbeit an einer Code-Basis während der Implementierung betroffen und abschließend muss eine Übergabe an die Qualitätssicherung erfolgen. Die Ausgestaltung der gemeinsamen Arbeit am Quellcode wird näher betrachtet. Dabei geht es darum Strategien kennenzulernen und den Einsatz von Werkzeugen zu bewerten.
Während der Bearbeitung von Quellcode treten zwei Herausforderungen auf. Zum einen besteht der Wunsch auf ältere Versionen zurückfallen zu können, falls ein Experiment nicht funktioniert. Zum anderen soll es möglich sein, dass mehrere Entwicklerinnen und Entwickler parallel an der gleichen Datei arbeiten können. Naive Lösungen für diese Probleme sind leicht verständlich aber aufwändig. Vor einem Experiment wird eine Sicherheitskopie angelegt. Falls das Experiment scheitert, wird die aktuelle Datei verworfen und wieder durch die Sicherheitskopie ersetzt. Für die gleichzeitige Arbeit an Dateien kann eine Kennzeichnung der Änderungen im Code erfolgen, anschließend ein Austausch der Dateien. Eine der Entwicklerinnen oder einer der Entwickler hat nun die Aufgabe, die Dateien zu vergleichen und die Änderungen in einer resultierenden Datei zusammenzufassen. Dass dieses Vorgehen zwar funktioniert, aber nicht sehr effizient ist, ist offensichtlich. Um diese Arbeitsschritte zu automatisieren haben sich Systeme für die Quellcode-Verwaltung entwickelt, die beide Herausforderungen erleichtern. Im Folgenden wird ein kurzer Überblick über diese Systeme gegeben bevor Strategien für die Zusammenarbeit damit vorgestellt werden.

3.1 Quellcode-Verwaltungs-Systeme

Die Entwicklung von Software für die Quellcode-Verwaltung findet bereits seit den Siebziger Jahren statt und hatte mit *IEUPDAT* von IBM [Co22; IB22] ein erstes rudimentäres Werkzeug. Weitere Systeme waren das *Source Code Control System (SCCS)* [Mc21] oder das *Revision Control System (RCS)* [GN22] und mit *Concurrent Versions System (CVS)* auch ein System, bei dem mehrere Entwicklerinnen und Entwickler gleichzeitig an einer Datei arbeiten können [Mc21]. Aktuellere Systeme sind z. B. *Git* [CL22], *Subversion* [Th22], *Mercurial* [Me22] oder *Team Foundation Version Control (TFVC)* [Mi22a]. Diese Systeme können nach ihrer grundlegenden Funktionsweise unterschieden werden:

- Lokale Versionskontrolle: Lokale Versionierung, oft von einer einzelnen Datei. Z. B. SCCS, RCS
- Zentrale Versionskontrolle: Hier ist auch die Kommunikation über das Netzwerk möglich. Wichtig ist, dass der aktuelle Stand des Quellcodes immer auf dem zentralen Server ist. Bei der Bestätigung einer Änderung wird diese auf das zentrale Repository übertragen. Z. B. CVS, Subversion, TFVC
- Verteilte Versionskontrolle: Auch hier ist die Netzwerkkommunikation möglich. Oft existiert auch hier ein zentrales Repository, dieses dient aber nur dem gemeinsamen Austausch. Jede Entwicklerin und jeder Entwickler hat ein voll funktionsfähiges lokales Repository. Z. B. Git, Mercurial

Die Plattform *Openhub* [Sy22] veröffentlicht die Verteilung auf Quellcode-Verwaltungs-Systeme von bei ihr gehosteten Projekten. Im November 2022 hatte Git einen Anteil von 73 Prozent, gefolgt von Subversion mit 22 Prozent sowie Mercurial und CVS mit jeweils einem Prozent. Andere Systeme sind hier nicht relevant. Anzumerken ist, dass hier ausschließlich Open-Source-Projekte in die Statistik fallen, weshalb davon auszugehen ist, dass auch weitere Systeme im Unternehmenskontext relevant sind. Deutlich sichtbar ist aber die Bedeutung von *Git*, weshalb sich diese Arbeit in der Folge auf dieses System konzentriert.

3.2 Quellcode-Hosting-Lösungen

Es existieren verschiedene Plattformen, die die gemeinsame Bearbeitung von Quellcode und das Hosting von Projekten ermöglichen. Plattformen, die auf Git aufbauen sind z. B. *Github*, *BitBucket* (das ursprünglich rein auf *Mercurial* fokussiert war) und *Gitlab*. Im Gegensatz zur reinen Versionsverwaltung, auf der diese Plattformen aufbauen steht die Zusammenarbeit und das Präsentieren des Projekts im Vordergrund.

3.3 Strategien zur Quellcode-Verwaltung

Die genannten Quellcode-Verwaltungs-Systeme helfen bei der gemeinsamen Bearbeitung des Quellcodes, die als stetes Aufspalten und Zusammenführen von Quellcode gesehen werden kann. Im Folgenden werden verschiedene Strategien und Workflows vorgestellt, um diesen Vorgang zu gestalten. Diese Strategien lassen sich grundlegend auch mit anderen Quellcode-Verwaltungs-Systemen anwenden, für die Beispiele wird jedoch Git verwendet. Es wird herausgestellt, welche Vor- und Nachteile die Systeme haben und welche Herausforderungen sie haben. Einige Begriffe sollten eingeführt werden, um Unschärfen zu vermeiden. Das sind die folgenden:

- Repository: Der Ort, in den Regel ein Verzeichnis, an dem die Quellcode-Dateien sowie Metadaten und Informationen für die Quellcode-Verwaltung gespeichert sind. Oft gibt es ein *zentrales Repository*, das über das Netzwerk erreichbar ist und zum Austausch im Team dient.
- Klonen: Beim Klonen eines, meist zentralen, Repositorys wird eine lokale Kopie erstellt, auf der gearbeitet werden kann. Die Bearbeitungen können später wieder mit dem zentralen Repository vereinigt werden.
- Commit: Die *Bestätigung*, dass die Arbeit an Quellcode-Dateien gesichert werden soll. Mit einem Commit wird der aktuelle Stand in die Quellcode-Verwaltung aufgenommen und kann später wieder abgerufen werden oder an ein *zentrales Repository* geschickt werden.
- Branch: Eine Folge von *Commits*. Für einen Branch sind mehrere Betrachtungsweisen möglich. Als Beispiel ist ein zentrales Repository gegeben, welches genau einen Branch enthält. Beim Klonen des Repositorys exisitert dieser Branch dann zwei mal. Bei einem Commit auf dem lokalen Branch ändert sich der Branch auf dem zentralen Repository erstmal nicht. Dadurch ergeben sich zwei unterschiedliche Branches, die jedoch den gleichen Namen besitzen können, wobei sich der Speicherort unterscheidet. Das wäre eine Betrachtung von Branches. Im Fall von Git wird auch das dedizierte Erstellen von Branches angeboten, die neben dem *Hauptbranch* existieren und beim Klonen mit berücksichtigt werden. Diese können sowohl im zentralen als auch im lokalen Repository angelegt werden. Das wäre die zweite Betrachtung von Branches.
- Mergen: Wenn mehrere Branches existieren besteht in der Regel die Anforderung, diese zu einem späteren Zeitpunkt wieder zu vereinigen. Dieser Vorgang wird als Mergen bezeichnet. Es bedeutet, dass die Commits aus einem Branch in einen anderen Branch übernommen werden. Dabei kann es zu Konflikten kommen, wenn eine Datei in beiden Branches bearbeitet wurde.
- Integration: Das Mergen von Änderungen in den *Hauptbranch* des zentralen Repositorys.

Die Betrachtung der folgenden Strategien beruht auf der Annahme, dass jede Version der Software auf der vorangegangenen basiert. Das bedeutet nicht, dass nicht mehrere Versionen gleichzeitig unterstützt und sogar noch mit Hotfixes versehen werden können. Aber wenn an einer Software kundenspezifische Anpassungen vorgenommen wurden, dann dürfen die jeweiligen Anpassungen nur in den Versionen des jeweiligen Kunden oder der Kundin auftauchen. In diesem Fall ist eine komplexere Strategie und mehrere parallele Entwicklungszweige meist notwendig.
Wie Fowler [Fo20] schreibt, ist nicht das Branchen ein Problem, sondern das spätere Mergen. Deshalb soll zuerst auf die Probleme hierbei eingegangen werden, bevor Strategien zum Branchen vorgestellt werden.

3.3.1 Merging

Im optimalen Fall ist das Design einer Software so modular und leicht erweiterbar, dass eine Quellcode-Datei nie von mehreren gleichzeitig bearbeitet wird. Dann würde es beim Mergen keine Probleme geben. In der Realität ist es aber oft der Fall, bzw. teilweise notwendig, dass eine Datei parallel bearbeitet wird. In diesem Fall ist eine gute Strategie für das Mergen notwendig, um die auftretenden Konflikte zu minimieren bzw. eine Auflösung zu erleichtern.
Im Falle, dass es parallele Bearbeitungen einer Datei gibt, lassen sich im wesentlichen zwei Konfliktypen unterscheiden, *textuelle* und *semantische* Konflikte [Fo20]. Ein textueller Konflikt tritt dabei auf, wenn z. B. der Name einer Klasse von zwei Entwicklerinnen oder Entwicklern jeweils unterschiedlich geändert wird. Ein semantischer Konflikt tritt auf, wenn Entwicklerin A den Name der Klasse ändert, Entwickler B parallel diese Klasse mit ihrem alten Namen verwendet. Während ein textueller Konflikt vom Quellcode-Verwaltungs-System leicht erkannt wird und von den Entwicklerinnen und Entwicklern gelöst werden kann, wird ein semantischer Konflikt nicht erkannt und fällt erst zur Kompilierzeit oder im schlimmsten Fall zur Laufzeit auf. Neben dem technischen Mergen des Quellcodes sind also auch hier eine Strategie und Werkzeuge nötig, um semantische Konflikte zu erkennen, bevor eine Integration stattfindet. Eine Möglichkeit sind z. B. *Unit Tests*, die nach dem Mergen der Änderungen auf dem lokalen Repository durchgeführt werden bevor zum zentralen Repository integriert wird. Dadurch wird dafür gesorgt, dass alle Branches auf dem zentralen Repository immer in einem stabilen Zustand sind.
Weitere Herausforderungen ergeben sich aus der Häufigkeit des Mergens, bzw. des Integrierens. Abbildung 5 zeigt zwei unterschiedliche Möglichkeiten der Visualisierung von Branches. Dabei wird veranschaulicht, dass mit fortschreitender Zeit Branches immer weiter divergieren, anstatt parallel nebeneinander zu laufen. Durch häufiges Integrieren, z. B. mithilfe der Continuous Integration, lässt sich diese Divergenz begrenzen, wodurch das Mergen weniger Konfliktpotential besitzt. Nicht in allen Teams ist das allerdings möglich, da hierfür eine gute Testumgebung notwendig ist [Fo20], insbesondere die Open-Source-Community unterscheidet sich hier deutlich vom Unternehmenskontext.

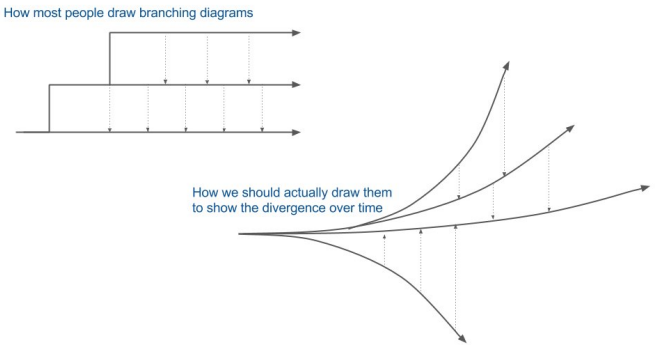

Abb. 5: Möglichkeiten für die Visualisierung von Branching-Diagrammen. Übernommen von [Le17]

3.3.2 Ein-Branch-Strategie

Der einfachste Workflow besteht darin, dass nur ein Branch genutzt wird. In Git wird dieser inzwischen standardmäßig *main* genannt. Wenn Entwicklerinnen oder Entwickler an einem Feature arbeiten möchten, klonen sie das zentrale Repository und arbeiten auf ihrem lokalen Repository auf dem *main*-Branch. Hier ist also die erste, oben genannte Betrachtungsweise, eines Branches relevant. Wenn das Feature fertig ist, können die Änderungen zurück zum zentralen Repository gepusht werden. Falls dort bereits andere Änderungen committed wurden, muss zuvor ein Merge durchgeführt werden.
Dieser Workflow ist lediglich für wenig komplexe Projekte geeignet, wie z. B. die Erstellung dieser Arbeit, bei der wenige Entwickler beteiligt sind. Ein Nachteil ist, dass ein Commit, der an das zentrale Repository gepusht wird, potentiell für einen instabilen Zustand sorgt. Insbesondere, wenn an einem Feature gearbeitet wird, das länger dauert und aus mehreren Commits besteht, kann es notwendig sein, dass ein Austausch über die bisherigen Commits stattfindet. Der einzige Weg, der hier möglich ist, ist der main-Branch auf dem zentralen Repository. Wenn das unfertige Feature nicht sauber vom Bestandscode getrennt ist, kann es zu Problemen kommen. Der Vorteil ist, dass der Workflow sehr leicht zu verstehen ist. Bei Git wird diese Strategie zum Teil auch als *Basic Workflow* bezeichnet [An21]. Die genannten Probleme könnten vermieden werden, wenn dieser Workflow in Kombination mit einer Forking-Strategie (siehe auch Abschnitt 3.3.4) genutzt wird.
Alternativ ist ein Vorgehen nach dem Continuous Integration möglich. Zentraler Punkt hierfür ist das Integrieren sobald ein stabiler Zustand erreicht ist. Dabei muss auch bei länger dauernden Features gewährleistet werden, dass diese abgekoppelt sind, indem z. B. das Interface als letztes implementiert wird [Fo20] und tatsächlich stabile Zustände erreicht werden. Es findet hier also eine Verlagerung der Arbeit statt. Anstelle aufwändige und fehleranfällige große Merge-Vorgänge durchzuführen, wird der Fokus darauf gelegt, mehr Aufwand in die Erreichung von stabilen Zuständen zu stecken und dafür kleinere, einfachere Merge-Vorgänge zu haben.

3.3.3 Multi-Branch-Strategie

Eine weitere Möglichkeit, um die Nachteile der Ein-Branch-Strategie zu vermeiden, ist die Nutzung von weiteren Branches neben dem Hauptbranch. Durch die Nutzung von eigenen Branches, im Sinne der zweiten, oben genannten Betrachtungsweise, für die Entwicklung von Features, wird der *main*-Branch in einem stabilen Zustand gehalten und trotzdem können auf dem Feature-Branch Commits vorgenommen und auch ausgetauscht werden.
Die Kernidee besteht darin, dass einzelne Features getrennt voneinander entwickelt werden, und zwar bis sie komplett fertig sind. Da die Feature-Branches aber auch auf dem zentralen Repository gespeichert werden können, ist trotzdem ein Austausch der Änderungen mit anderen Entwicklerinnen und Entwicklern möglich ohne den *main*-Branch in einen instabilen Zustand zu bringen. Nicht zu unterschätzen ist trotzdem das Problem, die einzelnen Branches wieder in den *main*-Branch zu mergen. Denn auch wenn die einzelnen Features inhaltlich unabhängig sind, kann es vorkommen, dass an den gleichen Stellen Änderungen im Bestandscode notwendig sind.
Ein konkretes Beispiel für diese auch *Feature-Branching* gennannte Strategie ist *git-flow* [Dr10]. Folgende Branches existieren in diesem Workflow:

- Main (oder Master): Dieser spiegelt zu jeder Zeit einen Stand wider, der ausgeliefert werden kann. Zusammen mit Develop bilden sie die Haupt-Branches.
- Develop: Dieser spiegelt den aktuellen Stand der Entwicklung wider. Von diesem Branch gehen Feature- und Release-Branches ab und hierhin wird integriert.
- Feature Branches: Werden von Develop abgespalten und dorthin integriert. Sind für die dedizierte Entwicklung von Features vorgesehen.
- Release Branches: Werden von Develop abgespalten, um unabhängig von weiteren Integrationen nach Develop letzte Fehler zu korrigieren und einen stabilen Zustand herzustellen, der nach Main integriert werden kann. Wird auch wieder zurück nach Develop integriert.
- Hotfix Branches: Werden von Main abgespalten, um einen schlimmen Fehler einer veröffentlichten Version zu beheben. Wird wieder nach Main und Develop integriert.

3.3.4 Weitere Strategien

Neben den ausführlich beschriebenen Strategien, die zu den am häufigsten beschriebenen gehören, gibt es weitere, die je nach Anwendungsumfeld eine Alternative darstellen können. Dabei entstehen beständig neue Vorschläge oder Varianten, um sich an die neuen Entwicklungsumgebungen und Anforderungen anzupassen. Dazu gehören z. B. der *Forking Workflow* [An21], *OneFlow-Workflow* [Ru17] oder *Github Flow* [Ch11].
Der *Forking-Workflow* hat sich insbesondere in der Open-Source-Community und mit der Verbreitung von Plattformen wie GitHub oder GitLab durchgesetzt, wo zum Teil viele

Entwicklerinnen und Entwickler mit unterschiedlichen Arbeitsweisen zusammenarbeiten. Anstelle eines einzigen zentralen Repositorys, zu dem alle ihre Änderungen pushen besitzt jeder ein eigenes zentrales Repository (ein sogenannter *Fork*) auf dem gearbeitet wird. Dadurch können lokale Änderungen veröffentlicht und geteilt werden, um beispielsweise Feedback zu erhalten, ohne das originale Repository in einen instabilen Zustand zu bringen. Der Vorteil besteht darin, dass die Besitzerin oder der Besitzer des originalen Repositorys keinen weitreichenden Zugriff auf das Repository an viele Entwicklerinnen und Entwickler gewähren muss. Über Pull-Requests kann jedoch eine Anfrage an das originale Repository gestellt werden, die Änderungen auch dort zu integrieren. Dieser Workflow kann je zentralem Repository mit anderen Workflows kombiniert werden, sodass an einem Fork auch mehrere arbeiten können.
OneFlow ist ebenso wie *Github Flow* eine Reaktion auf den *GitFlow-Workflow*, der für moderne Web-Entwicklung nicht mehr optimal war und als zu komplex und fehleranfällig betrachtet wurde [Ru15][Ch11]. Die Kernidee ist hierbei, möglichst wenige langlebige Branches zu haben, um die Versionsgeschichte möglichst klar zu lassen. Ein weiterer Vorteil weniger Branches ist, dass es weniger Verwirrung über die richtige Verwendung der Branches und weniger Probleme beim Mergen gibt. Dies wird erreicht, indem sehr häufig integriert wird und der main-Branch immer in einem stabilen Zustand gehalten wird, z. B. mithilfe von selbsttestendem Code. Insbesondere bei der Webentwicklung und Continuous Delivery/Deployment vereinfacht das die Arbeit unter der Voraussetzung.

3.4 Grenzen von Git

Bei der Wahl einer Strategie sollte auch berücksichtigt werden, welche Artefakte während der Implementierung anfallen. Insbesondere Binärdateien können dabei problematisch sein. Binärdateien können zum Beispiel 3D-Modell-Daten sein [Bl22]. Auch hier fallen oft lediglich kleine Anpassungen an, sodass es unnötig wäre, während des Zusammenführens die komplette Datei zu kopieren. Dies ist aber notwendig, da Git bei Binärdateien nicht nur die Änderungen speichern kann. Dadurch erhöht sich der Speicherbedarf des Repositorys stark und auch die Performance kann darunter leiden, wenn jedes Mal viele Daten übertragen werden müssen. Hier muss eine Einschätzung getroffen werden, wie häufig Binärdateien geändert werden und wie groß diese sind. Wenn nur wenige, kleine Binärdateien im Repository vorhanden sind, die zudem selten geändert werden, zum Beispiel finale Dokumentations-PDFs, kann Git ohne Bedenken genutzt werden. Im Falle vieler, potentiell großer Binärdateien, wie zum Beispiel bei der Entwicklung von 3D-Computerspielen stellt dies einen Einsatz von Git in Frage. Zur Lösung dieses Problems gibt es Ansätze als Erweiterungen von Git [Ke16]. Der Kerngedanke ist dabei, dass nicht die Binärdateien selbst im Repository gespeichert sind, sondern Zeiger auf diese Dateien [Ge19]. Dadurch kann der Speicherbedarf und die Performance von Git optimiert werden. Die Versionierung und parallele Bearbeitung der Binärdateien muss dann aber durch das entsprechende Bearbeitungsprogramm sichergestellt werden können.

3.5 Zwischenfazit

Die vorgestellten Strategien bieten eine Grundlage für die Entscheidungsfindung bei der Wahl von Richtlinien und Werkzeugen während der Zusammenarbeit bei der Implementierung. Nicht vergessen werden sollte, dass diese Strategien nicht für sich stehen und auch keinen Anspruch auf Gültigkeit erheben. Sie müssen immer im Kontext der gesamten Entwicklungsumgebung stehen und natürlich gegebenenfalls an die konkrete Situation im Projektteam angepasst werden. In diesem Zusammenhang ist auch eine Abstimmung mit der Strategie des Qualitätsmanagements notwendig, auf das im Folgenden näher eingegangen wird. Eine Untersuchung zur Softwarequalität in Abhängigkeit der Branching-Strategie kam dabei zu dem Ergebnis, dass zu viele Branches einen negativen Einfluss haben und die Branching-Strategie mit der Software-Architektur und der Team-Struktur abgestimmt sein sollte, um negative Konsequenzen zu vermeiden [SBZ12].

4 Qualitätsmanagement und Qualitätssicherung

Die Software ist ein großer Teil unseres Lebens geworden und ist Bestandteil vieler Produkte, die wir nutzen: von den Handys bis zu den Autos. Wir erwarten, dass die Programme korrekt und fehlerlos funktionieren, ohne dabei nachzudenken, welcher Verwaltung- und Sicherungsaufwand dahintersteckt. Wenn diese Anforderungen erfüllt werden, spricht man von „guter Qualität“. [Sc12, S. 1, 2] Was bedeutet überhaupt „gute Qualität“ und wie lässt sich diese messen? Welche Maßnahmen sind dabei notwendig, um die hohe Qualität zu leisten? Welche Ansätze zur Qualitätssicherung sind für welche Projektgrößen und Vorgehensmodellen anwendbar und besonders vorteilhaft? In diesem Kapitel wird versucht, diese Fragen zu beantworten.

4.1 Definition und Eingrenzung

Es gibt unterschiedliche Wege, die Softwarequalität zu bewerten. In der Softwareentwicklung wird häufig der Standard ISO/IEC 9126 zur Bewertung der Softwarequalität verwendet. Diese Norm definiert den Begriff Software-Qualität als:

> „Gesamtheit der Merkmale und Merkmalswerte eines Software-Produkts, die sich auf dessen Eignung beziehen, festgelegte Erfordernisse zu erfüllen.“ [Ho08, S. 6]

Konkret bedeutet das, dass die Software-Qualität sich aus mehreren verschiedenen Kriterien zusammensetzt. Der Standard definiert dabei 6 Qualitätsmerkmale: Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz, Wartbarkeit und Portierbarkeit. Dabei beeinflussen sich die Merkmale gegenseitig: positiv und negativ. Zum Beispiel, eine Verbesserung der Wartbarkeit

ergibt eine Verbesserung der Funktionalität, aber eine Verschlechterung der Effizienz. Somit lässt sich sagen, dass eine Verbesserung aller Qualitätsmerkmale gleichzeitig nicht möglich und ein Kompromiss zu suchen ist [Ho08, S. 11][Go11, S. 132]. Um die Software-Qualität optimal zu verbessern, werden Maßnahmen zur Qualitätssicherung gebraucht. Diese lassen sich in die Produktqualität- und Prozessqualitätsmaßnahmen unterteilen.
Bei der Produktqualität steht das Softwareprodukt im Vordergrund. Die Maßnahmen dieser Gruppe werden in konstruktive Maßnahmen (Fehlervermeidung während der Entwicklung) und analytische Maßnahmen (Fehlerfindung nach der Entwicklung) unterteilt.
Die Prozessqualität umfasst die in Kapitel 2 beschriebenen Vorgehensmodellen und andere Verwaltungstechnologien. Für diese Arbeit wird der Bereich der Prozessqualität. Es werden lediglich die Auswirkungen und Vorteile auf die Projekte in unterschiedlichen Vorgehensmodellen beim Einsatz der verschiedenen Produktqualitätsmaßnahmen beschrieben.

4.2 Konstruktive Qualitätssicherung

Zu dem Gebiet der konstruktive Qualitätssicherung fallen insbesondere Maßnahmen, die eine frühzeitige Erkennung und Vermeidung der Fehler während des Entwicklungsprozesses ermöglichen. In diesem Kapitel werden 3 wichtigsten davon: **Softwarerichtlinien**, **Vertragsbasierte** Programmierung und **Dokumentation** vorgestellt.
Softwarerichtlinien bezeichnen alle Mittel, die den Programmierprozess über die syntaktischen und semantischen Regeln der jeweiligen Programmiersprache hinweg regelt [Ho08, S. 65]. Die wichtigsten Ziele dieser Maßnahmen sind *Vereinheitlichung* und *Fehlerreduktion*. Um diese Ziele zu erreichen, werden *Notations-* und *Sprachkonventionen* eingesetzt.

Die meisten Software-Ingenieure entwickeln im Laufe der Jahre ihren eigenen Programmier-Stil, worauf sich die Person gewohnt. In Teams kann diese Tatsache aber dazu führen, dass mehrere Mitarbeiter den Code von anderen nicht richtig verstehen oder mehr Zeit dafür brauchen. Um dieses Problem zu lösen, werden **Notationskonventionen** verwendet. Diese umfassen die Namensgebung der verwendeten Strukturen und Variablen, die Verwendung von Leerzeilen und Charakteristik und die Dokumentation des geschriebenen Codes. Oft gibt es unternehmensspezifische Konventionen, dennoch haben sich folgende Notationsvorschriften in der Software-entwicklung durchgesetzt:

- **Pascal Case**: die Variablennamen beginnen mit dem Großbuchstaben und jedes weitere Wort wird ohne Trennzeichen mit großen Buchstaben eingefügt (z.B. ForegroundColor).
- **Camel Case**: gleich wie Pascal Case, aber das erste Wort wird mit Kleinbuchstaben angefangen (foregroundColor).
- **Uppercase/Lowercase**: der gesamte Name wird entweder in Groß- oder Kleinbuchstaben geschrieben (FOREGROUNDCOLOR).

Oft werden verschiedene Notationsstile miteinander kombiniert, damit verschiedene Arten von Bezeichner dargestellt werden. Dies sorgt für ein besser lesbares und verständliches Code.
Um ein weiteres Ziel der Softwarerichtlinien, die Fehlerreduktion, zu ermöglichen, werden die **Sprachkonventionen** verwendet. Im Gegensatz zu den Notationskonventionen, werden hier die Entwickler darin eingeschränkt, bestimmte Sprach-Konstrukte unter bestimmten Voraussetzungen zu verwenden. Dazu gehören zum Beispiel, mit welchen Variablentypen auf die einzelnen Bits zugegriffen werden darf (unsigned int in MISRA C Standard) [Ho08, S. 76] [Mi22b].

Einen weiteren Bestandteil der konstruktiven Qualitätssicherung bildet die **Vertragsbasierte** Programmierung. Das Konzept wurde von Bertrand Meyer unter dem Namen Design by contract eingeführt [Me92]. Kurz beschrieben, basiert die vertragsbasierte Programmierung auf den folgenden Prinzipien:

- **Vor- und Nachbedingungen**: die Objekte müssen bestimmte Voraussetzungen erfüllen, um Routinen betreten und verlassen zu können. Dafür werden bestimmte „require“ Blöcke in das Programm eingefügt.
- **Invarianten**: im Programm werden globale Beschränkungen der Wertebereiche bestimmter Variablen und Strukturen gesetzt.
- **Zusicherungen**: bezeichnen die Überprüfungen, die im Gegensatz zu Invarianten nur an der Stelle des Auftretens ausgewertet werden.

All diese Prinzipien werden auch heute bei der Anforderungsspezifikation und für die Definition von Tests eingesetzt [Ho08, S. 96] [Me92]

Verwaltung der **Projektdokumentation** ist auch eine wichtige Maßnahme zur Herstellung der Qualitätssicherheit. Dabei lässt sich diese in die externen und internen Dokumente aufteilen. Die externe Dokumentation dient dem Austausch mit dem Kunden. Diese hat oft, insbesondere in großen Projekten klare Vorgaben und Muster wie sie zu erstellen ist (z.B. für Pflichtenheft) [Ho08, S. 141]. Bei der internen Dokumentation handelt es sich um andere Dokumente, die nur unternehmensintern zur Verfügung stehen. Diese Dokumente umfassen zum einen die oben beschriebene Software-Richtlinien und auch die Programmdokumentation und andere Spezifikationen.
Einer der wichtigsten Dokumente in vielen Vorgehensmodellen ist die Spezifikation. In dem Dokument werden die Anforderungen an ein Software-System genauer beschrieben. Diese Spezifikation kommt daher oft als externes Dokument und wird bei vielen Projekten formal gehalten. Die Anwender tauschen dann während der Entwicklung die sogenannten Implementierungsdokumente aus, wo anhand von Kommentaren im Code die implementierten Funktionen beschrieben werden. Die Formulierung dieser Dokumente kann großen

Aufwand, vor allem in dokumenten-basierten Vorgehensmodellen beanspruchen, weswegen auch viele dieser Modelle, wie z.B. das V-Modell kritisiert wird. [AA15c]

4.3 Analytische Qualitätssicherung

Bei der analytischen Qualitätssicherung wird, im Gegensatz zur konstruktiven, geht es nicht um Fehlervermeidung, sondern um Fehlerfindung und Beseitigung nach dem Entwicklungsprozess [Ho08, S. 20] [Go11, S. 132]. Dies wird mithilfe verschiedener Test- und Verifikationsverfahren nach und während der Entwicklung ermöglicht. Diese werden im Folgenden vorgestellt.

4.3.1 Software-Tests

Bereits im Jahr 1979 war in der Software-Entwicklung bekannt, dass ungefähr die Hälfte der Zeit- und Geldkosten in das Testen investiert werden muss [My12]. Diese Tatsache hat sich bis heute nicht wesentlich verändert, wenn überhaupt, nimmt die Implementierung und Durchführung der Tests in manchen Projekten einen wesentlich höheren Anteil an Aufwand. Diese Tests lassen sich bezüglich verschiedener Merkmale in verschiedene Klassen einteilen [Ho08, S. 158]:

- **Prüfebene**: In welcher Projektphase wird der Test durchgeführt?
- **Prüfkriterium**: Welche inhaltlichen Aspekte werden getestet?
- **Prüfmethodik**: Wie werden die Tests konstruiert?

In dieser Arbeit werden verschiedene Testarten anhand ihres Prüfkriteriums vorgestellt. Diese Tests werden anhand ihres Inhaltes unterschieden und können dabei in 3 Kategorien unterteilt werden: Funktionale, Operationale und Temporale Tests. Die wichtigsten Arten von **funktionalen** Tests sind [Ho08, S. 170] [Me18, S. 242]:

- **Funktionstests**: sind die am häufigsten verwendete Tests, bei diesen Tests wird überprüft, ob ein richtiges Ergebnis geliefert wurde, also ob die Applikation korrekt läuft.
- **Crashtests**: es wird versucht, ein System zum Abstürzen zu kriegen. Es findet eine gezielte Suche nach Schwachstellen statt, durch die Bekämpfung dieser, können sicherheitskritische Systeme geschützt werden.
- **Zufallstests**: es wird nicht mit vorgegebenen, gezielt erzeugten Eingabedaten getestet werden, sondern mit zufälligen. Da diese Methode aber schwer systematisch anzuwenden ist, kann sie nur als Ergänzung verwendet werden.

Die **operationalen** Tests beinhalten im Großen und Ganzen die Installation, Sicherheit und Bedienbarkeit von Systemen. Diese Tests sind insbesondere in der Web-Branche und anderen modernen Applikation wichtig, da die Bedienbarkeit und User-Experience häufig im Vordergrund stehen [Ho08, S. 170–174] [Me18, S. 242]. In den **temporalen** Tests werden vor allem die Effizienz und Schnelligkeit von Software getestet. Zum Beispiel wird es bei *Last-* und *Stresstesten* geprüft, wie sich ein System an den oder über den definierten Grenzen verhält. Die Geschwindigkeit von Software wird in den *Komplexitäts-* und *Laufzeittests* ermittelt, wodurch auch entschieden werden kann, ob die implementierten Algorithmen den Anforderungen entsprechen.

4.3.2 Statische Analyze und Verifikation

Durch die **statische Code-Analyze** umfasst alle Maßnahmen, bei denen der Quellcode ohne Programmausführung untersucht wird. Die untersuchenden Bestandteile des Codes werden mithilfe von Software-Metriken beschrieben. Durch diese Analyze wird die Zuverlässigkeit und die Funktionalität des Codes geprüft. Zu den verbreiteten Software-Metriken gehören die LOC (Lines of Code) und NCSS (Non-Commented Source Statements). Bei diesen Software-Metriken wird die Anzahl der geschriebenen Programmzeilen, bzw. ausführbaren Zeilen des Codes gezählt und dadurch wird eine grobe Aussage über die Programmkomplexität gemacht [Ho08, S. 249, 250]. Jedoch sagt dieser Ansatz in der Realität nicht viel über das Programm aus, weil unterschiedliche Programmiersprachen unterschiedliche Anzahlen von Befehlen und dadurch Zeilen haben können und auch den Aspekt der optimalen Nutzung der Werkzeuge betrachtet werden muss. Deswegen werden heutzutage viele andere Metriken benutzt, zum Beispiel wird es untersucht, wie oft eine Funktion überschrieben wird (in objektorientierten Sprachen) oder wie oft die Funktion im Programm aufgerufen/wiederverwendet wird. Bei diesem Ansatz wird schnell über die Rahmen der statischen Analyze gegangen und es wird in einer **Software-Verifikation** automatisch geprüft. Dabei wird der Code ausgeführt und mithilfe von verschiedenen mathematischen Analysen untersucht [Ho08, S. 334, 335].

4.4 Produktqualitätsmaßnahmen in Vorgehensmodellen

Wie bereits oben beschrieben, gibt es viele verschiedene Ansätze und Maßnahmen zur Qualitätssicherung in der Softwareentwicklung. Es gibt dennoch keine Vorgaben, welche dieser Methoden für welche Arten von Projekten angewendet werden können. Es lässt sich aber sowohl aus den Projektgrößen als auch der verwendeten Vorgehensmodellen die passenden Qualitätsmaßnahmen herausleiten. Dabei gibt es allgemeine Maßnahmen, durch die alle Projekte profitieren können und die auch oft eingesetzt werden. Dazu gehören, zum Beispiel, **Softwarerichtlinien** (4.2), weil man mithilfe der Sprach- und Notationskonventionen viele Fehler vermeiden kann, ohne dabei einen großen Managementaufwand einzusetzen.

Die anderen Methoden werden im Folgenden tabellarisch auf verschiedene Vorgehensmodelle und Projektgrößen angewendet (siehe Abbildung 6): Wie man sieht, werden viele

Qualitätsmaßnahme	Passendes Vorgehensmodell	Projektgröße
Vertragsbasierte Programmierung	V-Model, Wasserfallmodel	Mittel bis hoch
Projektdokumentation	Alle, aber insbesondere V-Model und Wasserfallmodel	Je größer das Projekt, desto mehr Dokumentation
Funktionale Software-Tests	Allgemein einsetzbar, aber sehr wichtig in Scrum und RUP	Für alle Projektgrößen
Operationale Tests	Besonders wichtig für Scrum	Für alle Projektgrößen
Temporale Tests	V-Modell und RUP	Große Projekte
Statische Code-Analyze und -Verifikation	V-Modell	Große Projekte

Abb. 6: Anwendung verschiedener Qualitätsmaßnahmen

Maßnahmen im V-Modell und nicht in Scrum verwendet, weil im V-Modell der Fokus auf die Qualitätssicherung und Dokumentation gelegt wird. Im Gegensatz dazu wird bei Scrum schneller auf die Anforderungsänderungen reagiert und es wird schneller ein Ergebnis an den Kunden geliefert.

5 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde die Bedeutung einer gelungenen Zusammenarbeit bei der Erarbeitung großer Software-Projekte hervorgehoben, wobei bereits gerade auch sehr kleine Teams große Probleme bei der Zusammenarbeit haben können. Es wurden vier Vorgehensmodelle in Bezug auf die Zusammenarbeit vorgestellt und verglichen. Als Ergebnis lässt sich hier festhalten, dass es auf die gesamte Entwicklungsumgebung und die Anforderungen ankommt, welches Modell geeignet ist. Aufgrund ihrer Komplexität sind Modelle wie das V-Modell oder RUP eher für größere Teams geeignet, wohingegen Scrum bereits von kleinen Teams leicht umgesetzt werden kann. Insgesamt beschäftigt sich die Forschung noch zu wenig mit konkreten Umsetzungen der Zusammenarbeit in verschiedenen Vorgehensmodellen, gerade auch in Bezug auf Werkzeuge.
Die vorgestellten Modelle und Strategien sind dabei lediglich Vorschläge, die in der Regel angepasst werden sollten. Auch wenn diverse Anbieter, gerade auch von Werkzeugen, versprechen, dass diese Out-of-the-Box funktionieren, ist dies selten der Fall. Adam Ruka [Ru22], Entwickler bei großen Unternehmen wie Amazon und Apple schreibt beispielsweise, dass die großen Tech-Unternehmen oft kein agiles Vorgehensmodell verwenden, das aber auch nicht heißt, dass sie ein Wasserfall-Modell verwenden. Stattdessen haben sie eigene Methoden und Modelle entwickelt, die oft Vorteile aus beiden Welten übernehmen und spezifisch auf die Unternehmenssituation angepasst sind. Für Unternehmen aus dem Mittelstand ist die Entwicklung eines komplett eigenen Vorgehensmodells oder Strategien für die Zusammenarbeit während der Implementierung aufgrund der Personal-Ressourcen meist nicht so leicht möglich. Hier ist dann die Orientierung an den bekannten Modellen und

einer kleineren Anpassung vermutlich der beste Ansatz. Dabei gibt es keine allgemein gültige Lösung. Das richtige Vorgehensmodell sowie die Strategie während der Implementierung hängt von den Anforderungen, der Entwicklungsumgebung und dem Team ab.

Literatur

[AA15a] Adel, A.; Abdullah, B.: What is V-model- advantages, disadvantages and when to use it?, 2015, URL: http://tryqa.com/what-is-v-model-advantages-disadvantages-and-when-to-use-it/, Stand: 18. 11. 2022.

[AA15b] Adel, A.; Abdullah, B.: What is Waterfall model- Examples, advantages, disadvantages & when to use it?, 2015, URL: http://tryqa.com/what-is-waterfall-model-advantages-disadvantages-and-when-to-use-it/, Stand: 18. 11. 2022.

[AA15c] Adel Alshamrani; Abdullah Bahattab: A Comparison Between Three SDLC MODELS, 1. Jan. 2015.

[An21] Antony, A.: 5 Different Git Workflows, 2021, URL: https://medium.com/javarevisited/5-different-git-workflows-50f75d8783a7, Stand: 02. 12. 2022.

[BK08] Bunse, C.; von Knethen, A.: Vorgehensmodelle kompakt. Spektrum Akad. Verl., Heidelberg, 2008, ISBN: 3827419506.

[Bl22] Blender Community: Blender 3.5 Nutzerhandbuch - Assets, Files & Data System, 2022, URL: https://docs.blender.org/manual/de/dev/files/introduction.html, Stand: 02. 12. 2022.

[Ch11] Chacon, S.: GitHub Flow, 2011, URL: http://scottchacon.com/2011/08/31/github-flow.html, Stand: 02. 12. 2022.

[CL22] Chacon, S.; Long, J.: git –local-branching-on-the-cheap, 2022, URL: https://git-scm.com.

[Co22] Computer History Museum: The IEBUPDAT Program, 2022, URL: https://www.computerhistory.org/collections/catalog/102713291.

[Da19] Dalton, J.: Great Big Agile, An OS for Agile Leaders. Springer Science+Business Media, New York, 2019.

[Dr10] Driessen, V.: A successful Git branching model, 2010, URL: https://nvie.com/posts/a-successful-git-branching-model/.

[DW99] Dröschel, W.; Wiemers, M.: Das V-Modell 97, Der Standard für die Entwicklung von IT-Systemen mit Anleitung für den Praxiseinsatz. Dröschel, Wolfgang (Edited by) Wiemers, Manuela (Edited by), De Gruyter Oldenbourg, München, 1999, ISBN: 978-3-486-25086-2.

[Fo20] Fowler, M.: Patterns for Managing Source Code Branches, 2020, URL: https://martinfowler.com/articles/branching-patterns.html.

[Fr09] Friedrich, J.; Hammerschall, U.; Kuhrmann, M.; Sihling, M.: Das V-Modell® XT, Für Projektleiter und QS-Verantwortliche kompakt und übersichtlich. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, ISBN: 9783642014888.

[Ge19] Gehman, C.: How Git LFS Works: Overview of Git Large File Storage, 2019, URL: https://www.perforce.com/blog/vcs/how-git-lfs-works, Stand: 02. 12. 2022.

[GN22] GNU: GNU RCS, 2022, URL: https://www.gnu.org/software/rcs/.

[Go11] Goll, J.: Methoden und Architekturen der Softwaretechnik. Vieweg + Teubner, Wiesbaden, 2011, ISBN: 978-3-8348-1578-1.

[Ho08] Hoffmann, D. W.: Software-Qualität. Hoffmann, Dirk W. (VerfasserIn), Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, ISBN: 9783540763239.

[Hö08] Höhn, R.; Rausch, A.; Broy, M.; Höppner, S.; Petrasch, R.; Biffl, S.; Wagner, R.; Hesse, W.; Bergner, K.: Das V-Modell XT, Grundlagen, Methodik und Anwendungen. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, ISBN: 9783540302506.

[Hu15] Hughes, R.: Agile Data Warehousing for the Enterprise, A Guide for Solutions Architects and Project Leaders. Morgan Kaufmann, Amsterdam, Boston, Heidelberg, London, New York, 2015.

[HV70] Hackman, J. R.; Vidmar, N.: Effects of Size and Task Type on Group Performance and Member Reactions. Sociometry 33/1, S. 37–54, 1970, ISSN: 00380431, URL: http://www.jstor.org/stable/2786271, Stand: 04. 12. 2022.

[IB22] IBM: Quick References for IBM Mainframe Programming, 2022, URL: https://ibmmainframes.com/references/a20.html, Stand: 26. 11. 2022.

[Ke16] Kenlon, S.: Hot to manage binary blobs with Git, 2016, URL: https://opensource.com/life/16/8/how-manage-binary-blobs-git-part-7, Stand: 02. 12. 2022.

[Kr00] Kruchten, P.: The Rational Unified Process–An Introduction. In: the Rational edge - e-zine for the rational community. https://www.researchgate.net/publication/220018149_The_Rational_Unified_Process--An_Introduction, 2000.

[Kr99] Kruchten, P.: Der Rational Unified Process. Eine Einführung. Addison Wesley, München, 1999.

[Ky19] Kyeremeh, K.: Overview of System Development Life Cycle Models. SSRN Electronic Journal/, 2019.

[Le17] LeRoy, J.: How to draw a branching diagram, 2017, URL: https://twitter.com/jahnnie/status/937917022247120898, Stand: 04. 12. 2022.

[LL07] Ludewig, J.; Lichter, H.: Software Engineering, Grundlagen, Menschen, Prozesse, Techniken. dpunkt-Verl., Heidelberg, 2007, ISBN: 9783898642682.

[Mc21] McMillan, T.: A History of Version Control, 2021, URL: https://blog.tarynmcmillan.com/a-history-of-version-control.

[Me18] Meyer, A.: Softwareentwicklung: Ein Kompass für die Praxis. Mode of access: Internet via World Wide Web Text (nur für elektronische Ressourcen), De Gruyter Oldenbourg, Berlin, München und Boston, 2018, ISBN: 3110575809.

[Me22] Mercurial Community: Mercurial - Work easier - Work faster, 2022, URL: https://www.mercurial-scm.org.

[Me92] Meyer, B.: Applying 'design by contract'. Computer 25/10, S. 40–51, 1992, ISSN: 0018-9162.

[Mi22a] Microsoft: What is Team Foundation Version Control, 2022, URL: https://learn.microsoft.com/en-us/azure/devops/repos/tfvc/what-is-tfvc?view=azure-devops.

[Mi22b] Misra Association: MISRA, 2/12/2022, URL: https://www.misra.org.uk/, Stand: 02. 12. 2022.

[My12] Myers, G. J.: The art of software testing. J. Wiley & Sons, Hoboken, N.J., 2012, ISBN: 9781118133156.

[re18] bitkom research: Scrum - König unter den agilen Methoden, 2018, URL: https://www.bitkom-research.de/de/pressemitteilung/scrum-koenig-unter-den-agilen-methoden.

[Ru15] Ruka, A.: GitFlow considered harmful, 2015, URL: https://www.endoflineblog.com/gitflow-considered-harmful.

[Ru17] Ruka, A.: OneFlow - a Git branching model and workflow, 2017, URL: https://www.endoflineblog.com/oneflow-a-git-branching-model-and-workflow, Stand: 26. 11. 2022.

[Ru22] Ruka, A.: Big Tech uses neither Agile nor Waterfall, 2022, URL: https://www.endoflineblog.com/big-tech-uses-neither-agile-nor-waterfall, Stand: 02. 12. 2022.

[SBZ12] Shihab, E.; Bird, C.; Zimmermann, T.: The Effect of Branching Strategies on Software Quality. In: Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM '12, Association for Computing Machinery, Lund, Sweden, S. 301–310, 2012, ISBN: 9781450310567, URL: https://doi.org/10.1145/2372251.2372305.

[Sc05] Scholz, P.: Softwareentwicklung eingebetteter Systeme, Grundlagen, Modellierung, Qualitätssicherung. Springer-Verlag, Berlin, Heidelberg, 2005.

[Sc12] Schneider, K.: Abenteuer Softwarequalität: Grundlagen und Verfahren für Qualitätssicherung und Qualitätsmanagement. 2012.

[Sh19] Shiklo, B.: 8 Vorgehensmodelle der Softwareentwicklung: mit Grafiken erklärt, 2019, URL: https://www.scnsoft.de/blog/vorgehensmodelle-der-softwareentwicklung.

[SS20] Schwaber, K.; Sutherland, J.: The Scrum Guide: The Definitive Guide to Scrum: The Rules of the Game, 2020, URL: https://scrumguides.org.

[Sy22] Synopsys Inc.: Compare Repositories, 2022, URL: https://www.openhub.net/repositories/compare, Stand: 15. 11. 2022.

[Th22] The Apache Software Foundation: Apache Subversion, 2022, URL: https://subversion.apache.org.

[Va14] Vanshika Rastogi: Software Development Life Cycle Models-Comparison , Consequences./, 2014, URL: https://www.semanticscholar.org/paper/Software-Development-Life-Cycle-Models-Comparison-%2C-Rastogi/577cfae86ee8bd01d64783c1c6d240523bea3b03#extracted.

[Ve00] Versteegen, G.: Projektmanagement mit dem Rational Unified Process. Springer-Verlag, Berlin, Heidelberg, 2000.

[Wi22] Wickner, A.: Wie ist die Komplexität von Software-Projekten jetzt und in Zukunft zu bewältigen. Informatik Aktuell/, Aug. 2022, URL: https://www.informatik-aktuell.de/entwicklung/methoden/wie-ist-die-komplexitaet-von-software-projekten-jetzt-und-in-zukunft-zu-bewaeltigen.html.