1 Die klassische bzw. statische Softwareentwicklung

1.1 Definition

Klassische bzw. statische Softwareentwicklung ist eine Form der Softwareentwicklung bei welcher ein „Software Development Life Cycle (SDLC)“ nach festem Ablaufplan umgesetzt wird. [1]

1.2 Funktionsweise der klassichen bzw. statischen Softwareentwicklung

Die klassiche Softwareentwicklung funktioniert nach dem Prinzip des SDLC. Dieser besteht aus 6 Phasen, welche alle nacheinander vom Entwicklungsteam abgearbeitet werden müssen. [1] (siehe Abbildung 1)

1.2.1 Analyse und Planung

Die erste Phase dieses SDLC besteht aus der Analyse und Planung des Projekts. Sie ist gleichzeitig auch der wichtigste Entwicklungsabschnitt. Deshalb sollte dieser Schritt von den erfahrenen Teammitgliedern ausgeführt werden, um das bestmögliche Produkt zu erzeugen.
In der ersten Phase der Entwicklung wird ein Projektplan aufgestellt, der festlegt, wie das Projekt aussehen soll und welche Schritte durchlaufen werden. Dabei kommt es natürlich auch auf das genaue Modell an, welches zur Entwicklung benutzt wird. Ein Beispielmodell wird später genauer erklärt (siehe Das Wasserfallmodell)
Außerdem wird eine sogenannte Machbarkeitsstudie durchgeführt. Hierbei beschäftigt man sich damit, ob das Projekt aus wirtschaftlicher, bertrieblicher und technischer Sicht realisierbar ist. Das Ergebnis dieser Studie enthält verschiedene Softwareentwicklunsmethoden, welche dann nach geringstem Implementierungsrisiko ausgewählt werden können.
Zudem wird in dieser Phase geplant, welche Inhalte für die Software unbedingt erforderlich sind. Außerdem werden Projektrisiken identifiziert um dagegen vorzugehen. [1]
Ist dies geschafft, wird zur zweiten Phase übergegangen.

1.2.2 Definition der erforderlichen Inhalte

Im ersten Schritt wurden bereits die erforderlichen Inhalte der Software klar definiert und dokumentiert. In Rücksprache mit dem Kunden, werden diese dann gegebenfalls überarbeitet.
Die festgelegten Ziele werden in einer „Software Requirement Specification (SRS)“ festegehalten. Die SRS ist dann eine Auflistung aller erforderlichen Inhalte der Software. [1]
Danach geht es in der dritten Phase mit dem Projektaufbau weiter.

1.2.3 Projektarchitektur

In der dritten Phase des SDLC, entwickeln die Developer das Design bzw. die Architektur des Produktes. Dabei arbeiten sie mehrere Aufbaumöglichkeiten aus. Diese Möglichkeiten werden dann von allen interessierten Gruppen eingesehen und danach die Entscheidung für eine Variante gefällt. Die Entscheidung wird durch Abwägen bestimmter Kriterien, wie zum Beipsiel Risiko, Produktrobusheit, Designmethode, Budget, Dauer, etc., getroffen. [1]
Am Ende dieser Phase, ist der Aufbau des Produktes klar definiert und man kann zum Programmieren übergehen.

1.2.4 Produktimplemetierung

In der vierten Phase startet die Produktentwicklung. Das bedeutet, dass nun der Code geschrieben wird. Dabei wird sich an das Prinzip, welches in der dritten Phase, Projektarchitektur, festgelegt wurde, gehalten. Dabei muss sich zwischen verschieden Modellen und auch verschiedenen Programmiersprachen, zum Beispiel Java oder Phython entschieden werden. Dabei werden diese nach der Software, welche hergestellt werden soll, ausgewählt.[1]
Nachdem die Software programmiert wurde, muss sie in der nächtsten Phase getestet werden.

1.2.5 Testen des Produkts

In dieser Phase werden Fehler im Produkt von Testern getestet. Fehler die auftreten werden gemeldet, dann von den Programmierern verfolgt und später behoben.
Diese Phase erfolgt eigentlich auch schon in den anderen Phasen der Entwicklung, da es gerade bei moderneren Produkten schon während der Entwicklung getestet wird. [1]
Wenn alle Fehler behoben wurden, kann das Projekt veröffentlicht werden.

1.2.6 Veröffentlichung und Überareitung

Wenn all diese Schritte abgearbeitet wurden, kann das Produkt erst einmal auf einem Teil des Marktes veröfftlicht werden. Dann wird das Produkt unter realen Marktbedingungen getestet und Feedback eingeholt.
Je nachdem, wie dieses Feedback ausfällt, gibt es mehrere Möglichkeiten.

- Wenn keine Fehler gefunden werden, kann das Produkt gleich auf dem ganzen Markt veröffentlicht werden.
- Wenn Fehler gefunden werden, wird das Produkt überarbeitet und die Fehler behoben und mögliche Verbesserungen eingebaut. Danach wird das Produkt global gelauncht.

Werden nach der Veröffentlichung noch Probleme festgestellt, werden diese mit Updates behoben. [1]

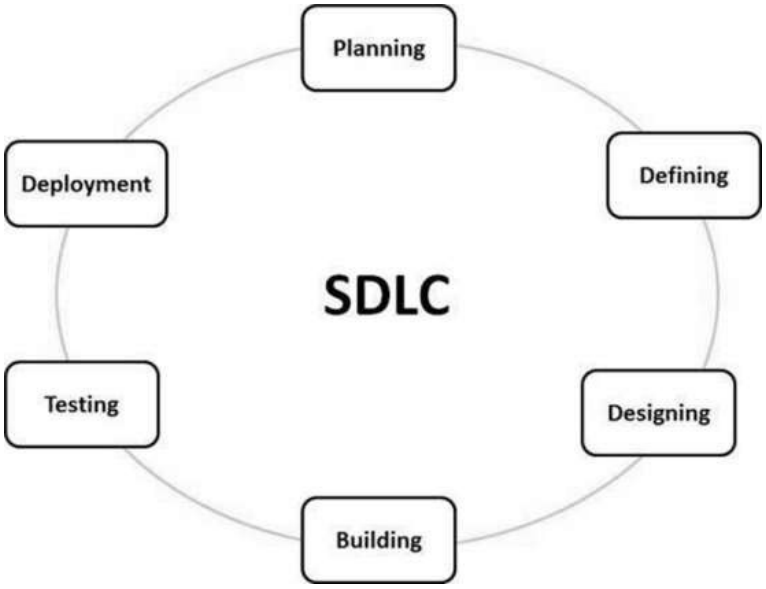

Abbildung 1: Der SDLC (Englisch)

1.3 Ziele der klassischen bzw. statischen Softwareentwicklung

Wie oben erwähnt, arbeiten klassische Softwareentwicklungsmodelle nach einem festen Plan (SDLC), welcher im Vorhinein festgelegt wird. Dieser Plan ist schwer veränderbar, d.h. Änderungen in den Anforderung müssen strikt überprüft werden, bevor sie implementiert werden.
Da jeder Arbeitsschritt klar festgelegt ist, ist es für die Entwickler einfach zu verstehen, was in dieser Phase der Entwicklung zu tun ist.
Damit dies funktioniert, ist es wichtig, dass die Anforderungen an das Produkt genau analysiert und festegehalten werden. Zudem ist eine sorgfältige Planung des Projektes von großer Wichtigkeit.
Damit Übersicht hergestellt werden kann, wird der Entwicklungsprozess sorgfältig dokumentiert und damit Fehler sofort festgestellt werden, wird die getane Arbeit streng kontrolliert. [1] Da nun die grundlegende Funktionsweise der klassischen bzw. statischen Softwareentwicklung bekannt gemacht wurde, wird im Weiteren ein Beispielmodell genauer erläutert.

1.4 Das Wasserfallmodell

1.4.1 Aufbau des Wasserfallmodells

Das Wasserfallmodell funktioniert nach einem einfachen Prinzip: Eine Phase muss abgeschlossen sein, damit die nächste beginnen kann. Am Ende jeder Phase wird das

Produkt überprüft, ob es die Anforderungen des Kunden erfüllt.
Beim Wasserfallmodell gibt es fünf Phasen, welche insgesamt dem SDLC ähneln.
In der ersten Phase werden die Anforderungen an das zu entwicklende Podukt zusammengetragen und analysiert.
Danach folgt das Designen des Systems in Phase zwei.
Darauf folgt Phase drei, bei welcher implementiert wird.
In Phase vier wird das Programm getestet.
In der finalen Phase fünf wird das System bereitgestellt und das System gegebenenfalls gewartet.
Alle Phasen werden genau dokumentiert um Übersicht herzustellen. [1]
Aus diesem Aufbau des Modells ergeben sich sowohl Vorteile, als auch Nachteile.

1.4.2 Vorteile des Wasserfallmodells

Das Wasserfallmodell ist durch die Gliederung in Phasen, also einzelne Arbeitsschritte mit klaren Anforderungen, welche nacheinander ausgeführt werden, leicht zu verstehen und anzuwenden. Dies ist zum Beispiel von Vorteil, wenn es neues Teammitglied zum Team stößt.
Außerdem wird durch das stufenweise Vorgehen eine einfache Koordination gewährleistet. In jeder Phase muss ein bestimmtes, festgelegtes Zeil erreicht werden. Das ist einfach zu verstehen und durchzuführen. [1]
Jedoch hat das Wasserfallmodell auch Nachteile.

1.4.3 Nachteile des Wasserfallmodells

Ein Nachteil des Wasserfallmodells ist es, dass es keine Flexibilität bei Anforderungsänderungen gibt, denn wenn diese eintreten, muss das Projekt neugeplant werden, wodurch von vorn begonnen werden muss.
Zudem gibt keine Prototypen des Produktes, bis alle Phasen durchlaufen wurden. Dadurch, weiß man bis zuletzt nicht, ob das Produkt den Anforderungen entspricht.
Außerdem ist es schwierig Probleme, welche beim Testen entstehen, zu beheben, da man dafür in die Systemdesignphase (zweite Phase) zurückkehren muss und somit damit nochmals alle anderen Phasen durchschreiten muss. [1]

2 Die Agile Softwareentwicklung

2.1 Definition

„ Agile Softwareentwicklung ist eine Entwicklungsmethode die schnell auf veränderte Anforderungen reagieren kann, da die Entwicklung in vielen kleinen und abgeschlossen Zyklen abläuft und eine erhöhte Kommunikation zwischen den Entwicklern stattfindet.“[1]

2.2 Leitsätze der Agilen Softwareentwicklung

Die Leitsätze des Agiles Modells wurden von 11 Softwareentwicklern, unter der Führung von Kent Beck im „Manifest für Agile Softwareentwicklung“ festgehalten. Es gibt vier wesentliche Leitsätze der Agilen Softwareentwicklung, welche sie von anderen Softwareentwicklungsmodellen abgrenzt:

- „**Individuen und Interaktionen** mehr als Prozesse und Werkzeuge“ [2]
- „**Funktionierende Software** mehr als umfassende Dokumentation“ [2]
- „**Zusammenarbeit mit dem Kunden** mehr als umfassende Vertragsverhandlungen“ [2]
- „**Reagieren auf Veränderung** mehr als das Verfolgen eines Plans“ [2]

Angemerkt sei, dass die Entwickler des Agilen Modelles keinesfalls die Werte auf der rechten Seite als unwichtig betrachten, sondern die Werte auf der linken Seite als wichtiger im Prozess der Softwareentwicklung einschätzen.
Außerdem gibt es im Manifest eine Erweiterung auf 12 Prinzipien, welche die oben genannten Leitsätze näher erklären.[2]

2.3 Prinzipien der Agilen Softwareentwicklung

Ein Prinzip des Agilen Modells, dass die Menschen, welche an der Entwicklung beteiligt sind miteinander interagieren.

Im Agilen Modell ist es wichtig, dass die Entwickler dem Kunden oft ihr Zwischenergebnis schicken, um den Kunden zufrieden zu stellen.

Dazu gehört die enge Zusammenarbeit zwischen Fachexperten und die Softwareentwickler sollten währendes Projektes täglich miteinander arbeiten.

Die Entwickler sollten Kunden alle Anforderungsänderungen erfüllen, um dem Kunden einen Wettbewerbsvorteil verschaffen zu können.

[1] aus:`https://www.gruenderszene.de/lexikon/begriffe/agile-softwareentwicklung?interstitial_click`

Die Entwickler sollten dem Kunden funktionierende Software innerhalb eines bestimmten Zeitraumes (einige Wochen oder Monate, aber schnellstmöglich) liefern, damit der Kunde den Fortschritt sehen kann.

Zudem sollte auf motivierte Individuen vertraut werden. Es sollte ihnen die Ünterstützung gewährleistet werden, damit sie die Aufgabe erledigen können. Das Wichtigste ist das Vertrauen in die Individuen, dass sie die Aufgabe bewältigen können.

Um im Entwicklungsteam Informationen auszutauschen, ist es die beste Methode dies im Gespräch zu erledigen.

Die Entwickler, Auftraggeber und die Benutzer sollen ein gleichmäßiges Tempo auf unbegrenzte Zeit halten können, um nachhaltige Entwicklung zu fördern.

Ständige Priorität sollten technische Exzellenz und gutes Design haben, um die Agilität zu fördern.

Einfachheit ist beim Agilen Modell sehr wichtig und wird als „die Kunst, die Menge nicht getaner Arbeit zu maximieren“[2] verstanden.

Die besten Ergebnisse mit diesem Modell können durch selbstorganisierte Teams erzielt werden.

Das gesamte Team soll in regelmäßigen Abständen selbst reflektieren, wann es seine Effektivität verbessern kann. [2]

2.4 Paarprogrammieren

Wie oben schon erwähnt, liegt bei der Agilen Softwareentwicklung mehr Fokus auf den Entwicklern und deren Interaktion, als auf den Werkzeugen, welche zur Programmierung verwendet werden. So kommt es, dass sich Methoden entwickelt haben, welche sich besonders gut für Agile Softwareentwicklung eignen.
Dazu wurde zum Beispiel die Methode des Paarprogrammierens (auch „Pair Programming“ genannt) entwickelt. Dabei lösen zwei Entwickler eine Aufgabe an einem gemeinsamen Rechner.
Ein Programmierer ist dabei „Driver“ und der andere „Observer“. Der Driver hat die Kontrolle über das bzw. die Eingabegeräte. Das bedeutet, dass er den Code schreibt oder designt.
Der Observer hingegen „überwacht“ den Driver. Seine Aufgabe sind es:

- auf Fehler aufmerksam zu machen, welche vom Driver begangen werden,
- sich Alternativen auszudenken,
- Resourcen zu finden, welche zur Entwicklung gebraucht werden oder

- die Auswirkungen der aktuellen Arbeit einzuschätzen und im Gesamtkontext zu beachten.

Im Entwicklungsprozess werden die Rollen immer in regelmäßigen Zeitabständen getauscht. Pair Programming ist sehr erfolgreich und wird daher oft in der Industrie eingesetzt.
Es verspricht eine höhere Qualität des Produktes mit geringerem Zeitaufwand. Dies wurde auch durch eine Studie der University of Utah belegt. Außerdem stellte sich heraus, dass sich die Programmierer beim Paarprogrammieren sicherer in ihrer Arbeit fühlen und mehr Spaß an der Arbeit haben, wenn sie zu zweit arbeiten anstatt allein. Wenn allein programmiert wird, mündet es bei Drucksituationen (zum Beispiel: wenig Zeit) in unsauberem Arbeiten, wodurch die Qualität des Endproduktes sinkt. Wenn jedoch zu zweit programmiert wird, ist es leichter einen „kühlen Kopf“ zu bewahren, weil der Partner helfen kann, wenn es Komplikatonen geben sollte. [3]

2.5 Ziele der Agilen Softwareentwicklung

Aus den bereits aufgezeigten Prinzipien und Leitsätzen der Agilen Softwareentwicklung, kann man nun die Ziele auch die damit verbunden Vorteile ableiten.
Schon die Prinzipien, die im „Agilen Manifest“ niedergeschrieben wurden, geben Aufschluss darüber, was die Entwickler, welche dieses Modell der Softwareentwicklung nutzen, hinaus wollen.
Es wird Wert auf „Individuen und Interaktionen, Funktionierende Software, Zusammenarbeit mit dem Kunden und das Reagieren auf Veränderung“ [2] gelegt. Wie man erkennen kann, liegt der Fokus der Softwareentwicklung eher auf den Menschen, welche am Prozess der Entwicklung teilnehmen. So steht zum Beispiel der Kunde mit seineN Wünschen im Vordergrund. Deshalb ist das Modell so ausgelegt, dass die Entwickler schnell auf Kundenwünsche bzw. -anregungen reagieren können.
Zudem werden auch die Entwickler mehr berücksichtigt und es werden auch spezielle Arbeitsmethoden angewendet (siehe oben).
Außerdem empfinden die Nutzer der Agilen Softwareentwicklung die akribische Dokumentation des Schaffensprozesses als Behinderung der Arbeit und bauen lieber auf „funktionierende Software“. [2]

2.6 Extreme Programming

Ein Beispiel eines agilen Modells ist Extreme Programming (auch als „XP“ genannt). XP wurde von Kent Beck (siehe „Agiles Manifest“) entwickelt. Ihm war beim Entwickeln mit dem Wasserfallmodell aufgefallen, dass die Kunden am Anfang des Projektes die Anforderungen noch nicht genau wussten und dadurch die Entwickler vor massive Probleme gestellt wurden, weil die Anpassung der Programme schwierig im langen SDLC umzusetzten war. Beck kam auf die Idee die Laufzeit des SDLC zu verkürzen - also den SDLC auf jede Entwicklung einzeln anzuwenden, anstatt auf das ganze Programm - und statt wenigen bzw. einer Wiederholung mehrere Wiederholungen des SDLC ablaufen zu lassen.

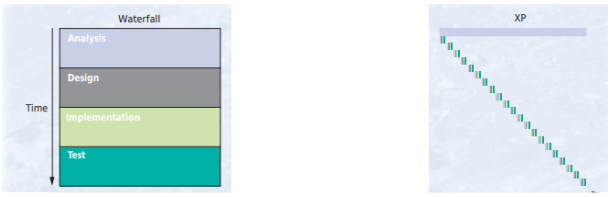

Abbildung 2: Vergleich des SDLC im Wasserfallmodell (links) zum XP (rechts) aus [4]

Bei XP soll keine weitreichende Zukunftsplanung betrieben werden, da sich die Anforderungen jederzeit ändern können. Stattdessen soll alles Schritt für Schritt geplant und ausgeführt werden, d. h. erst einen Teil des Projektes abschließen und dann den nächsten planen und diesen ausarbeiten.

Zudem hat Beck viele Arbeitstechniken benannt, die XP ausmachen.

Kent zählt hierbei zum Beispiel die Methode des „Planning Game“ [4]. Diese Praktik beinhaltet, dass der Kunde den Umfang und Zeitpunkt der Lieferung von Prototypen bzw. Ergebnissen auf der Grundlage der Schätzung der Programmierer bestimmt. Außerdem bedeutet dies, das die Programmierer nur das implementieren, was explizit vom Kunde gewünscht wird (siehe Prozesse zur Erleichterung des Programmierens).

Zu den vielfältigen Praktiken des XP gehört auch „Simple Design“ [4]. Dies bedeutet, dass der Code so einfach wie möglich gehalten werden sollte. Das heißt:

- kein doppelter Code,
- so wenige Klassen wie möglich und
- so wenige Methoden wie möglich.

Kent Beck nennt hier den Leitspruch „Say everything once and only once. (dt.: Sage alles einmal, nur einmal.)“[4] (siehe dazu auch Prozesse zur Erleichterung des Programmierens).

Außerdem zählt Kent Beck zu den Techniken des XP „Tests“ [4]. In XP sollten alle Proramme ausgiebig getestet werden, um Fehler immer schnell bzw. sofort zu finden.

Zudem ist eine wichtige Technik von XP das „Pair Programming“ [4] (siehe dazu Paarprogrammieren).

Ein weiteres Prinzip ist „Continous integration“ [4]. Hierbei wird neuer Code beim schreiben schon in den vorhandenen Code eingefügt und auch darin getestet. Wenn

dieser Code nicht im gesamten Umfeld funktioniert, wird er verworfen.

Das Prinzip „Collective Ownership“ [4] besagt, dass jeder Programmierer jederzeit den Code verbessern kann, wenn er eine Verbesserungsmöglichkeit entdeckt.

Außerdem ist beim Entwicklungsprozess immer ein Kunde beim Team, der „On-side costumer“ [4].

Zudem wird in XP eine „40-hour week“ [4] empfohlen, weil sich zu viel Arbeit negativ auf die Arbeitsmoral auswirkt. Kent meint dabei, dass niemand eine zweite Überstundenwoche in Folge leisten sollte („No one can work a second consecutive week of overtime“ [4]). Dadurch soll die Arbeitsmoral gesteigert werden.

Die Entwicklungsarbeit findet in einem „Open workspace“ [4] statt. Das Team arbeitet in einem großen Raum mit kleinen Abgrenzungen an den Seiten mit einzelnen Programmierern. In der Mitte des Raumes sollten die Paarprogrammierer sitzen.

Das letzte Prinzip, was von Kent Beck genannt wird, ist das „Just rules“-Prinzip [4]. Dies besagt, dass alle Entwickler die festgelegten Regeln beim Beitritt ins Team akzeptieren, die Regeln aber flexibel geändert werden können, wenn alle Teammitglieder damit einverstanden sind.

Zusammmenfassend lässt sich sagen, dass Extreme Programming ein gutes Beispiel für ein agiles Softwareentwicklungsmodell darstellt, da XP eine hohe Agilität - sowohl durch die ofte Wiederholung des SDLCs aber auch den angesprochenen Techniken, wie zum Beispiel dem Paarprogrammieren oder auch dem On-side costumer - aufweist.

3 Vergleich klassischer bzw. statischer und agiler Softwareentwicklung

Nun wurden sowohl die tradiotionelle bzw. statische Softwareentwicklung, als auch die agile Softwareentwicklung, genau untersucht. Im Folgenden werden die beiden Modelle verglichen und geschildert, für welche Projekte und Gegebenheiten sich die jeweiligen Softwareentwicklungsmodelle eignen.

Zuerst wird die Kommunikation in den einzelnen Modellen analysiert.
Die klassische Softwareentwicklung sieht eine formale Sprache zwischen den Projektbeteiligten vor, sodass eine Kommunikation auf formaler Ebene entsteht.
Beim agilen Modell hingegen, empfiehlt sich eine ungezwungene und offene Kommunikation um die Zwischenmenschlichkeit zu steigern und ein besseres Arbeitsverhältnis zu erzeugen.

Die Zielgruppe der Agilen Softwareentwicklung sind Programmierer, welche schon vie-

le Erfahrungen, auch mit agiler Softwareentwicklung, gesammelt haben und somit das Projekt gut leiten können und alles „im Auge“ haben, um auch bei kurzfristigen Änderungen der Anforderungen flexibel sein zu können.
Die klassische Softwareentwicklung ist hingegen auch für Anfänger geeignet, da alle Abläufe nach dem SDLC klar geregelt sind, so dass jeder sie einfach verstehen und ausführen kann.

Wie gerade angedeutet, gibt es in der klassischen Softwareentwicklung nach dem SDLC eine klar festgelegte Hierachie mit klarem „Herrschaftsverhältnis“, was bedeutet, dass die Rollen klar verteilt sind und der Vorgesetzte „das Sagen“ hat.
Zudem ist das Verhältnis zum Kunden bei der statsichen Softwareentwicklung sehr eingeschränkt. Der Kunde benötigt klare Vorstellungen von dem Produkt, was er bekommen möchte, d. h. er muss schon im Voraus sehr genau wissen, was das Produkt können soll oder nicht, da auf Veränderungen im SDLC nicht gut und schnell eingegangen werden kann, da die Phasen nacheinander ablaufen und mit wechselnden Änderungen neu begonnen werden müssten.
Hingegen ist die Beziehung zum Kunden bei der agilen Softwareentwicklung flexibel. Das bedeutet, dass der Kunde seine Anforderungen ändern kann und das Entwicklerteam daruf agil reagieren kann, da das Produkt immer Schritt für Schritt weiterentwickelt wird, sodass Änderungen schnell eingebaut werden können.

Das agile Modell beruht jedoch auf gleichgestellter Zusammenarbeit der Projektbeteiligten. Es soll gewährleistet sein, dass alle Teammitglieder auf der gleichen Ebene stehen und gleich wichtig für das Gelingen des Projektes sind.

Es wurden die zwischenmenschlichen Beziehungen in den jeweiligen Modellen betrachtet, jetzt folgen die informatischen Unterschiede der Modelle.

Durch die bereits erwähnte Agilität halten sich auch die Kosten für einen Neustart gering, da die einzelnen Programmteile immer noch verwenden können und nicht „von Null“ erneut begonnen werden muss.
Dagegen geht dies bei der statischen Softwareentwicklung nicht, da der SDLC neugestartet werden und alle Phasen erneut durchlaufen werden müssen. Dadurch steigen auch die Kosten immens an.

Aus den nun verglichenen Attributen der verschiedenen Softwareentwicklungsmodelle geht hervor, dass sich die agile und statische Softwareentwicklung maßgeblich unterscheiden, da die Zielsetzung der Modelle für unterschiedliche Zwecke gedacht ist.
In der statischen Softwareentwicklung hat die Stabilität des Entwicklungsprozesses und des Produktes die höchste Priorität, dafür wird aber die Flexibilität vernachlässigt. Das führt auch dazu, dass dieses Modell gut für große Projekte von großen Firmen mit vielen Mitarbeitern geeignet ist, da es wichtig ist eine geordnete Struktur bei vielen Aufgaben und Mitarbeitern zu besitzen.
Bei der agilen Softwareentwicklung hat die Agilität bzw. Flexibilität den größten Stellenwert, dafür ist die Entwicklung nicht so stabil wie beim klassischen Ansatz, da mit

vielen Teammitgliedern die Koordination schwierig ausfallen kann. Daraus lässt sich ableiten, dass ein agiler Ansatz eher für kleine Projekte mit einem kleinen Team geeignet ist, da so Agilität, als auch Stabilität hergestellt werden kann, weil es wenige Teammitglieder zu koordnieren gibt. Dabei steht der Kunde im Vordergrund und es kann von den Entwicklern aktiv auch auf größere Wünsche eingegangen werden.

4 Prozesse zur Erleichterung des Programmierens

4.1 Motivation

Es gibt natürlich sehr viele verschiedene Varianten und Modelle eine Anwendung zu programmieren (siehe Kapitel 1 und 2), dabei sollten aber einige Dinge beachtet werden, um das Programm möglichst ansprechend für den Kunden und einfach prorammierbar zu gestalten. Außerdem kann so die Qualität des Programmes erheblich erhöht werden.

4.2 Vermeidung von Überflüssigem

Hierbei gibt es im Wesentlichen 3 Methoden zu nennen, mit welchen überflüssige Programmteile vermieden werden können.

4.2.1 KISS

Die Abkürzung „KISS“ steht für „Keep it simple, stupid“, was übersetzt so viel heißt wie „Mach es einfach, Dummkopf“.
Der Name dieses Prinzips stammt von Kelly Johnson, einem englischen Ingenieur, der diesen Ausdruck im Jahr 1960 gesagt haben soll. [5]
Nach dem Prinzip KISS wird gehandelt, wenn das System möglichst einfach aufgebaut ist und wenig komplex ist. Durch erhöhte Komplexität, können spätere Änderungen und Erweiterungen nicht durchgeführt werden, ohne "die Stabiblität des Systems zu gefährden" [5].

Da KISS Einfachheit fordert, ergeben sich viele Vorteile.
Natürlich ist es einfacher ein einfaches, weniger komplexes System „zu verstehen, zu bauen, zu testen, zu ändern und zu warten“ [5].
Außerdem können weniger Fehler im System auftreten, da es auch weniger Komponenten gibt, welche eben diese aufweisen können.
Ein Nachteil von KISS ist es, dass einfache Systeme, meist schwieriger zu programmieren sind, da komplexe Lösungen meist einfacher zu implementieren sind.
Zudem sollte beachtet werden, dass KISS zwar beinhaltet, einfache Systeme zu bauen, jedoch keine monolithischen, welche ungegliedert sind und somit die Arbeit erschweren. [5]

4.2.2 YAGNI

„YAGNI“ kommt aus dem Englischen und steht für „You aren't gonna need it“, was übersetzt so viel bedeutet, wie „Du wirst es nicht brauchen.“
Diese Technik wurde maßgeblich von Ron Jeffries[2] geprägt und wird deshalb auch oft mit der agilen Softwareentwicklung in Verbindung gebracht (siehe Die Agile Softwareentwicklung) [6]
Generell soll mit diesem Prinzip „Over-Engineering“ [5] und eine „spekulative Generalisierung“ [5] vom Entwickler vermieden werden.
Als Over-Engineering wird die Entwicklung eines Produktes höherer Qualität und größerem Aufwand, als der Kunde gefordert hat, bezeichnet. Somit wird meist der Preisrahmen des Kunden überschritten.
Spekulative Generalisierung bedeutet, dass der Entwickler Abstraktionen einführt, die er denkt, noch gebrauchen zu können, obwohl diese meist später nicht mehr benötigt werden. „Generalisierungen verzögern [außerdem] das ursprüngliche Projekt, erschweren die Ausbaubarkeit und kosten Geld“ [5]. Auch Generalisierungen sollten vom Kunden gewünscht und bezahlt werden. Sonst ist es empfehlenswert diese wegzulassen.

Durch YAGNI ergeben sich viele Vorteile für den Entwickler und auch für den Kunden. Zum Einen ist der geschriebene Code einfach weiterentwickelbar, weil er im besten Fall wenig komplex und übersichtlich ist und außerdem nicht durch überflüssige Funktionalität behindert wird.
Durch generalisierte Feautures im Produkt können unnötige Einschränkungen auftreten, welche sich aber durch YAGNI verhindern lassen.
Zudem hat der Entwickler mehr Zeit sich auf die geforderten Funktionen des Kunden zu konzentrieren, weil er die Zeit, die er durch verzichtbare Generalisierung vergeudet, sinvoll nutzen kann.

Jedoch kann es bei einer falschen Anwendung von YAGNI auch zu einem Problem kommen. Denn wenn sich der Entwickler nur auf die Anwendungsfunktionen des Programmes fokussiert, kann es dazu kommen, dass die Weiterentwickelbarkeit des Programmes nicht berücksichtigt wird. Das ist falsch, denn auch wenn der Kunde die Infrastruktur des Produktes nicht kennt, sollte auch diese, nicht übertrieben, beachtet werden. [5]

4.2.3 DRY

„DRY“ kommt aus dem Englischen und steht für „Don't repeat yourself“, was auf Deutsch übersetzt soviel wie „Wiederhole dich nicht“ bedeutet.
Diese Abkürzung stammt von Andrew Hunt und David Thomas aus ihrem Buch „Der Pragmatische Programmierer“ [7] wurde aber keineswegs von diesen Personen erfunden. Davor war es als „Single-Source-Prinzip“ bekannt. [5]
Beim Programmieren können Redundanzen entstehen. Redundanzen oder auch Re-

[2] Ron Jeffries war einer der Initiatoren der agilen Softwareentwicklung und hat u. A. das „Agile Manifest“ mitentwickelt.

plikate sind Programmteile, welche identisch sind. Diese treten zum Beispiel durch Unaufmerksamkeiten, Zeitnot und fehlender Absprache zwischen den Teammitgliedern auf.
Diese Redundanzen sollen mit dem DRY-Prinzip verhindert werden, denn Replikate bringen oftmal Schwierigkeiten, wie zum Beispiel das Erzeugen von inkonsistenten Informationen bei Aktualisierung des Programmes, mit sich.
Deshalb sollten in einem Programm, in welchem dieses Prinzip beachtet wurde, alle Informationen des Projektes lediglich einmal vorkommen.

Durch DRY werden „Aktualisierungsprobleme vermieden (siehe oben) und dadurch die Komplexität verringert“ [5].
Außerdem wird der Code durch DRY besser lesbar, weil schlichtweg weniger Code vorhanden ist.
Jedoch kann es auch passieren, dass bei manchen Programmen Redundanzen vorhanden sein müssen, welche dann fälschlicherweise durch DRY entfernt wurden.

4.2.4 Zusammenfassung

Die drei vorgestellten Methoden nehmen keinen Einfluss auf die Struktur bzw. Architektur des Programmes, sondern sollten vom Entwickler beim Programmieren beachtet werden, damit die Programme besser verständlich und stabiler werden und zudem wechselseitige Abhängigkeiten abgeschwächt werden können. Somit kann die Komplexität des Produktes reduziert werden. [5]
Im nachfolgenden Abschnitt werden die Prinzipien zur Verringerung der Komplexität vorgestellt, welche auf Systemebene wirken.

4.3 Prozesse auf Systemebene

4.3.1 Divide And Conquer

„Divide and conquer“ oder zu Deutsch „Teile und Herrsche“ bezeichnet ein viel in der Informatik eingesetztes Prinzip.
„Teile und Herrsche“ funktioniert, indem ein großes Problem rekursiv zerlegt wird, so dass viele kleine Teilprobleme enstehen, welche dann einfacher gelöst werden können. Die Teilprobleme lassen sich dann am Ende zusammensetzten und bilden somit das Programm.
Somit kann man zum Beispiel Algorithmen entwerfen und zum Beispiel ein Programm in viele Module zerlegen.
Ziel von „Divide And Conquer“ ist es, ein schweres, komlexes Ausgangsproblem von oben nach unten „in kleinere, möglichst unabhängige Teilprobleme“ [5] zu zerlgen, um diese dann möglichst einfach zu lösen.
Auch „Teile und Herrsche“ bringt Vorteile, als auch Nachteile, mit sich.
So wird aus einem ursprünglich sehr komlexen Problem, ein Einfaches gemacht, wodurch die Komplexität verloren geht.
Außerdem ist es ein Vorteil, dass viele kleine Probleme gelöst werden, denn wenn bei

diesen ein Fehler auftritt, kann dieser leicht behoben werden und es entstehen keine großen Probleme.
Jedoch ist der Top-Down-Ansatz[3] nicht immer passend. Das bedeutet, „wenn eine untere Ebene, beispielsweise Bibliotheken, vorgegeben sind: Dann muss die Zerlegung so erfolgen,dass man diese Ebene auch „trifft“.“ [5]

4.3.2 Design to Test

Ein wichtiger Teil des Programmiers ist auch das Testen (siehe Kapitel 1.2). Jedoch kommt dieser Entwicklungsabschnitt meist zu kurz, da die Performance des Produktes eher im Vordergrund steht. Somit ist das Produkt am Ende, dass sich der Gesamttest ungenügend automatisieren lässt. Somit steigen auch die Kosten sowohl beim Gesamtsystemtest, als auch bei der Wartung.
Ein Gesamtsystemtest ist leichter durchzuführen, wenn alle Komponenten einzeln (siehe „Divide And Conquer“) getestet werden können. Dies stellt auch das Ziel des Prinzips „Design to Test“ dar. Bei dieser Methode soll ein Programmteil so entworfen werden, dass er einfach zu testen ist. Wenn dies nicht der Fall ist sollte der Programmteil verworfen werden und eine gut testbare Lösung gefunden werden.
Die gute Testbarkeit eines Programmes bringt Vorteile mit sich.
Zum Beispiel können alle Tests automatisiert ablaufen.
Außerdem können alle Programmteile besser getestet werden, dadurch senken sich die Kosten für das Testen.
Jedoch ist es ein Nachteil, wenn erst nach dem Entwurf feststellt wird, dass dieser schlecht testbar ist, da somit die Entwicklungskosten durch den Neuentwurf ansteigen. Deshalb sollte auch beim Planen, sowie bei allen Schritten der Produktentwicklung, auch die Testbarkeit im Vordergrund stehen, so dass hohe Kosten durch Neuentwürfe vermieden werden können. [5]

4.4 Bildquellen

`http://www.tutorialspoint.com/sdlc/sdlc_overview.htm`

[3] Zerlegung des Problems von oben nach unten

Literatur

[1] Marian Stoica, Marinela Mircea, and Bogdan Ghilic-Micu. Software Development: Agile vs. Traditional. 17(4/2013):64–76, December 2013.

[2] Kent M. Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andy Hunt, Ron Jeffries, Jon Kern, Brian Marick, R. C. Martin, Steve J. Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development. 2013.

[3] L. Williams, R.R. Kessler, W. Cunningham, and R. Jeffries. Strengthening the case for pair programming. *IEEE Software*, 17(4):19–25, August 2000.

[4] K. Beck. Embracing change with extreme programming. *Computer*, 32(10):70–77, October 1999.

[5] Joachim Goll. *Entwurfsprinzipien und Konstruktionskonzepte der Softwaretechnik: Strategien für schwach gekoppelte, korrekte und stabile Software*. Springer Vieweg, Wiesbaden, 2018. OCLC: 1022114526.

[6] Ron Jeffries, Ann Anderson, and Chet Hendrickson. *Extreme Programming Installed*. Addison-Wesley, 2000.

[7] Andrew Hunt and David Thomas. *Der Pragmatische Programmierer*. Fachbuchverlag Leipzig im Carl Hanser Verlag, 2003.