Jim Highsmith

Adaptive Software Development

Seminararbeit im Rahmen des Seminars
Agile vs. klassische Entwicklungsmethoden

Vorgelegt von
Christoph Eberle
Zürich

Universität Zürich
Wirtschaftswissenschaftliche Fakultät
Institut für Informatik
Prof. Dr. Martin Glinz
Betreuer Christian Seybold
11.11.2003

Inhaltsverzeichnis

1. Einleitung

In Jim Highsmith's Buch „Adaptive Software Development" geht es um die Anforderungen, einer sich immer schneller verändernden Welt, an die Softwareentwicklung und wie man den daraus entstehenden Problemen begegnen könnte. Highsmith hat nicht die Absicht, eine Patentlösung, eine so genannte „Silver Bullet", zu entwickeln, sondern viel mehr, bereits vorhandenes Wissen zu bündeln und zu einem neuen Rahmenwerk zusammenzustellen. So mag einem auf den ersten Blick vieles sehr bekannt vorkommen und einem zur Frage verleiten: „Wissen wir das nicht alles schon?"
Highsmith gibt am Ende seiner 320 Seiten sogar selber zu, dass er eigentlich wenig Neues beigesteuert hat und es ihm vielmehr um die kombinierte Sichtweise verschiedenster moderner Techniken geht.
Meiner Meinung kann es auf jeden Fall nichts schaden, sich diese Maximen und Paradigmen immer wieder in Erinnerung zu rufen, um sie auch im Projektalltag richtig einsetzen zu können.

2. Jim Highsmith

James A. Highsmith III hält einen Bachelor in Elektrotechnik und einen Master in Betriebswirtschaft. Seine professionelle Karriere startete er als Softwareprogrammierer im Rahmen des Apollo Raumfahrtsprojektes. Später wechselte er in die Consultingbranche und wirkt seither als Berater verschiedenster Grossfirmen und als Autor diverser Artikel zum Thema Projektmanagement. Er ist verheiratet hat zwei Töchter und sein grösstes Hobby ist Bergsteigen, was er auch immer wieder anhand kleiner Annekdoten und Analogien zur Softwareentwicklung im Buche zum Ausdruck bringt.

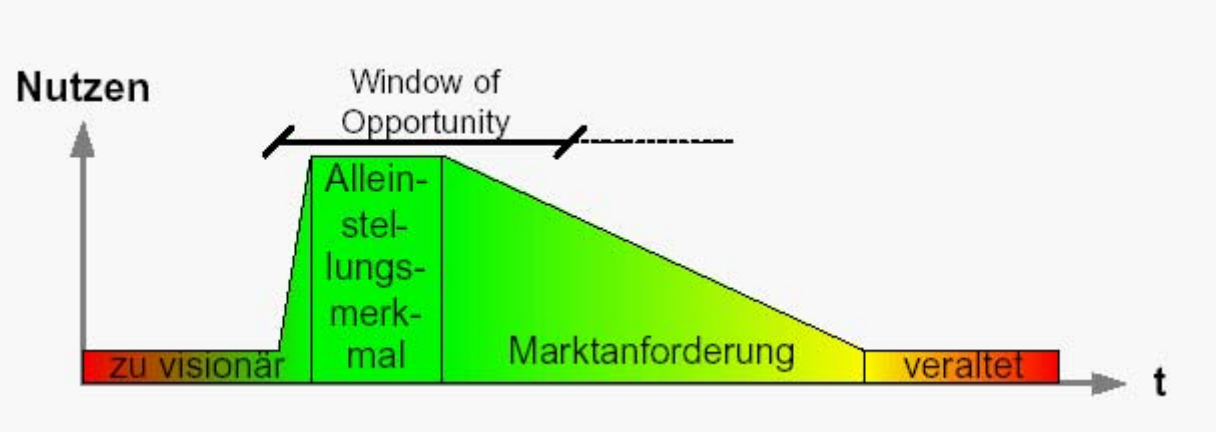
3. Die Softwarefalle

Die Anfänge der Softwareentwicklung waren geprägt von chaotischen Tätigkeiten, so genanntem „Code and fix". Die Software wurde fast ohne zugrunde liegenden Plan geschrieben und das Design des Systems wurde aus vielen kurzfristigen Entscheidungen zusammengesetzt. Das funktionierte solange ziemlich gut, solange das System klein blieb, mit zunehmendem Wachstum aber, wurde es immer schwieriger, dem System neue Features hinzuzufügen. Ausserdem verbreiteten sich immer mehr Bugs, die immer schwieriger zu beheben wurden. Ein typisches Merkmal eines solchen Systems war eine lange Testphase, nachdem das System eigentlich als „funktional vollständig" galt. Diese lange Testphase warf dann alle Zeitpläne über den Haufen, weil es unmöglich war, Testen und Debuggen zeitlich zu planen. Eine Studie von IBM in den 70er Jahren zeigte schliesslich auf, dass die Kosten eines Fehlers im Laufe der Zeit exponentiell ansteigen. Es galt deshalb als oberste Maxime, Fehler um jeden Preis zu vermeiden.

Bald schon kamen deshalb Methodologien oder Vorgehensweisen auf, mit dem Ziel die Softwareentwicklung zu einem disziplinierten Prozess, und damit vorhersagbarer und effizienter zu machen. Dabei lag, in Anlehnung an andere Ingenieursdisziplinen, eine starke Betonung auf der Planung der Prozesse und auf Dokumentation.

Bekannte Beispiele dieser Methodologien sind: „Wasserfallmodell" und „strukturierte Verfeinerung"

Diese Methodologien gibt es, wie erwähnt, schon lange. Bis jetzt sind sie jedoch nicht dafür bekannt, schrecklich erfolgreich zu sein. Noch weniger sind sie bekannt dafür, beliebt zu sein. Die häufigste Kritik an diesen Methodologien ist, sie seien bürokratisch. Der Aufwand für Nebentätigkeiten ist dabei so gross, dass die Entwicklungsgeschwindigkeit insgesamt langsamer wird. Daher werden sie oft auch als schwere Methodologien bezeichnet, oder um einen Begriff von Jim Highsmith zu gebrauchen, als „monumentale“ Methodologien.
Neben dem Unbehagen bei den Mitarbeitern stellt sich aber ein noch viel grösseres Problem bei den monumentalen Methodologien, sie sind zu langsam. Durch die grosse Redundanz in der Dokumentation wird das Projekt durch jedes einzelne Softwaremodul weiteraufgeblasen und es droht die Gefahr, dass bei Fertigstellung der Software, die Anforderungen der Auftraggeber bereits andere sind, als ursprünglich angenommen. Diese Tatsache hat sich im Laufe der 90er Jahre zunehmend verstärkt, nicht zuletzt durch Aufkommen von neuen Technologien wie World Wide Web und E-mail, aber auch durch neue Produktionsmethoden wie „Just in Time“ und „virtuellen Unternehmen“.
Es gilt deshalb neue Definitionen zu finden, was denn überhaupt als korrekte Software betrachtet werden kann. Wir wollen hier deshalb den Begriff der „wirtschaftlich korrekten Software“ einführen. Dabei kann eine Software nur innerhalb einer gewissen Zeitspanne als korrekt gelten. Es nützt also nichts, eine vollständig korrekte, vielleicht sogar mathematisch bewiesene, Software ein halbes Jahr zu spät auf den Markt zu bringen. Dies soll die untendargestellte Grafik[1] verdeutlichen:

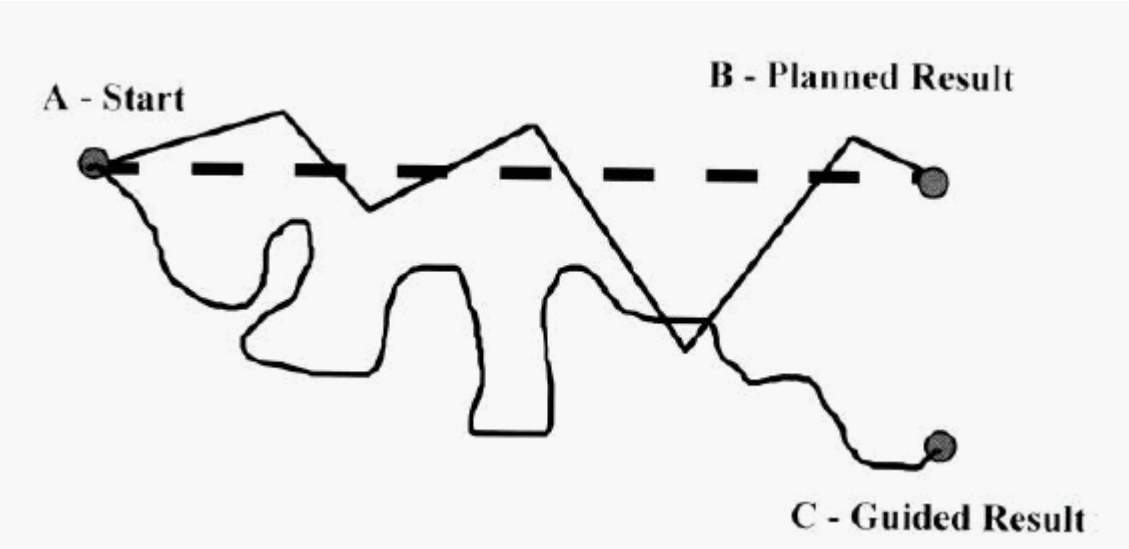


Das bekannteste Beispiel für eine Firma, die diese Tatsache erkannt hat, dürfe (zum Ärger vieler) Anwender Microsoft sein. Man fragt sich nun aber, wie schafft es diese Firma so erfolgreich zu sein. Es handelt sich ja immerhin um Programme im 1’000'000 Codezeilenbereich, wobei ein Adhocverfahren sicher ausscheiden muss. Die Antwort lautet „Agile Software Entwicklung“.

[1] Coldeway Consulting 2001

4. Agile Software Entwicklung

Die immer schneller ändernden Anforderungen machen neue Methodologien notwendig. Es macht keinen Sinn mehr im Voraus einen Weg von A nach B zu planen, wenn man mitten im Projekt merkt, dass eigentlich C das richtige Ziel wäre.[2]



Ein wichtiger Unterschied der agilen Methoden, ist ihre Reaktion auf die Bürokratie der monumentalen Methodologien, sie versuchen, einen brauchbaren Kompromiss zwischen gar keiner Ordnung und zuviel Vorschriften zu finden, indem sie eben gerade genug Vorschriften vorsehen, um einen vernünftigen Erfolg zu erzielen.

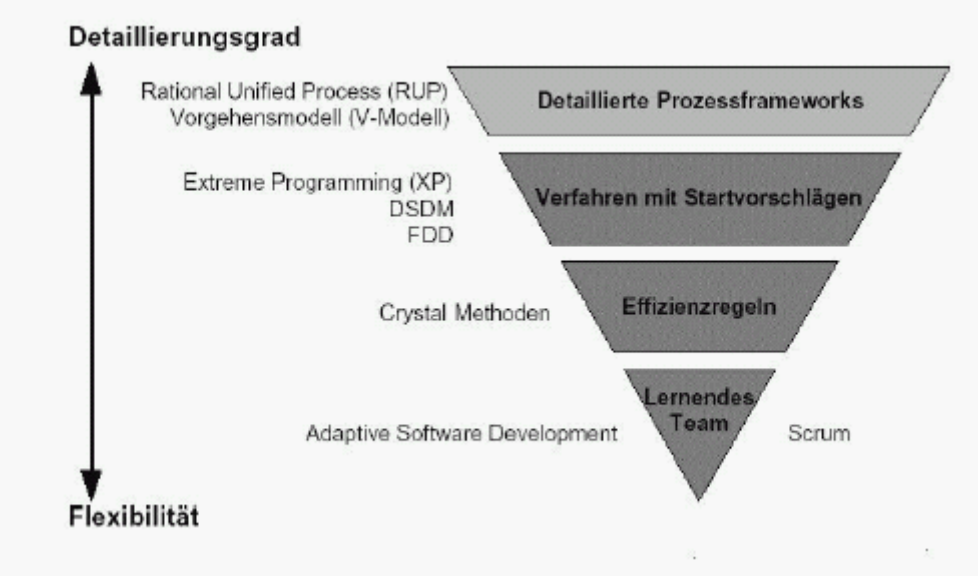
Die agilen Methoden verlagern deshalb die Interessenschwerpunkte gegenüber den schwergewichtigen Methoden. Der unmittelbarste Unterschied ist, dass sie weniger dokumentenorientiert sind und üblicherweise eine geringere Menge Dokumentation für einen bestimmten Aufgabenschritt verlangen. In vieler Hinsicht sind sie recht codeorientiert: Sie verfolgen eine Linie, die besagt, dass der entscheidende Teil der Dokumentation der Sourcecode ist.

Allerdings ist das nicht das Entscheidende an agilen Methoden. Verzicht auf Dokumentation ist nur ein Symptom einer vollkommen anderen Denkhaltung:
Agile Methoden sind adaptiv und nicht vorhersagend. Schwere Methoden neigen dazu, einen grossen Teil des Softwareentwicklungsprozesses im Detail zu planen. Das funktioniert genau solange gut, bis sich etwas ändert. Daher ist es ihre Natur, Änderungen Widerstand entgegen zu setzen. Die agilen Methoden jedoch heissen Änderungen willkommen. Sie versuchen, sich anpassende Prozesse zu sein, die im Wandel erst richtig aufblühen.
Agile Methoden sind auf Menschen ausgerichtet, nicht auf Prozesse. Sie betonen ausdrücklich, im Einklang mit der menschlichen Natur zu arbeiten und nicht gegen

[2] James A. Highsmith III: *Adaptive Software Development - A Collaborative Approach to Managing Complex Systems*, Seite 43

sie. Ein weiteres Merkmal ist, dass Softwareentwicklung eine Tätigkeit sein sollte, die Spass macht.

Die agilen Methodologien untereinander unterscheiden sich ganz erheblich in Umfang und Detaillierungstiefe. „ASD“, „Scrum“ und „Crystal Methods“ können eher als Meta-Prozesse angesehen werden. Sie geben nur Rahmenbedingungen, anhand derer ein erfahrenes Team einen konkreten Prozess entwickeln kann, verzichten aber auf genaue Vorgaben für die tägliche Arbeit.
„XP“, „DSDM“ und „FDD“ sind im Vergleich dazu konkrete Verfahren. Sie liefern detaillierte Startvorschläge für das Projekt, z.B. die Gestaltung der Iterationszyklen, die Beziehung zum Kunden oder das Vorgehen bei Implementierung und Testen. Folgende Grafik soll dies verdeutlichen.[3]



Allen agilen Methodologien gemein ist aber die Philosophie dahinter. So kamen im Februar 2001 verschiedene Vertreter zusammen und einigten sich auf folgendes, sehr allgemeingehaltenes Manifest:

Manifesto for Agile Software Development
We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
Individuals and interactions over **processes and tools**
Working software over **comprehensive documentation**
Customer collaboration over **contract negotiation**
Responding to change over **following a plan**
That is, while there is value in the items on the right, we value the items on the left more.[4]

[3] Jens Coldewey. *Einführung in Agile Entwicklung*. www.coldewey.com, 2002.

[4] www.agilemanifesto.org.

5. Adaptive Software Development

5.1 Was Adaptive Software Development *nicht* ist

Wenn man von neuen Softwareentwicklungsmethoden hört, dann denkt man oft an irgendwelche unkontrollierten Herangehensweisen, wo man einfach darauflosprogrammiert ohne ein wirkliches Konzept vor Augen zu haben. Diese, auch adhoc genannten, Verfahren kamen in den 90er Jahren zusammen mit der Verbreitung der Personal Computer auf. Als Extremfall, kann man sich zum Beispiel ein, von einer Sekretärin geschriebenes, Excelmakro vorstellen, welches im Laufe der Zeit zu einer vollständigen Workflowlösung ausgebaut wird. Dass hier die Wartbarkeit und Skalierbarkeit auf der Strecke bleibt, ist offensichtlich.
Deshalb möchte ich damit beginnen, was Adaptive Software Development *nicht* ist.

ASD produziert *nicht* kurzfristige Wegwerfsoftware
ASD bedeutet *nicht* Anarchie in der Unternehmung
ASD verzichtet *nicht* auf Planung
ASD verzichtet *nicht* auf Entwicklungswerkzeuge

5.2 Komplexe Adaptive Systeme

Highsmith beginnt sein Buch mit einer Einführung in komplexe adaptive Systeme und versucht daraus, Schlussfolgerungen auf die Softwareentwicklung abzuleiten. Seiner Meinung nach ist die Softwareentwicklung zu stark vom darwinistischen Gedanken des „survival of the fittest“ geprägt, was zur Folge hat, dass Software fortlaufend optimiert wird. Dabei wird das Unternehmen als grosse Maschine betrachtet, welche aus tausenden kleinen Zahnrädchen, den Mitarbeitern, besteht, und jeden Tag versucht durch „Business Process Reengineering“ noch effizienter zu werden. Highsmith nennt diese Betrachtungsweise „a world of decreasing returns“, um darauf eine neue Welt, nämlich die der „increasing returns“ auszurufen. Er prägt auch noch gleich einen weiteren neuen Begriff, um Darwin entgegenzutreten, den des „arrival of the fittest“. Doch was meint Highsmith damit?
Um diese Frage zu beantworten zu können, müssen wir uns zuerst einmal in die Entstehungszeit des Buches versetzen, nämlich ins Frühjahr 2000; in eine Zeit, wo eine Firma, die in einer Garage einen Webserver betrieb, eine höhere Börsenkapitalisierung erreichte als General Motors.
Vor diesem Hintergrund mag man eine gewisse Euphorie verstehen, aber heute wissen wir, dass die Gesetzte der „alten“ Ökonomie auch weiterhin gelten. Trotzdem wollen wir weiteruntersuchen, was denn Highsmith sich unter dieser neuen Welt vorstellt.

Für Highsmith stellt ein Softwareentwicklungsteam ein komplexes Gebilde dar, in dem mehrere autonome Agenten zusammen eine Lösung entwickeln. Man mag nun daraus folgern, dass komplexe Systeme auch komplexe Regeln brauchen, um funktionieren zu können. Doch weit gefehlt; die wirklich innovativen Firmen, wie Microsoft, benutzen keinesfalls viele komplexe Regeln um ihre Agenten zu koordinieren. Vielmehr genügen wenige einfache Grundregeln, so genannte Paradigmen. Dabei entsteht die Software, wie zufällig. Highsmith nennt dieses Phänomen „Emergenz“.

5.3 Emergenz

Highsmith betrachtet Emergenz als eines der wichtigsten Merkmale adaptiver Softwareentwicklung. Darunter versteht er die Eigenschaft, dass das Gesamte mehr erarbeitet, als seine einzelnen Teile. Er setzt Emergenz gleich mit Innovation, Problemlösen und Kreativität. Wir alle kennen dieses Phänomen, wenn gewisse Dinge ganz plötzlich einfach von der Hand gehen, wenn ein Fussballteam plötzlich eine herausragende Mannschaftsleistung vollbringt, oder wenn wir in einer Prüfung die Antworten einfach hinschreiben können. Leider ist Emergenz aber schwierig zu messen, geschweige denn zu planen, was zur Folge hat, dass wir sie in unseren Überlegungen oft vernachlässigen. Es ist deshalb eine grosse Herausforderungen für den Teamleader, diesen Zustand zu erreichen. Highsmith kann hier natürlich auch kein Patentrezept liefern, er sagt nur soviel: „Ein guter Manager leitet sein Team am Rande des Chaos zum Ziel.“ Das heisst soviel, dass der Entwicklungsprozess, obwohl gewissen Regeln folgend, immer eine gewisse Spannung und Ungewissheit beinhalten muss, um Dynamik und Kreativität aufkommen zu lassen.

6. Adaptive Development Framework

Highsmith entwickelt im Laufe des Buches ein dreiteiliges Rahmenwerk um ein adaptives Softwareprojekt erfolgreich durchzuführen.
Er unterscheidet zwischen:

- Adaptive Conceptual Model

- Adaptive Development Model

- Adaptive (Leadership-Collaboration) Management Model

6.1 Adaptive Conceptual Model

Im Rahmen des Adaptive Conceptual Models bildet Highsmith die Grundlagen, um ein adaptives Softwareprojekt konzeptuell erfassen zu können. Er nennt insbesondere zwei wichtige Grundpfeiler auf die sich ein Projektteam stützen sollte: Die Projekt Mission und den Adaptive Development Life Cycle.

6.1.1. Die Projekt Mission

Gute Projektmissionen fallen nicht einfach vom Himmel. Obwohl in vielen Projekten die Mitglieder der Meinung sind, dass sie eine gewisse Mission erfüllen, kann dies nicht gleichgesetzt werden mit dem, was man in ASD als Mission versteht. Eine Mission sind nicht leere Worte auf einem Flipchart, eine gute Mission ist eine gemeinsame Passion. Dabei sind einige Dinge zu beachten. Um eine brauchbare Software zu entwickeln, muss die Mission muss relativ spezifisch auf konkrete Fragen Antwort geben; gleichzeitig darf sie aber nicht zu eng gefasst sein, um nicht die Kreativität zu erdrücken. Ein weiterer wichtiger Bestandteil der Mission ist das Umgehen mit Änderungen. Es muss ganz klar definiert werden, wann Entscheidungen gefällt werden, und damit Kompromisse in Kauf genommen werden, da man in adaptiven Projekten keine Zeit verlieren darf. Trotzdem geht es in einer Mission nicht um das Festlegen von einzelnen Regeln im „wenn, dann“-Stil, sondern viel eher um das Verständnis von „Patterns“, also Denkmustern, die es jedem einzelnen erlauben seinen eigenen Weg zu beschreiten um seine Aufgaben zu erfüllen.

Gute Projekte werden oft daran gemessen, ob sie die Kriterien Umfang, Zeit, Budget und Qualität einhalten. Highsmith bezeichnet dies als „pure rubbish“. Er ist der Meinung, dass man bei einem grösseren Projekt, und insbesondere bei einem adaptiven seinen Fokus auf eines der Merkmale richten sollte. Natürlich dürfen die übrigen Kriterien nicht ganz vernachlässigt werden, aber um Kompromisse („Trade-Offs“) eingehen zu können, muss man wissen, welches Kriterium als Topppriorität gilt.

Anschliessend gilt es drei Fragen zu beantworten:

Um was geht es bei diesem Projekt?

Das Ziel muss sein, dass jeder Beteiligte innerhalb von fünf Sätzen eine businessorientierte, nichttechnische Antwort geben kann, um was es bei diesem Projekt geht.

Warum sollen wir das Projekt durchführen?

Diese Frage ist extrem wichtig, weil etwa 60% der gescheiterten Projekte, gar nicht über die Machbarkeitsstudie hinaus fortgeführt werden hätten dürfen. Oftmals spielen aber innerhalb des Betriebs andere Komponenten, wie persönliches Ego und interne Politik eine wichtige Rolle.

Wie sollen wir das Projekt durchführen?

Highsmith nennt drei wichtige Dokumente, die diese Frage beantworten sollen.

1. Die „Project Vision“; darin werden innerhalb von zwei bis zehn Seiten folgende Fragen beantwortet: Hauptvorteil des Projekts, spätere Anwender, Projekthintergrund, Projektumfang in Functionpoints, Zeitrahmen, Budget, Personalbedarf, Exekutiver Sponsor, Risiko und grössere Projektvision.

2. Das „Project Data Sheet“; darin geht es darum, das Allerwichtigste des Projektes auf einer A4-Seite zusammenzufassen, was sich oftmals als sehr schwierig herausstellt, und im Spruch „Wenn ich genug Zeit gehabt hätte, dann hätte ich einen kürzeren Brief geschrieben“ am besten ersichtlich wird.

3. Die „Product Specification Outline“; dabei ist es sehr wichtig den Projektrahmen abzustecken. Oftmals ist es sogar wichtiger zu nennen, was das Projekt *nicht* beinhaltet. Da es bei adaptiven Projekten sehr viele Änderungen gibt, gilt es aber auf allzu detaillierte Spezifikation zu verzichten und die genaueren Details, wenn überhaupt, erst im Laufe des Projektes zu dokumentieren.

Eine weitere wichtige Frage, die in diesem Zusammenhang auftaucht, ist die der Qualität. Was verstehen wir unter „Softwarequalität“. Dabei handelt es sich sicher um eines der am kontroversesten diskutierten Themen der Softwareentwicklung. Ohne allzu weit auszuholen, kann man die Meinungen in zwei Gruppen aufteilen. Auf der einen Seite stehen die technisch orientierten, meist akademisch geprägten Befürworter von absolut sauberer, höchsten Qualitätsstandards genügenden, Ingenieurprodukten, auf der anderen Seite stehen „business orientierte“ Pragmatiker, für die Software gerade gut genug sein soll, um ihre Aufgabe zu erfüllen. Highsmith zählt eindeutig zu letzterer Gruppe, so zitiert er auch immer wieder Microsoft als vorbildliche Firma. Für Highsmith ist es insbesondere wichtig, nicht jede Software an den selben Massstäben zu messen, so sollte es klar sein, dass man eine Nasaraumfähre anders entwickelt, als einen Webbrowser.

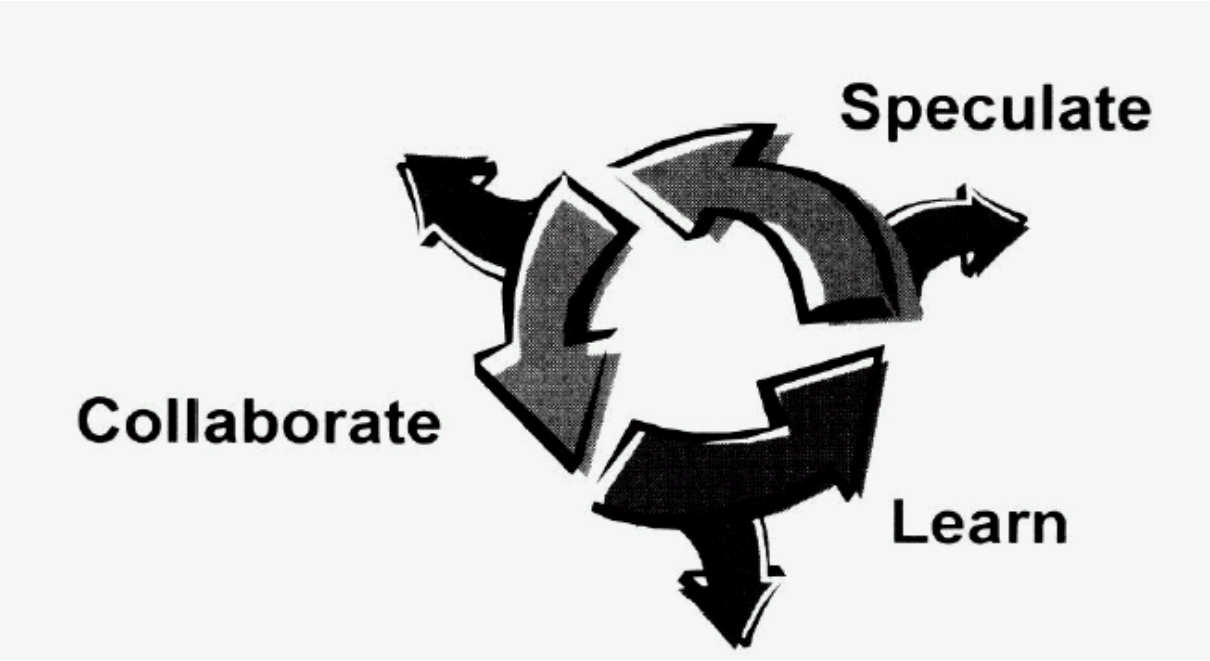
6.1.2. Adaptive Development Life Cycle

Während vieler Jahre galt das Wasserfallmodell als „state of the art“. Dabei ging es darum sequentiell zuerst alles zu planen, dann die Software entsprechend der Spezifikation zu entwickeln und sie schlussendlich dem Kunden zu übergeben. Diese Methode konnte natürlich mit zunehmenden Änderungen nicht mehr aufrechterhalten werden, was zum Einsatz von iterativen Methoden, wie „evolutionäre Entwicklung“ und Spiralmodell[5] führte. Dabei wird versucht mit Meilensteinen das Projekt in kleinere Teile zu zerlegen. Implizit wird aber immer noch eine Vorhersagbarkeit, jetzt einfach für einen kürzeren Zeitraum, angenommen. So erstaunt es nicht, dass die einzelnen Teile immer noch sehr genau spezifiziert und dokumentiert werden. Für Projekte die sehr schnell durchgeführt werden müssen, wurden in den letzten Jahren neue Methoden entwickelt, zum Bespiel die „RADical Life Cycles“, wobei man wirklich fast nichts mehr dokumentiert, und nur noch toolunterstützt einzelne Notizen im Quellcode anbringt. Diese Methode ist jedoch für Projekte mit vielen Änderungen nicht zu gebrauchen, weshalb Highsmith einen Bedarf für adaptive Methoden sieht.

	low Speed	high Speed
low Change	Waterfall	RAD
high Change	Spiral + „evolutionary“	Adaptive

Wer jetzt eine weltbewegende Neuschöpfung wie einen spiralförmig, aufwärtsfliessenden Wasserfall erwartet, wird jedoch schon bald enttäuscht. Highsmith übernimmt praktisch die evolutionären Methoden, verzichtet aber aufgrund der sich schnell ändernden Anforderungen auf strikte Aufgabenverteilung und Kontrollstrategien. Er vertraut vielmehr auf die obenangesprochene Emergenz, das stillschweigende, gemeinsame Suchen nach der besten Lösung. Die Grafik auf der nächsten Seite soll dies veranschaulichen.

[5] Boehm, Barry „A spiral model of Software Development and Enhancement“ IEEE Computer, Vol 66, No. 5 pp 61-72



Highsmith benennt die Phasen des iterativen Zyklus um und erweitert ihn um daraus ausscherende Pfeile. Die traditionellen Phasen des iterativen Ansatzes „Plan,Revise, Build“ werden zu „Speculate, Learn und Collaborate“ und somit zum „Adaptive Development Life Cycle[6]“.

Bei der „Speculate“-Phase geht es darum, einen nächsten Zyklus zu planen. Der Name soll aber klar machen, dass man sich von der traditionellen Sicht, dass eine Abweichung vom Plan, etwas Schlechtes ist und sofort korrigiert werden muss, falsch ist. Vieleher soll man spekulieren dürfen, denn nur so kann Emergenz entstehen.

Die „Collaboration“-Phase bezeichnet die eigentliche Implementierung eines Zyklus’, wobei Zusammenarbeit eine grosse Rolle spielt. So muss der Begriff von der passiven Kommunikation abgegrenzt werden, wo es mehr um das Verschicken von vorgefertigten Berichten, als um das Anstreben eines gemeinsamen Zieles, und damit das Erreichen von Emergenz geht.

Unter der „Learn“-Phase subsumiert Highsmith alles was mit Kundenbefragen, Testen, Prüfen und Reviewen zu tun hat. Zugegebenermassen tönt lernen besser als Fehler korrigieren.

Die ausscherenden Pfeile sollen nochmals zusätzlich betonen, dass auch unkonventionelle Vorgehensweisen und Ideen willkommen sind, um eben diese sagenumwobene Emergenz zu erreichen.

---

[6] James A. Highsmith III: *Adaptive Software Development - A Collaborative Approach to Managing Complex Systems* , Seite 41

Meiner Ansicht nach wird hier etwas zuviel auf magische Dinge wie Emergenz abgestützt; da hilft auch eine abschliessende Aussage von Highsmith, wie diese nicht viel: „Ein gutes Projektteam weiss, wann Strenge angebracht ist, während ein chaotisches Team Strenge als Bürokratie abtut.“

6.2 Adaptive Development Model

Das “Adaptive Development Model” beschreibt wie die Projektzyklen zu organisieren sind und was innerhalb dieser Zyklen alles zu beachten ist. Jeder Zyklus besteht aus den bereits kennen gelernten Phasen „Speculate“, „Collaborate“ und „Learn“. Wichtigster Unterschied zu traditionellen iterativen Modellen ist, dass die adaptiven Zyklen resultatorientiert und nicht taskorientiert sind. Das heisst, dass das Ergebnis jedes Zyklus ein Stück vorführbare Software ist. Dieser Punkt ist sehr wichtig, da bei herkömmlichen Projekten oftmals, während der Planungsphase, einfach die Tasklisten des letzten Projekts kopiert werden und man der Meinung ist, dass das Ganze etwa ähnlich verlaufen wird. Hierbei kann man natürlich keinesfalls von Planung sprechen. Ein weiterer Vorteil des Komponentenansatzes ist, dass das Team jederzeit genau vor Augen hat, was als nächstes entwickelt wird und nicht bloss voneinander separierte Kleinstaufgaben löst, oder noch schlimmer, nur Papier produziert.

Ein weiterer wichtiger Punkt des adaptiven Ansatzes ist, dass die ganze Philosophie auf das „Cyclen“ ausgerichtet ist. Dabei soll im Gegensatz zum Wasserfallmodell, wo das Augenmerk auf einem kontinuierlichen „Flow“ liegt, zum Ausdruck kommen, dass Änderungswünsche als etwas Positives, als Chance etwas Neues zu lernen, angesehen werden und nicht als Schikane oder schlechte Planung.
Wie man das einem überarbeiteten Mitarbeiter klar machen soll, bleibt mir persönlich allerdings schleierhaft.

Noch wichtiger ist das so genannte „Time-Boxing“. Es geht dabei darum für jeden Zyklus einen verbindlichen Zeitrahmen zu setzen, der notfalls mit Hilfe von schmerzlichen Kompromissen eingehalten werden muss. Ausnahmen können nur gemacht werden, wenn alle Betroffenen, das heisst Management und Kunden zustimmen. Ein weiterer grosser Vorteil von strengen Zeitrahmen ist, dass auch unfertige Software den Teamkollegen präsentiert werden muss. Dies ist für das gegenseitige Voneinanderlerner sehr hilfreich, da im Normalfall nur sehr ungern Halbfertiges präsentiert wird.

Wie sieht nun so ein adaptives Projekt konkret aus? Es mag den Leser verwundern, aber eigentlich doch sehr ähnlich wie ein „normales“ Projekt:

Vorbereitung:

Schritt 1
- Machbarkeitsstudie
- Basisarchitektur
- Umfang gemessen in Functionpoints
- Exekutivsponsor

Schritt 2
- Projektdauer
- Personalressourcen

Schritt 3
- Anzahl Zyklen und Timebox pro Zyklus

Schritt 4
- Welche Komponenten werden von welchen Zyklen produziert

Projektstart:

Zyklus 1

Der erste Zyklus ist nie etwas Angenehmes. Er ist meist von Unsicherheit, Fehlerhaftigkeit und Ärger geprägt. Sehr oft kommt es vor, dass die Mitarbeiter bereits den ersten Zeitrahmen ausdehnen wollen. Dieser Versuchung ist unter allen Umständen zu widerstehen, da von höchster Wichtigkeit ist, möglichst schnell ein erstes Mal vor den Kunden zu treten und etwas Konkretes zu präsentieren, um sofort Unklarheiten auszubügeln.
Produkte des ersten Zyklus sind: Menus und Navigationsleisten, Anfragemasken, Middlewarekomponenten wie Datenbankanbindungen, alle Technologiekomponenten, um sicherzustellen, dass für Installation und Training genügend Zeit bleibt und Klarheit über Verwendung von Namensstandards und Designgrundsätze, die vom hoffentlich bereits installierten Tool, angeboten werden

Zyklus 2

Der zweite Zyklus dient dazu, neu gewonnene Anforderungen aus dem ersten Kundenreview umzusetzen und die komplexeren Algorithmen, die in Zyklus 1 noch aufgeschoben wurden, zu implementieren. Weiter werden Fehler beseitigt und rigoros getestet. Ergebnis des Zyklus 2 muss ein ausreichend stabiles System sein, um den Kunden selber erste Erfahrungen damit sammeln zu lassen. Auf Dokumentationsseite gilt es einen ersten Rahmenentwurf für Auslieferung und Installationsprozess bereitzustellen.

Zyklus 3

In Zyklus 3 werden alle Komponenten so sehr verfeinert, dass sie allen Anforderungen genügen und ein akzeptabler Fehlerlevel erreicht wird. Das System sollte in diesem Zyklus von technischer Seite her fertig gestellt werden.

Zyklus 4

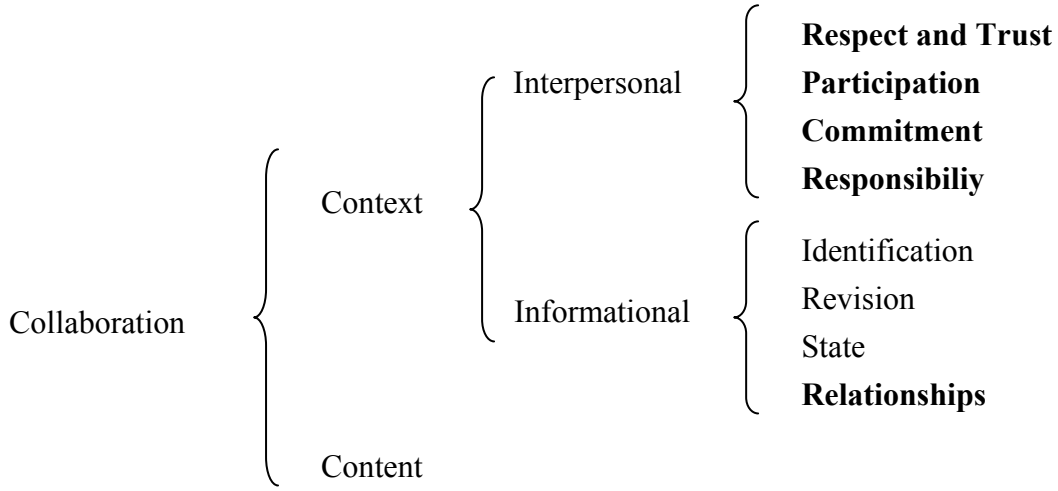
In Zyklus 4 folgen alle noch zu erledigenden Aufgaben wie Anwenderdokumentation, Installationsroutinen und Trainingsprogramme.

6.3 Adaptive (Leadership-Collaboration) Management Model

Highsmith beginnt dieses Kapital mit dem Resultat, einer von ihm durchgeführten Umfrage. Die Kandidaten sollten verschiedene Projektalltagstätigkeiten auf einer Skala von 1 bis 100 nach Beliebtheit bewerten. Dabei erhielten folgende Tätigkeiten deutlich unter 30 Punkten: Meetings, Präsentationen, Dokumentation und Zusammenarbeit mit anderen Gruppen.
Das Resultat zeigt, dass viele Softwareentwickler nach wie vor am liebsten alleine arbeiten und Zusammenarbeit als unnütz und langweilig empfinden. Während dies teils sicher auf die zunehmend individualisierte Gesellschaft zurückzuführen ist, spielt aber auch der vorherrschende hierarchische Führungsstil eine bedeutende Rolle. So gibt es in einer Unternehmung, in der jeder als ersetzbar gilt, sehr wenig Anreize, sein Wissen mit anderen zu teilen.
In letzter Zeit wird aus diesem Grund immer mehr Wert auf Teamarbeit gelegt. Dieses Motto, bleibt aber ein leeres Bekenntnis, wenn man nicht die richtigen Leute aussucht und nicht die Zusammenarbeit des Teams sauber vorausplant. Als erstes muss aber erwähnt werden, dass man nicht vollständig auf Hierarchie verzichten kann. So braucht jedes Projekt einen starken, im Umgang mit Menschen ausgesprochen talentierten Projektleiter. Im Wort Leader soll zum Ausdruck kommen, dass „Befehl“ durch „Leadership“ und „Kontrolle“ durch „Zusammenarbeit“ ersetzt wird.

Der schwierigste Prozess ist sicher das Finden der richtigen Leute. Man braucht für jede Projektrolle die richtige Person. So macht es keinen Sinn, die fünf besten Programmierer in ein Team zu stecken, aber auf einen guten Kommunikator, der die Kunden für sich gewinnen kann, zu verzichten.

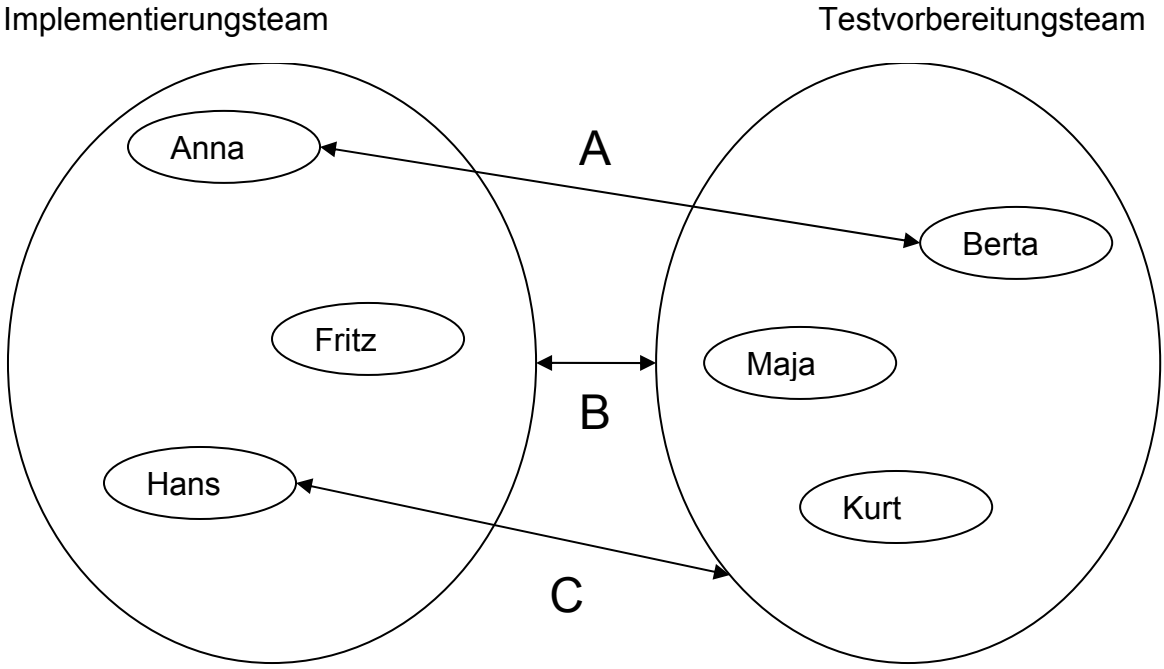
Schliesslich gilt es die Kommunikation und Kollaboration zu planen. Natürlich denkt man jetzt an die unzähligen, auf dem Markt erhältlichen Groupwaretools. Highsmith geht es dabei aber weniger um Technik, als um die zwischenmenschlichen Beziehungen. Dies sei an folgendem Diagramm[7] verdeutlicht:



Während Dinge wie Dokumentation und aktuelle Quelltexte sehr gut toolunterstützt an alle beteiligten Personen verteilt werden können, ist das Erreichen von interpersonellen Eigenschaften nur durch Beziehungen möglich. Diese Beziehungen gilt es durch so genannten „rich context“ zu festigen. Highsmith versteht darunter direkte Beziehungen zwischen einzelnen Teammitgliedern, die unter Umständen global verteilt sein können. Dabei können diverseste Kommunikationsformen, wie E-mail, Chat, oder gemeinsames Zeichnen auf einem virtuellen Whiteboard angewendet werden. Noch besser aber erachtet Highsmith die Möglichkeit sein Gegenüber mittels Videokonferenz „live“ sehen zu können und so eine Beziehung zu etablieren, die über die einzelnen spezifischen Projektfragen hinausgeht.

[7] James A. Highsmith III: *Adaptive Software Development - A Collaborative Approach to Managing Complex Systems* , Seite 266

Bei teamübergreifenden Projekten schlägt Highsmith zudem vor, konkrete 1:1-Beziehungen zu etablieren, um die Interteamkommunikation sicherzustellen. Auf diese Art soll vermieden werden, dass zum Beispiel das Dokumentationsteam vom Implementierungsteam zu spät über eine Änderung informiert wird. Dies ist auf folgender Graphik zu sehen:



Da Berta früher auch einmal im Implementierungsteam war, wird sie von Anna gerne um gute Ratschläge gebeten (A). Gewisse Informationen müssen offiziell von einem Team zum anderen übertragen und dokumentiert werden (B). Hans arbeitet an einem experimentellen Teil der Software und hält das Testteam auf dem Laufenden, damit sie sich schon mal damit auseinandersetzen können (C).

Um all diese Techniken erfolgreich zu etablieren wird auch ein Leader eines adaptiven Projektes nicht um Meetings herumkommen. Da ein guter Leader aber auch einmal auf seine Führungsrolle verzichten kann, werden so genannte „Facilitator“ eingesetzt. Dabei handelt es sich um Kommunikationsprofis, deren einzige Aufgabe es ist, Meetings zu planen und durchzuführen.

7. Beurteilung

Nachdem meine persönlichen Gedanken bereits an verschiedenster Stelle, während dieses Textes zum Ausdruck gekommen sind, möchte ich mit den vier Fragen[8], auf welche eine neue Technologie Antwort geben können muss, schliessen:

Was erweitert / verbessert Adaptive Software Development (ASD)?

ASD fordert Systemdenken und könnte ein Anstoss sein, die riesigen, selten befolgten Projekthandbücher zu kürzen. Es wird nach wie vor zu wenig Denken investiert und zuviel auf strikte Prozesse gebaut. Ob aber ASD der Weisheit letzter Schluss auf dieses Problem ist, bleibt fraglich. Schlussendlich bringt ASD kaum neue Ideen, sondern mischt nur Vorhandenes neu zusammen.

Was wird durch Adaptive Software Development überflüssig?

Eventuell wird man auf das dicke Projekthandbuch ganz verzichten können und sich mehr der dynamischen Zusammenarbeit widmen können.

Was kommt mit Adaptive Software Development zurück?

Der Faktor Mensch erhält wieder eine wichtigere Rolle, und wird nicht mehr nur als sauber funktionierendes Zahnrad im Getriebe einer Unternehmung betrachtet.

Welche unbeabsichtigten Konsequenzen könnte Adaptive Software Development auslösen?

Die grösste Gefahr besteht darin, dass man unter dem Deckmantel von ASD beginnt aus Bequemlichkeit auf unangenehme Aufgaben ganz zu verzichten und das Projekt so zum Adhoc-Vorgehen ausartet.
Schlussendlich denke ich, steht und fällt ein Projekt mit den beteiligten Personen. Auch ein ASD-Ansatz wird nicht fehlendes Wissen oder nichtvorhandenes Engagement ersetzten können. Wie in so vielen Dingen im Leben, ist es wohl einzig und allein die Erfahrung, die einem auf den richtigen Weg führen kann.

[8] McLuhan, Laws of Media: The new Science. Toronto University Press, 1988.

Quellenverzeichnis

Highsmith III, James A.: *Adaptive Software Development - A Collaborative Approach to Managing Complex Systems ,* Dorset House Publishing, 2000

Coldewey, Jens. *Einführung in Agile Entwicklung*. www.coldewey.com, 2002.

Boehm, Barry *„A spiral model of Software Development and Enhancement“* IEEE Computer, Vol 66, No. 5 pp 61-72

www.agilemanifesto.org

McLuhan, Laws of Media: *The new Science*. Toronto University Press, 1988.