Klassische vs. agile Methoden der Softwareentwicklung

Vorgetragen am 03. November 2004 durch

Jonathan Weiss
Emel Tan

Erstellt für

SWT
Methoden und Werkzeuge zur Softwareproduktion

Einleitung

Agile und klassische Methoden bilden jeweils das Ende eines Spektrums im Rahmen der Softwareentwicklung. Klassische Methoden konzentrieren sich auf den formalisierten Prozess, während agile Methoden die eigentliche Entwicklung und die Kommunikation zwischen Entwickler und Kunde im Mittelpunkt sehen. In dieser Arbeit werden beide Methoden kurz beleuchtet und abschließend bewertet. Die klassischen Methoden werden anhand des Wasserfall- und des Spiral-Modells verdeutlicht, während im Rahmen der agilen Methoden Extreme Programming (XP) im Vordergrund steht.
Die Bewertung beider Methoden erfolgt durch die Untersuchung ausgewählter Kriterien. Anhand der Ausprägung der Kriterien ermöglicht sich eine Auswahl der zu nutzenden Methodik.

Methoden der Softwareentwicklung

Bis in die späten Sechziger Jahre wurde Softwareentwicklung ohne Planung und Methode betrieben. Dieser als „Code and Fix“ bezeichnete Ansatz erfolgt ohne einen definierten Prozess. Softwareentwicklung unter Code and Fix ist ein chaotischer Vorgang ohne durchgängiges Design. Das Testen findet meist fast ausschließlich am Ende der Entwicklung statt und ist unberechenbar, da die Entwicklung nicht durch Planung und qualitätssteigernde Prozesse begleitet wird. Dies führt dazu, dass man bei großen Projekten schnell die Möglichkeit verliert, die Software zu erweitern oder Fehler schnell zu finden und beheben. Diese Schwierigkeiten bringen große, verteilte Projekte oft zum Scheitern.

Definierte Prozesse und Methoden entstanden als Ansatz, um eine effektivere und planbarere Softwareentwicklung zu ermöglichen. Hierbei wird durch den Prozess ein organisatorischer Rahmen um die Softwareentwicklung gelegt. Dieser definiert zu durchschreitende Phasen, deren Inhalt und Aktivitäten, zu befolgende Standards und zu benutzende Werkzeuge, Art und Umfang der Dokumentation und Rollen, die besetzt werden sollen.

Klassische Methoden

Klassische Methoden der Softwareentwicklung (*heavyweight methods*) übertragen ingenieurwissenschaftliche Ansätze der Projektdurchführung auf Softwareprojekte. Wie bei den „klassischen“ Ingenieurwissenschaften, wird hier viel Wert auf den Prozess gelegt. Dies bedeutet, dass die Planungsphase ausführlich ausfällt und allen Phasen eine detaillierte Dokumentation folgt, die wiederum als Eingabe für die darauf folgende Phase dient.
Diesen Ansatz bezeichnet man auch als Software Engineering.

Wasserfallmodell

Das erste Softwareentwicklungsmodell wurde von Royce 1970 veröffentlicht. Hierbei werden die fundamentalen Aktivitäten der Entwicklung (Spezifikation, Design, Entwicklung und Test) als separate Phasen dargestellt. Die Resultate jeder Phase werden dokumentiert und überprüft. Dieses Ergebnis fließt in die nächste Phase ein. Werden Fehler in einer Phase gefunden, so werden sie zunächst in der aktuellen Phase korrigiert. Falls sie allerdings ihren Ursprung in einer der vorherigen Phasen haben, wird in diese Phasen zurückgegangen und die Fehler werden dort behoben.

In der Anforderungsanalyse werden in Kooperation mit dem Kunden die Ziele und Funktionalitäten des Systems entwickelt. Die daraus abgeleiteten Anforderungen an das System dienen als Systemspezifikation, zum Beispiel in Form eines Pflichten-/Lasten-Heftes. Die so festgelegte Systemspezifikation ist die Grundlage für die Festlegung der Systemarchitektur und des Systemdesigns in der Designphase. In der Implementierungsphase findet die eigentliche Softwareentwicklung statt. Das aus der vorherigen Phase stammende Designdokument dient als Vorgabe für die Implementierung. Anschließend an die Implementierungsphase folgt die Systemintegration, in der die individuellen Module zusammengefügt, integriert und als ganzes System getestet werden. Dieser als Systemtest bezeichnete Schritt bildet den Abschluss der eigentlichen Entwicklung, das System wird nun dem Kunden übergeben und befindet sich im Einsatz. Dies fällt in die Phase der Benutzung und Wartung, welches unter normalen Umständen die längste Phase innerhalb des Softwareentwicklungsprozesses ist. Die Wartung des Systems beinhaltet, dass die bei der Benutzung des Systems gefundenen Fehler korrigiert werden und Veränderungen am System gemacht werden, falls sich neue Anforderungen ergeben.

Spiral-Modell

Anstatt den Softwareprozess als eine Sequenz von Aktivitäten mit einigen Iterationen darzustellen, hat Boehm (1988) eine Präsentation des Softwareprozesses als Spirale vorgeschlagen. Dieses Modell dient insbesondere zur Planung von großen, risikoreichen Softwareprojekten. Jede Schleife in der Spirale repräsentiert eine Phase des Softwareprozesses, wie auch im Wasserfallmodell. So könnte die innerste Schleife sich mit der Anwendbarkeit, die nächste mit den Anforderungen und die darauf folgende mit dem Design beschäftigen. Im Spiralmodell fin-

den aber im Gegensatz zum Wasserfallmodell eine vorhergehende Risikoanalyse und ein abschließendes Review statt.

In jedem der Zyklen (Schleifen) im Spiralmodell werden vier Phasen durchschritten. Beim *Objective Setting* (I Quadrant) werden die spezifischen Ziele und Anforderungen für die jeweilige Phase des Softwareentwicklungsprozesses identifiziert und definiert. Hierbei erfolgt ebenfalls die Feststellung der Risiken des Projektes, aus denen alternative Strategien abgeleitet und entwickelt werden. Die hier festgestellten Risiken hat innerhalb des *Risk Assessment and Reduction* (II Quadrant) eine detaillierte Analyse zur Folge. Es werden die nötigen Schritte zur Risikoanalyse vorgenommen und gegebenenfalls Prototypen entwickelt, um verschiedene Risiken zu überprüfen. Nach der umfassenden Risikoanalyse wird innerhalb des *Development and Validation* (III Quadrant) ein Entwicklungsmodell für das System ausgewählt. In dieser Phase geschieht in späteren Iterationen die eigentliche Entwicklung. Die Planung der nächsten Iteration erfolgt im *Planning* (IV Quadrant). Hier wird nach Überprüfen und Reviewen der abgelaufenen Phase des Projektes entschieden, ob mit einem weiteren Zyklus in der Spirale fortgefahren wird. Im Falle des Fortfahrens werden Pläne zur weiteren Vorgehensweise erstellt.

Agile Methoden

Agile Methoden der Softwareentwicklung (*lightweight methods*) entstanden Mitte der 90er Jahre als Reaktion auf die bürokratischen, dokumentlastigen klassischen Methoden und deren Schwierigkeiten im Umgang mit sich schnell verändernden Anforderungen. Ihr Fokus liegt auf der eigentlichen Softwareentwicklung und nicht auf Prozess, Design oder Dokumentation. Um das Ziel der schnellen Anpassung an Änderungen zu erreichen, pflegen agile Methoden sehr kurze Iterationen, bei denen inkrementell entwickelt wird. Durch kurze Iterationen und kurze Release-Zyklen wird schnelles Feedback beim Kunden über den Fortschritt und Funktionsumfang eingeholt. Dieses schnelle Feedback dient zur Steuerung der weiteren Entwicklung.

Im „Agile Manifesto“ drücken die führenden agilen Verfechter ihre Wertevorstellung der agilen Entwicklung aus. Das Agile Manifesto stellt Individuen und deren Interaktion über Prozesse und Pläne, d.h. die Konzentration der Methodologie findet auf individueller Ebene und nicht auf abstrakter Prozessebene statt. Die Interaktion der Individuen führt zu besseren Ergebnissen als die Verfolgung eines Plans, der die gewünschte Realität schlecht widerspiegelt und träge auf Veränderungen reagiert. Wichtig ist auch die aktive Einbindung des Kunden in die Entwicklung, d.h. durch ständige Kommunikation und Feedback des Kunden werden seine Vorstellungen besser realisiert als wenn man strikt den gedruckten Anforderungskatalog implementiert. Ziel einer agilen Entwicklung ist es möglichst schnell dem Kunden eine funktionierende und ihm wertliefernde Software zu übergeben, auch wenn dazu eine geringere oder gar andere Funktionalität implementiert wird, als ursprünglich vereinbart. Dies bedeutet beispielsweise, dass vom Kunden kommunizierte Änderungswünsche umgesetzt werden, obwohl sie nicht im ursprünglichen Vertrag enthalten waren.

Trotz dieser Einigkeit in der Zielsetzung und Wertevorstellung der Softwareentwicklung hat sich seit der Mitte der 90er doch eine beachtliche Anzahl von agilen Methoden entwickelt. Eine grobe Übersicht ist auf Folie 16 der Präsentation gegeben. Von den genannten Methoden wird exemplarisch Extreme Programming näher erläutert.

Extreme Programming

Extreme Programming (XP) ist ein von Kent Beck, Martin Fowler, Erich Gamma und Ward Cunningham aus der Praxis für kleine bis mittelgroße Softwareprojekte entwickelter agiler Prozess, der speziell darauf abzielt im Umfeld der sich schnell wandelnden Anforderungen zu bestehen. Der Name Exterme Programming leitet sich von der Tatsche ab, dass bestehende und bekannte Verfahren ins Extreme getrieben und kombiniert werden.

Der Grundgedanke hinter XP ist, dass die Cost of Change (als Summe) nicht mit der Zeit exponentiell ansteigen, sondern durch bestimmte Verfahren nahezu konstant gehalten werden können. Wichtig ist hierbei, dass XP nicht annimmt, dass Fehler oder Änderungen, die spät im Programm auftauchen werden weniger Kosten verursachen. Die Annahme ist hier, dass weniger Änderungen und Fehler gefunden werden und dass Änderungen leichter durchzuführen sind. Somit ist die Summe der Kosten geringer.

Diese Annahme wird in XP durch ein einfaches Design (Simplicity), automatische Tests und ständiges Refactoring gerechtfertigt. Der Gedanke ist, dass man ein einfaches Design entwickelt, dass die aktuellen Anforderungen erfüllt und dieses durch ständige Veränderungen in die richtige Richtung und durch ständiges Refactoring spätere Änderungen einbindet. Als Konsequenz ergibt sich, dass man kein „design for tomorrow“ betreibt. Das heißt man implementiert nur das jetzt Benötigte und spekuliert nicht auf mögliche weitere Erweiterungen. Die einfachste Lösung, die funktioniert ist die Richtige. Erst wenn ein Feature tatsächlich benötigt wird, wird es

implementiert und das bisherige Design durch Refactoring in diese Richtung in kleinen, inkrementellen Schritten gebracht.

Die wichtigsten Bestandteile und Verfahren von XP werden im Folgenden kurz erläutert:

- **Kurze Release-Zyklen:** Durch kurze Iterationen und Releases wird schnell Feedback vom Kunden eingeholt. Dies dient dazu, die bisherige Entwicklung zu bewerten und Änderungen möglichst früh einzubringen.
- **Metapher:** Bei XP wird die Systemarchitektur oft gegen eine Metapher getauscht, da es keine große Planungsphase für das Design gibt und das Design sich ständig im Rahmen des Refactoring verändert. Eine zentrale Metapher bündelt hier die Anstrengungen der Entwickler und richtet diese auf eine gemeinsame Zielvorstellung aus. So kann beispielsweise der Satz „das Programm soll als Schreibmaschine funktionieren" als Anfang für die Bildung einer Metapher für eine Textverarbeitung gesehen werden.
- **Einfaches Design (Simplicity):** Das Design ist möglichst einfach zu halten, so dass es trotz fehlender Design-Dokumente von allen verstanden werden kann. Refactoring und Änderungen werden so vereinfacht.
- **Ständiges Refactoring:** Das Programm wird ständig durch kleine Änderungen vereinfacht und verbessert. Ziel ist es Duplizitäten zu entfernen und das Programm den Anforderungen anzupassen.
- **Kollektiver Besitz des Quellcode (Collective Code Ownership):** Die Verantwortlichkeit des Quellcodes liegt bei allen Entwicklern zu gleichen Teilen. Es gibt keine „Code-Owner". Somit ist jeder Entwickler gezwungen, sich mit jedem Teil des Codes auseinander zu setzten, und Refactoring kann betrieben werden, ohne Änderungen erst beim Code-Owner einzureichen.
- **Kunde vor Ort (On-site Customer):** Ein Kundenrepräsentant ist Teil des Entwicklungsteams und gestaltet somit die Software mit. Der Kunde entwickelt auch die funktionalen Tests und klärt Fragen der Entwickler. Dies ist Voraussetzung für das schnelle Feedback.
- **Codier Standards (Coding Standards):** Standards sind nötig, damit jeder Entwickler schnell und einfach Änderungen und Refactoring betreiben kann.
- **40 Stunden Woche:** Da das Design nur in den Köpfen der Entwickler existiert, ist es wichtig, dass diese ausgeruht und einsatzfähig sind. Dies ist nur gewährleistet, wenn diese genug Ruhezeit hatten.
- **Pair-Programming:** Pair-Programming bedeutet, dass zwei Entwickler an einem Rechner (mit nur einer Tastatur und einer Maus) entwickeln. Einer schreibt den Code, der andere überprüft diesen und denkt über das weitere Vorgehen nach. Dadurch dass ein anderer Entwickler auf den Code schaut, ist dieser sauberer, befolgt eher die Codier Standards und ist besser durchdacht. Man diskutiert Lösungen mit dem Programmier-Partner, bevor sie umgesetzt werden. Dies erhöht die Code-Qualität und sorgt auch dafür, dass beide den Code verstehen. Pair-Programming-Partner werden am Tag mehrmals gewechselt, so dass sichergestellt ist, dass jeder von dem Wissen der anderen profitieren kann. In der Praxis ist Pair-Programming durchaus effektiv, die Produktivität der beiden Partner zusammen ist höher als die Summe der einzelnen Produktivitäten.

Des Weiteren stellen das „Planning Game" (die kurze Planung am Anfang der Iteration), das Testen und die kontinuierliche Integration wichtige Säulen von XP dar. Diese werden nun ausführlicher erläutert.

Planning Game

Das Planning Game (Planspiel) in XP ist das Gegenstück zur Projektplanung bei den klassischen Methoden. Im Gegensatz zur klassischen Planung ist es nur auf einen kurzen Zeithorizont fixiert. Ziel ist es nicht das ganze Projekt/Produkt detailliert zu planen, sondern nur den Inhalt und Umfang von maximal zwei Iterationen zu definieren. Eine Iteration beträgt hierbei zwischen zwei und vier Wochen.

Das Planning Game lässt sich in zwei Planungsarten unterteilen, der Releaseplanung und der Iterationsplanung. In der Releaseplanung wird der Umfang der einzelnen Releases in einem Treffen der Entwickler mit dem Kunden festgelegt, während in der Iterationsplanung die Entwickler die nötigen Aufgaben untereinander verteilen.

Die Releaseplaung ist in drei Phasen gegliedert. In der *Exploration Phase* schreibt der Kunde seine Anforderungen (Customer Stories) an die Software auf so genannte Story Cards. Diese kleinen Karteikarten beschreiben kurz eine Funktion und dienen später als Grundlage für die funktionalen Tests. Neben der Beschreibung werden noch einige Angaben, wie die Nummer oder das Datum vermerkt. Die Entwickler belegen nun jede Story Card mit einem geschätzten Umfang. Diese Schätzung basiert auf Vergangenheitswerten oder Expertenschätzungen. In der *Commitment Phase* legt der Kunde die Priorität der Story Cards fest, während die Entwickler ihre Schätzung mit einem Risko versehen. Diese neuen Informationen werden nun als Grundlage für die *Steering Phase* genutzt, in der eine Iteration über die vorherigen Phasen durchgeführt wird. So können von beiden Seiten die Information zur Neubewertung der Story Cards genutzt werden. Ergebnis der Iteration ist nun eine Kategorisierung der Story Cards nach Umfang, Risiko und Priorität. Dies hilft dem Kunden eine Auswahl der Story Cards

für das nächste Release vorzunehmen, wobei er durch die von den Entwicklern angegebene Gesamtarbeitszeit beschränkt ist. Ergebnis der Releaseplanung sind die Story Cards, die bis zum nächsten Release zu implementieren sind.

In der Iterationplanung wird nun dieser Stapel Story Cards unter den Entwicklern aufgeteilt. Hierbei werden wieder die drei Phasen *Exploration, Commitment* und *Steering* durchschritten. In der ersten Phase schreiben die Entwickler gemeinsam für jede Story Card eine bis mehrere Tasks Cards. Dies sind ebenso Karteikarten, auf denen Aufgaben vermerkt sind, die zu Realisierung der Funktionalität einer Customer Story nötig sind. In der Commitment Phase übernehmen nun die einzelnen Entwickler eine Tast Card und schätzen dabei selber deren Umfang ab. Wichtig hierbei ist, dass nur der zuständige Entwickler den Umfang seiner Task Card abschätzt, da er die Aufgabe auch später realisieren muss und bei einer Fremd- oder Kollektivschätzung seine tatsächliche Arbeit noch weniger mit der Schätzung übereinstimmen würde. Stellt ein Entwickler fest, dass er zu wenig oder zuviel Aufgaben hat, so tritt er diese wieder an die Gruppe ab und sie werden wieder neu verteilt. In der Steering Phase findet nun die eigentliche Entwicklung der Aufgaben oder Tast Cards statt. Sind alle Task Cards einer Story Card implementiert, wird diese anhand der funktionalen Tests überprüft. Am Ende der Iteration wird eine neue Release- und Iterationsplanung angesetzt.

Testen und kontinuierliche Integration

XP unterscheidet drei Arten von Tests:

- Unit-Tests: Vom Entwickler geschriebene Test, die seinen Code auf Korrektheit überprüfen.
- Funktionale Tests: Vom Kunden geschriebene Tests, die die Erfüllung der Funktion/Story Cards überprüfen.
- Weitere Tests: Stresstests, Paralleltest und andere Tests werden am Ende einer Iteration durchgeführt.

Der Schwerpunkt von XP liegt beim Unit- und Funktionalen Testen. Diese werden nun näher erklärt.

Implementiert ein Entwickler eine Tast Card (mit einem Pair-Programming-Partner), so schildert sich der Prozess unter XP wie folgt: Zunächst implementiert er alle Tests, die die zu implementierende Funktionalität überprüfen könnten. Erst wenn die Tests (Unit-Tests) geschrieben sind, wird die eigentliche Funktion implementiert. Ist dieser Prozess abgeschlossen, so werden alle geschriebenen Tests genutzt, um den Code zu verifizieren. Läuft ein Test nicht durch, so wird der Code korrigiert, bis alle Tests zu 100% laufen. Wichtig hierbei ist, dass die Tests vorher geschrieben werden. So macht sich der Entwickler vor der Implementierung Gedanken über das Problem und erörtert so im Kopf verschiedene Lösungen. Laufen alle Tests zur neuen Funktionalität, so wird diese in das System integriert und alle Regressionstests und Tests der anderen Entwickler werden gestartet. Durch diese kontinuierliche Integration werden Fehler schnell gefunden und sich konkurrierende Arbeiten und Änderungen schnell entdeckt. Alle Tests müssen hier zu 100% laufen. Tauchen Fehler auf, so sind diese umgehend zu korrigieren. Sind alle Fehler behoben, kann sich der Entwickler einer neuen Task Card zuwenden.
Diese Art der Entwicklung wird auch Test Driven Development genannt, wobei es hier allerdings auch Formen gibt, in denen immer erst ein Test geschrieben wird, dann soweit Implementiert wird, dass er läuft und erst nun der nächste Test geschrieben wird.

Funktionale Tests zur Überprüfung einer Story Card werden vom Kunden geschrieben. Da er meist dazu nicht in der Lage sein wird, formuliert er die Tests und ein Entwickler implementiert diese. Sinn ist es die Erfüllung der geforderten Funktionalität nachzuweisen. Der Kunde weiß so, dass das Programm das leistet, was er sich vorgestellt hat und der Entwickler hat so auch die Gewissheit, dass er das Richtige implementiert hat. Dies verschafft beiden Seiten Zuversicht in die Entwicklung und schnelles Feedback, ob gewünschte Funktionalität auch im Sinne des Kunden implementiert wurde.

Vergleich und Bewertung

Bei der Wahl zwischen klassischen und agilen Methoden, muss man beachten, dass nicht jede Methode für jedes Projekt geeignet ist. Beide Methoden weisen spezifische Vor- und Nachteile aus, die sie für verschiedene Situationen auszeichnen. Um die, für das jeweilige Projekt, richtige Wahl zwischen diesen Methoden zu treffen, muss man die wichtigsten Charakteristika und Risiken des Projektes kennen und verstehen. Es gibt verschiedene Kriterien anhand deren Ausprägung man eine Entscheidung treffen kann. Beispielhaft werden hier folgende Kriterien behandelt: Entwickler, Kunde, Anforderungen, Größe, Zielsetzung und Wartbarkeit.

Agile Methoden beruhen auf der Kommunikation zwischen den Entwicklern untereinander und mit dem Kunden. Auch das komplette Design existiert lediglich in den Köpfen der Entwickler. Somit stellen sich hohe Persönlichkeitsanforderungen an den Entwickler, da dieser eben talentiert und kommunikativ genug sein muss, um auch unter diesen Umständen zu bestehen. Diese Forderung impliziert jedoch nicht, dass ausschließlich hoch

fähige Mitarbeiter in einem Projektteam sein müssen. Auch ein gesunder Mix aus erfahrenen und weniger erfahrenen Entwicklern kann bei agilen als auch bei klassischen Methoden zum Erfolg führen, mit dem Unterschied, dass man bei agilen Methoden sich stark auf das implizite Wissen der Entwickler verlässt. Dies birgt allerdings auch eher die Gefahr in sich, dass diese kurzsichtige Entscheidungen treffen oder gar schwerwiegende Architekturfehler begehen. Klassische Methoden benutzen Pläne und Dokumente, um dieses Risiko zu minimieren. Diese Pläne und Dokumente können dann zu weiterer Überprüfung an externe Experten weitergegeben werden. Dabei riskiert man jedoch, dass bei schneller Anforderungsänderung diese Pläne überflüssig werden oder nur unter hohen Kosten aktuell gehalten werden können.

Bei der Auswahl der Methode spielen die Kunden eine sehr wichtige Rolle. Um agil arbeiten zu können, müssen sich diese stark in das Projekt involvieren und bei der Entwicklung eine aktive Rolle einnehmen. Der Kunde (On-site Customer) muss bereit sein, Arbeit und Zeit zu investieren. Durch seine Präsenz ist es erst möglich, das für die agile Entwicklung nötige schnelle Feedback zu erlagen. Auch muss der Kunde in der Lage sein, alle späteren Anforderungen an das System zu beschreiben, da er die funktionalen Tests gestaltet. Klassische Methoden kommen mit deutlich weniger Involvierung des Kunden aus. Dieser stellt zu Projektbeginn die Anforderungen an das System und erarbeitet mit dem Projektteam die Anforderungsdokumente. Oftmals ist er dann nur noch bei Meilensteinen eingebunden.

Auch die Rolle der Anforderungen bestimmt die Wahl der Methode. Agile Methoden eigenen sich in den Fällen besonders gut, in denen die Anforderungen zu Projektbeginn nicht vollständig vorliegen und Anforderungen sich im Projektverlauf stark ändern. Dies ist eines der Ziele von agilen Methoden, die Fähigkeit Änderungen auch spät im Projektverlauf aufzunehmen. Sind hingegen die Anforderungen relativ stabil oder im Vorhinein bekannt, zahlt es sich eher aus klassisch vorzugehen, da man nun die Architektur und das Design schon in der frühen Phase des Projektes festlegen kann. Allerdings reagieren diese Methoden auf Änderungswünsche des Kunden sehr unflexibel.

Ein weiteres Kriterium zur Auswahl der Methode ist die Größe des Projektes und somit des Projektteams. Es hat sich herausgestellt, dass agile Methoden in großen Teams schwer umzusetzen sind. Die Koordination des stark zusammen agierenden Projektteams, das für agile Methoden außerordentlich wichtig ist, wird bei einer Teamgröße von über 20 Personen besonders schwierig. Es sind kleine Teams notwendig, um die räumliche Nähe zu gewährleisten, die unter anderem für das Pair-Programming und die Kommunikation der Teammitglieder von besonderer Bedeutung ist. Große und verteilte Projekte lassen sich eher mit klassischen Methoden handhaben, da diese sich nicht auf die Face-2-Face Kommunikation der Teammitglieder verlasen, sondern auch Dokumente und Pläne, die sich leicht digital verbreiten lassen. Durch die detaillierte Planung lassen sich auch Teilprojekte und –aufgaben leichter identifizieren und können weitgehend unabhängig von einander gelöst werden.

Die Zielsetzung des Projektes kann ebenfalls zur Entscheidungsfindung herangezogen werden. Zielt das Projekt auf schnelle Auslieferung funktionierender Software ab, so sind agile Methoden besonders gut geeignet, denn dies ist die höchste Priorität der agilen Softwareentwicklung. Jedoch birgt die frühe Auslieferung der Software die Gefahr des Treffens von kurzsichtigen Entscheidungen. Denn das Design, welches für die erste Version der Software entwickelt wurde, skaliert für das Endprodukt eventuell nicht genug. Das können Klassische Methoden zumeist verhindern, da hier das Design ja von Anfang an für das Endprodukt erstellt worden ist. Zudem sehen klassische Methoden als ihr Hauptziel die korrekte und vollständige Implementation der Anforderungen, was sie besonders geeignet für sicherheitskritische Projekte macht. Auch stehen bei diesen Methoden die Vorhersagbarkeit, Wiederholbarkeit und Optimierung der Prozesse im Vordergrund, was bei sich ändernden Anforderungen wiederum zum Nachteil werden kann.

In die Entscheidung sich für oder gegen eine Methode zu entscheiden, sollte auch mit eingehen, welche Rolle die Wartbarkeit des Produktes spielt. Im Hinblick auf die Wartbarkeit ist es wichtig zu wissen, dass agile Methoden sich dadurch auszeichnen, dass der Hauptteil des Designs und der Planung in den Köpfen der Mitarbeiter stattfindet und nicht oder nur geringfügig dokumentiert wird. Somit ist die Dokumentation oder die Planung nur in den Köpfen der Teammitglieder verfügbar. Sind diese Teammitglieder nicht mehr verfügbar, so ist auch das Wissen in ihren Köpfen über das System verloren. Dies erschwert eine Wartung der Software, erstrecht durch ein fremdes Team. Diesem Problem entgegnen klassische Methoden durch die ausführliche Dokumentation am Ende jeder Phase. Diese Dokumentation kann als Grundlage für die Wartung und Erweiterung der Software dienen, wobei ihre Qualität und Aussagekraft eine wichtige Rolle spielt. Die Dokumentation muss erweitert und aktualisiert werden, falls Änderungen in der Software vorgenommen werden.

Dies ist eine Auswahl der Kriterien, die zur Entscheidungsfindung herangezogen werden können. Jedoch ist es vor allem für den E-Business Sektor schwierig, sich für die schnelle Entwicklung oder hohe Zuverlässigkeit und Korrektheit zu entscheiden. Sie benötigen vielmehr eine Kombination von Eigenschaften. Somit sind Methoden

gefragt, die diese Eigenschaften in sich vereinen können. Hierfür eignen sich besonders die so genannten hybriden Methoden, wie der Rational Unified Process (RUP). Diese von Rational Software Corporation zur Unterstützung und Umsetzung des Designs mit UML entwickelte Methode ist eine eher klassische Methode zur Softwareentwicklung. Allerdings lässt sie sich mit agilen Elementen erweitern und so ihre Vorteile nutzen. Dazu kann man z.B. in einer der Phasen des klassischen Prozesses die eigentliche Entwicklung agil vornehmen.

Abschließend lässt sich sagen, dass keine der Methoden in jeder Situation der anderen überlegen ist. Der Einsatz jeder Methode ist vorher mit den herrschenden Bedingungen und dem Anwendungsgebiet der Methode abzustimmen.

Literatur

Bücher und Artikel:

Beck, K. (2000). *Extreme programming explained*. Addison-Wesley.
Beck, K. (2003). *Test driven development by example*. Addison-Wesley.
Boehm, B. (2002). Get ready for agile methods, with care. *Computer*, Januar 2002, pp. 64-69.
Sommerville, I. (2004). *Software engeneering*. Addison-Wesley.

Webseiten:

Agile Alliance - www.agilealliance.org
Extreme Programming - www.xprogramming.org
Extreme Programming - www.extremeprogramming.org