Seminar
Agile vs. klassische Methoden der Software-Entwicklung

Die Rolle der Anforderungen in agilen Methoden

Seminararbeit
von

Alex Bögli
Salstrasse 45, 8400 Winterthur
Matrikel-Nr. 00-703-538

Angefertigt am Institut für Informatik der Universität Zürich
Prof. Dr. M. Glinz
Betreuer: C. Seybold

Inhaltsverzeichnis

1. Einleitung

Jede Softwareentwicklung basiert auf den Anforderungen an ein System. Sie stellen die Erwartungen der Kunden an das System dar und deren Erfüllung ist der wichtigste Erfolgsfaktor jeder Softwareentwicklung. Und da es kein System ohne Anforderungen geben kann, hat auch jedes Vorgehen zur Softwareentwicklung, ob methodisch oder nicht, einen bestimmten Ansatz um mit Anforderungen umzugehen.

Entwicklungen ohne bestimmte Methode stehen und fallen mit der Erfahrung der Entwickler, wobei das Spektrum vom sehr pragmatischen Vorgehen zum planlosen Chaos reicht. Den Anforderungen wird dabei meist nur wenig Aufmerksamkeit zuteil, und dann vor allem eher zu Beginn der Entwicklung, nämlich dann, wenn der Aufwand und damit der Preis des Systems abgeschätzt werden soll. Solche Entwicklungen sind generell nicht empfehlenswert und werden daher nicht weiter betrachtet.

Die methodische Softwareentwicklung wird heute primär in zwei Arten unterteilt: Entwicklungen nach klassischen Methoden und solche nach agilen Methoden. Diese haben, neben vielen anderen Unterschiede, vor allem einen Hauptunterschied: die Planbarkeit der Entwicklung. Bei klassischen Methoden wird alles daran gesetzt, den Ablauf der Entwicklung so gut wie möglich planbar zu machen, um sich danach an den Plan halten zu können, wogegen es die agilen ermöglichen, auf Unvorhergesehenes flexibel zu reagieren.

Daher wird der Entscheid, welche Art Entwicklungsmethode für ein bestimmtes Projekt zu wählen ist, sehr wesentlich von der Planbarkeit bestimmt, wobei diese wiederum primär von der Stabilität der Anforderungen abhängt. Martin Fowler schreibt dazu: „Everything else in software development depends on the requirements. If you cannot get stable requirements you cannot get a predictable plan.” [Fowler03]. Das bedeutet also, dass im Fall von stabilen Anforderungen mit Vorteil eine klassische Methode angewendet wird, während dem sich bei ständig ändernden Anforderungen eine agile empfiehlt.

Durch diesen Zusammenhang wird klar, dass die Rolle, welche eine bestimmte Entwicklungsmethode den Anforderungen zumisst, von grosser Tragweite ist. In dieser Arbeit soll daher diese Rolle anhand einiger ausgewählten Methoden aufgezeigt werden und eine Entscheidung für die eine oder andere (zumindest aus Sicht der Behandlung von Anforderungen) erleichtern.

1.1. Definition

Anforderung
1. Eine Bedingung oder Fähigkeit, die von einer Person zur Lösung eines Problems oder zur Erreichung eines Ziels benötigt wird.
2. Eine Bedingung oder Fähigkeit, die eine Software erfüllen oder besitzen muss, um einen Vertrag, eine Norm oder ein anderes, formell bestimmtes Dokument zu erfüllen.
   (IEEE 610.12-1990)

2. Klassische Methoden

2.1. Philosophie und Schwerpunkte

Die so genannten „klassischen" Methoden der Software-Entwicklung basieren alle auf der gleichen Philosophie und setzen ähnliche Schwerpunkte, seien es nun die „Urväter" der Prozessmodelle wie das Wasserfall- oder das Spiralmodell, oder deren Weiterentwicklungen wie der Rational Unified Process (RUP)[1] oder das V-Modell[2].

Sie alle teilen die Entwicklung in eine Folge von Tätigkeiten auf, in denen bestimmte Ergebnisse erarbeitet werden. Einige führen die Tätigkeiten nur einmal aus, andere wiederholen sie iterativ, aber alle erkennen im Gesamtablauf eine Abfolge von (wenigen) Phasen, wie beispielsweise Systemdefinition, Anforderungen, Konzipierung und Realisierung beim Spiralmodell (vgl. [Glinz01] S. 27) oder Inception, Elaboration, Construction und Transition beim RUP (vgl. [Kruchten00] S. 62ff).

Die Herausforderung für diese Methoden ist die grosse zeitliche Differenz zwischen der Anforderungsanalyse und der Einführung respektive das fehlende Feedback des Kunden während der Entwicklung. Sie entgegnen diesem Problem mit der Vorgabe, dass die jeweiligen Tätigkeiten sehr genau und umfassend durchzuführen und die Ergebnisse ausführlich schriftlich zu dokumentieren sind. Damit erhofft man sich, die Ergebnisse der einzelnen Phasen stabil halten zu können, damit die späteren effektiv darauf aufbauen können.

2.2. Rolle der Anforderungen

Im Hinblick auf die Wichtigkeit der Anforderungen für den Projekterfolg erstaunt es nicht, dass die klassischen Methoden gerade in diesem Bereich einen starken Schwerpunkt setzen und eine sehr genaue Analyse fordern. Das Ziel muss es sein, die Anforderungen so genau zu erfassen, dass sie über die Projektdauer hinweg möglichst stabil bleiben, damit das fehlende Feedback nicht zum Tragen kommt. Dies macht es zudem möglich, dass sie zum wesentlichsten Bestandteil des Vertrags zwischen Kunden und Lieferanten werden, das heisst, sie werden in den allermeisten Fällen auch rechtlich bindend sein. Es lohnt sich also für beide Seiten, dem Erfassen der Anforderungen genügend Beachtung zu schenken.

Wird die Anforderungsanalyse so systematisch angegangen, spricht man von Requirements Engineering (oder auch Anforderungstechnik), wie es beispielsweise Kotonya und Sommerville beschreiben [Kotonya98]. Sie identifizieren vier Tätigkeiten in diesem Prozess, nämlich

- Requirements elicitation,
- Requirements analysis and negotiation,
- Requirements documentation, and
- Requirements validation.

Diese Tätigkeiten sind logisch voneinander abhängig, haben aber keine klare Abgrenzung und laufen parallel und iterativ ab. Zu Beginn werden die Anforderungen zum Beispiel durch Befragungen oder Dokumentenanalysen erarbeitet (elicitation). Danach werden diese analysiert (analysis) und es wird entschieden, welche Anforderungen für das System relevant sind (negotiation). Dies ist nötig, da es zwangsläufig zu Konflikten zwischen Anforderungen aus verschiedenen Quellen kommt. Die akzeptierten Anforderungen müssen dann in einer für alle Projektbeteiligten verständlichen Form dokumentiert werden (documentation). Dies wird im Normalfall in

[1] http://www.rational.com/products/rup/index.jsp oder z.B. auch [Kruchten00]
[2] http://www.v-modell.iabg.de/

natürlicher Sprache und mit Diagrammen geschehen. Am Ende sollten die Anforderungen sorgfältig auf Konsistenz und Vollständigkeit geprüft werden (validation).

Das Resultat einer Anforderungsanalyse ist ein Anforderungsdokument, das folgende Aspekte abdeckt [Glinz01]:

- Funktionaler Aspekt:
  - Daten
  - Funktionen
  - Verhalten
  - Fehler
- Leistungsaspekt:
  - Datenmengen
  - Verarbeitungs-/Reaktionsgeschwindigkeit
  - Verarbeitungszeiten und –intervalle
- Qualitätsaspekt:
  - geforderte Qualitäten
- Randbedingungsaspekt:
  - einzuhaltende / zu verwendende Schnittstellen
  - Normen und Gesetze
  - Datenschutz, Datensicherung
  - Explizite Vorgabe des Auftraggebers

Da dieses Dokument die Anforderungen der Kunden an das System darstellt, sollte es fixiert und als Pflichtenheft zu Erstellung der Software verwendet werden. Änderungen während der Projektdauer sind sorgfältig zu prüfen und nur als Vertragszusatz zuzulassen. Daher sollte gleich zu Beginn des Projekts ein durchdachtes Change Management installiert werden.

2.3. Techniken

Kotonya und Sommerville nennen folgende Techniken, mit denen Anforderungen erarbeitet werden können:

- Interviewing
- Scenarios
- Soft Systems Methods
- Prototyping
- Participant observation

Alle diese Techniken haben ihre spezifischen Vor- und Nachteile und sollten durchaus auch in Kombination verwendet werden. So können beispielsweise Interviews sehr schnell und kostengünstig durchgeführt werden, es ist aber sehr schwierig, damit die Anforderungen korrekt und vollständig zu erfassen. Daher sollten sie unbedingt validiert werden, beispielsweise mit einem Prototyp.

3. Agile Methoden

3.1. Philosophie und Schwerpunkte

Der Begriff „agil" wurde massgeblich von den Autoren geprägt, die zusammen das so genannte „Manifesto for Agile Software Development"[3] unterzeichneten. In diesem Manifest erklären sie die Werte, die ihnen wichtig sind.
Diese sind:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

Im Hinblick auf die Rolle der Anforderungen ist sicher der dritte Punkt am interessantesten. Sie gewichten die Zusammenarbeit mit Kunden höher als das Aushandeln eines Vertrags und verzichten damit bewusst auf eine zentrale Funktion des Requirements Engineering zugunsten höherer Flexibilität. Dies zeigt sich noch ausgeprägter in einem der Prinzipien[4], die dem Manifest zugrunde liegen, welches lautet: "Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage."

3.2. Extreme Programming

3.2.1. Ursprung

Die Wurzeln des Extreme Programming (XP) liegen im Projekt „Chrysler Consolidated Compensation" (C3) bei Daimler-Chrysler, einem internen Abrechnungssystem. Nachdem die Entwicklung des Systems nicht wie geplant verlief, engagierte man (neben anderen) Kent Beck, Ron Jeffries und Martin Fowler, um es wieder in Gang zu bringen. Dies gelang nicht schlecht und Kent Beck hat danach die dort entstandenen Ideen zu einer Methode ausgearbeitet und sie im Buch „Extreme Programming Explained – EMBRACE CHANGE" publiziert [Beck99]. Später haben Ron Jeffries und Martin Fowler ihre Erfahrungen ebenfalls publiziert und XP damit wesentlich mitgeprägt [Beck00][Jeffries00].

3.2.2. Rolle der Anforderungen

Wie in allen Methoden spielen die Anforderungen auch bei XP eine zentrale Rolle, da sie die Wünsche des Kunden an das System darstellen und die Kundenzufriedenheit oberstes Gebot ist.

In XP werden die Anforderungen in so genannten „User Stories" erfasst. Diese werden vom Kunden selbst geschrieben, enthalten keine technischen Angaben und sind so kurz zu halten, dass sie auf eine kleine Karte passen (etwa A6). Sie beschreiben eine bestimmte Funktionalität (ein Feature), die der Kunde vom System erwartet. Die Beschreibung muss einfach und für Kunden und Entwickler gleichermassen verständlich sein. Dies ist mit ein Grund, warum die Anforderungen natürlich-sprachlich und ohne spezielle Notationen erfasst werden.

Die Anforderungsanalyse ist also absichtlich oberflächlich. Der Grund dafür liegt in einer einfachen Kosten-Nutzen-Abschätzung. Da die User Stories primär zur Planung eines Release gebraucht werden, darf nicht mehr Aufwand betrieben werden, als zur Abschätzung der Dauer einer Story nötig. Die Details werden dann erst während der Umsetzung einer Story mit dem Kunden definiert. Kent Beck schreibt dazu: „The story represents a concept and is not a detailed specification. *A user story is nothing more*

[3] http://www.agilemanifesto.org
[4] http://www.agilemanifesto.org/principles.html

than an agreement that the customer and developers will talk together about a feature." [Beck00, Seite 46].

Da das Schreiben der Stories natürlich alles andere als einfach ist, wird empfohlen, dies vorgängig zu üben. Dabei erstellt der Kunde erste Stories, die dann von den Entwicklern auf die Verwendbarkeit begutachtet werden. Sie prüfen vor allem, ob sie auf dieser Basis gute Aufwandschätzungen abgeben können. Ist das noch nicht der Fall, muss der Kunde die Stories umschreiben, bis sich beide Seiten finden.

Sind die Techniken einmal eingeübt, wird ein erster Satz an Stories geschrieben, welche die Hauptanforderungen an das System darstellen. Auf dieser Basis wird ein übergreifender Releaseplan erstellt. Während dem Projekt werden laufend Stories neu hinzugefügt, bestehende geändert oder gar entfernt. Aus dieser Menge wird jeweils eine Teilmenge herausgegriffen, die dann während einer Iteration entwickelt wird. Der Releaseplan wird dabei laufend geändert und verfeinert.

Die oberflächliche Natur der Stories zeigt sich auch deutlich nach deren Umsetzung. So empfiehlt Beck, dass die Kärtchen nicht etwa aufgehoben, sondern einfach vernichtet werden. Dies scheint auf den ersten Blick schade, da doch eine wesentliche Dokumentation verloren geht. Beck lässt dieses Argument aber nicht gelten und schreibt: „Remember that the stories are encoded in far more detail and accuracy in the acceptance tests, so no information will be lost if the cards themselves are destroyed." [Beck00, Seite 52].

3.2.3. Beurteilung

Der Umgang mit Anforderungen in XP besticht durch seine Einfachheit und bietet einen erfrischenden Kontrast zum Requirements Engineering in klassischen Methoden. Leider ist es aber mit diesem Ansatz nicht möglich, die Anforderungen als Vertragsgrundlage zu verwenden, da sie bei weitem nicht im nötigen Detaillierungsgrad erfasst werden. Zudem kann der Ansatz nur gelingen, wenn andere Praktiken von XP eingesetzt werden, wie zum Beispiel einen Kunden vor Ort zu haben. Dieser Kundenvertreter muss insbesondere auch kompetent genug sein, die schwierige Aufgabe zu übernehmen, die (meist divergierenden) Anforderungen aller Interessensgruppen (Stakeholders) zu gewinnen und zusammenzuführen.

Hier zeigt sich eben der Ursprung von XP als Entwicklungsmethode für interne Projekte. Dort ist eher ein Kundenvertreter verfügbar, es muss nicht unbedingt auf vertraglicher Basis gearbeitet werden und es ist generell wichtiger, auf ändernde Anforderungen flexibel reagieren zu können als (rechtlich abgesichert) ein zu Beginn definiertes System zu erhalten. Ist man nun ebenfalls in einer solchen Situation und hat das Problem sich ständig verändernder Anforderungen, ist der bei XP gewählte Ansatz sicher besser geeignet als ein Requirements Engineering wie bei klassischen Methoden, da er sehr viel flexibler ist.

3.3. Extreme Requirements

3.3.1. Ursprung

Extreme Requirements (XR) versteht sich als Weiterentwicklung von XP, speziell im Bereich der Anforderungen. Präsentiert wurde es von Julio Cesar Sampaio do Prado Leite als Keynote an einer Konferenz in Sevilla [Leite01]. Er sieht im Zusammenhang mit Anforderungen bei XP vor allem folgende 5 Probleme (übersetzt):

1. Die Annahme, dass ein einziger Kunde alle Stakeholders repräsentieren kann
2. Die fehlende Beachtung nichtfunktionaler Anforderungen aus Kundensicht
3. Die fehlende explizite Verbindung zwischen Stories, Tasks und Code
4. Das Fehlen eines Prozesses zur Definition funktionaler Tests
5. Das Fehlen eines Prozesses zur Gewinnung von Stories und Tasks

3.3.2. Lösungsvorschläge

Da ein einzelner Kundenvertreter kaum das gesamte Business repräsentieren kann (Problem 1), müssen noch andere involviert werden. Dazu wird die Ergänzung der Stories und Tasks mit Szenarien empfohlen. Diese werden ebenfalls vom Kunden vor Ort geschrieben, können dann aber von weiteren Kundenvertretern begutachtet werden. Dabei sollten diese sorgfältig ausgewählt werden, um das Business adäquat zu repräsentieren.

Die fehlende Beachtung der nichtfunktionalen Anforderungen (Problem 2) kann zu einer Erhöhung der Kosten und zu schwerwiegenden Qualitätsproblemen führen. Daher wird vorgeschlagen, diese so früh wie möglich zu definieren und sie zusammen mit den funktionalen Anforderungen (User Stories) festzuhalten.

Um eine Verbindung zwischen Stories, Tasks und Code herzustellen (Problem 3), wird das Führen eines Glossars („language extended lexicon") empfohlen, in dem die Begriffe aus der relevanten Geschäftswelt beschrieben werden. Jeder Eintrag („symbol") wird als Begriff erklärt und dessen Verhalten im gegebenen Kontext beschrieben. Durch die konsequente Verwendung dieser Begriffe in allen Artefakten werden die Verbindungen klarer.

Um zu guten funktionalen Tests zu gelangen (Problem 4), sollten ebenfalls Szenarien erstellt werden und aus diesen dann die Testfälle abgeleitet werden. Der Kunde sollte dabei zu jedem Szenario mehrere Beispiele erarbeiten, die sowohl den Normalfall wie auch die Ausnahmefälle abdecken.

Da XP kein systematischer Prozess zur Gewinnung von Stories und Tasks definiert (Problem 5), wird das in XR nachgeholt. Der vorgeschlagene Prozess definiert die nötigen Schritte um sowohl User Stories als auch die detaillierten Szenarien zu gewinnen. Diese werden zusätzlich noch verifiziert und validiert, was sicherlich wünschenswert ist.

3.3.3. Beurteilung

Extreme Requirements ist vor allem insofern ein interessanter Vorschlag, als er einige Probleme von XP im Bezug auf den Umgang mit Anforderungen aufzeigt. Die Lösungen (Szenarien, Glossar) sind jedoch ziemlich offensichtlich aus dem Bereich der klassischen Methoden entnommen und es ist daher fraglich, ob damit nicht ein Teil der Agilität verloren geht.

3.4. Scrum

3.4.1. Ursprung

Scrum ist ein leichtgewichtiger Management Prozess, dessen Ideen in den neunziger Jahren entstanden und der 2001 von Ken Schwaber und Mike Beedle in einem Buch veröffentlicht wurde [Schwaber01].

3.4.2. Rolle der Anforderungen

Bei Scrum werden die Anforderungen im so genannten Product Backlog verwaltet. Dies ist eine nach Prioritäten geordnete Liste der noch zu entwickelnden Funktionalitäten. Sie ist grundsätzlich allen zugänglich, wird aber nur vom Product Owner geführt. Er nimmt Wünsche zur Ergänzung der Liste entgegen und entscheidet über deren Aufnahme. Ebenfalls schätzt er zusammen mit dem Scrum Team den Aufwand und priorisiert den Eintrag entsprechend.

Die Entwicklung der Software geschieht in Iterationen, den Sprints. Sie dauern etwa 30 Tage und müssen ein gesetztes Ziel erreichen. Dazu wird für jeden Sprint eine Teilmenge des Product Backlog ausgewählt (das Sprint Backlog) und dem Team als Ziel vorgegeben. Für die Umsetzung wird die Funktionalität weiter unterteilt in Backlog Items, die dann während dem Sprint umgesetzt werden. Am Ende eines Sprints wird das Erreichte im Sprint Review Meeting kritisch begutachtet und das weitere Vorgehen beschlossen. Abbildung 1 zeigt diesen Ablauf schematisch auf.

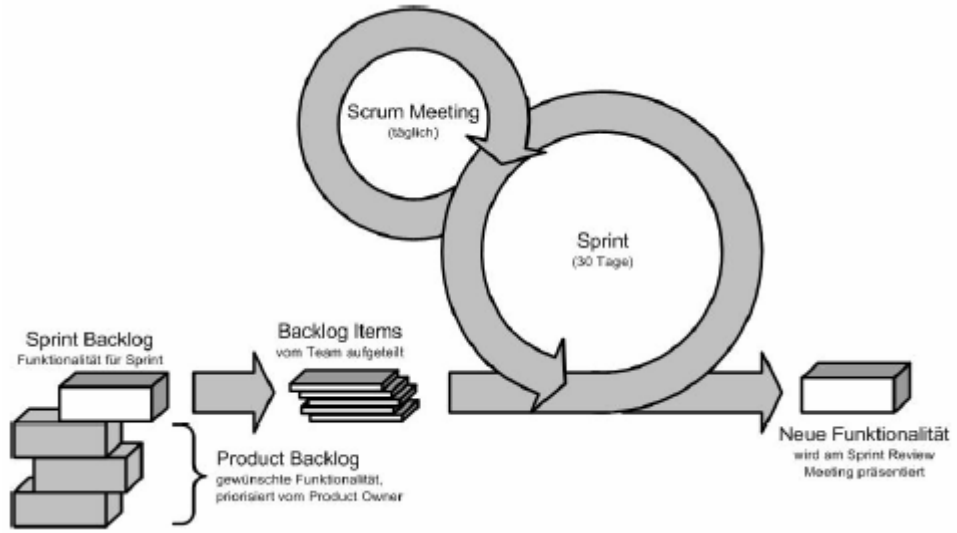

Abbildung 1 - Funktionsweise von Scrum (aus [Schweitzer03])

Im Zusammenhang mit Anforderungen stehen bei Scrum also folgende Punkte im Vordergrund:

- Das Product Backlog als priorisierte Liste aller Anforderungen
- Der Product Owner als alleiniger Verwalter des Product Backlog
- Während dem Sprint sind die Anforderungen fixiert
- Das Sprint Review als Mittel zum Informationsaustausch zwischen Kunden und Entwickler

Scrum definiert weder eine bestimmte Form, wie die Anforderungen zu erfassen sind, noch ist das Erfassen selbst Bestandteil der Methode. Es wird vorausgesetzt, dass der Product Owner und der Kunde genügend kompetent sind, um dies durchzuführen. Die

Einträge im Backlog können durchaus zu Beginn noch vage sein. Wichtig ist, dass sie mit steigender Priorität laufend detaillierter beschrieben werden.

3.4.3. Beurteilung

Die Zentralisierung der Anforderungen in einer Liste, verwaltet von einer Person, bietet den Vorteil, dass die häufig anzutreffende Situation, dass mehrere Versionen der Anforderungsdokumente im Umlauf sind, nicht mehr entstehen kann. Zudem bringt die Fixierung der Anforderungen (während den Sprints) eine Stabilität in die Entwicklung, wie sie sonst eher von klassischen Methoden bekannt ist.

Leider bietet Scrum keine Hilfestellung, wie die Anforderungen erarbeitet werden können, das heisst, es sind weder Techniken noch Vorgehensweisen angegeben. Hier ist der Product Owner gefordert, sich entweder an den klassischen Methoden zu orientieren oder Scrum mit einer agilen wie Extreme Programming zu kombinieren, wobei diese (in diesem Zusammenhang) auch nur beschränkt nützlich sind.

Zudem spielen bei Scrum die Anforderungen keine Rolle als Vertragsgrundlage. Dies macht es schwierig, Scrum für andere Projekte als interne Entwicklungen zu verwenden da nur wenige Kunden bereit sein werden zu akzeptieren, dass das Team entscheidet wann es was macht.

4. Fazit

Die Anforderungen haben in klassischen und in agilen Methoden grundsätzlich verschiedene Rollen. In den klassischen werden sie erfasst, um als verbindliche Grundlage für die Entwicklung zu gelten. Daher können sie auch als Grundlage für Verträge verwendet werden. In den agilen definieren sie zwar ebenfalls, was entwickelt werden soll, die verbindliche Komponente fehlt aber. Dies macht es unmöglich, auf dieser Basis Verträge auszuhandeln. Dort müssen die Leistungen anders definiert werden, etwa als Leistung und Abgeltung nach Zeit.

Die vorgestellten agilen Methoden weichen zwar in Details voneinander ab, weisen aber starke Ähnlichkeiten auf:

- Sowohl User Stories als auch das Product Backlog garantieren eine zentralisierte Liste von Anforderungen.
- Durch den Kunden vor Ort wie durch den Product Owner werden die Entwickler entlastet von sich widersprechenden Anforderungen und der Priorisierung.
- Kleine und regelmässige Iterationen ermöglichen zu lernen und die Anforderungen weiterzuentwickeln.

Leider sind sie sich auch ähnlich in fehlender Unterstützung für denjenigen, der die Anforderungen erfassen sollte. Sie geben weder einen Prozess noch eine Technik vor, wie die Anforderungen gewonnen und priorisiert werden können. Des Weiteren fehlen wichtige Aspekte wie nichtfunktionale Anforderungen oder Ableiten von Testfällen.

Weiterentwicklungen wie Extreme Requirements versuchen, diese Probleme zu umgehen und Abhilfe zu schaffen, geraten aber zu leicht ins Fahrwasser der schweren, klassischen Methoden.

I. Literaturverzeichnis

[Beck99] Kent Beck
Extreme Programming Explained – EMBRACE CHANGE
Addison-Wesley, 1999

[Beck00] Kent Beck, and Martin Fowler
Planning Extreme Programming
Addison-Wesley, 2000

[Fowler03] Martin Fowler
The New Methodology
martinfowler.com, 2003
http://martinfowler.com/articles/newMethodology.html

[Glinz01] Martin Glinz
Software Engineering I, Vorlesungsskript
Institut für Informatik, Universität Zürich, 1999

[Jeffries00] Ron Jeffries
Extreme Programming Installed
Addison-Wesley, 2000

[Kotonya98] Gerald Kotonya, and Ian Sommerville
Requirements Engineering, Processes and Techniques
John Wiley & Sons, Chichester, 1998

[Kruchten00] Phillippe Kruchten
The Rational Unified Process: An Introduction (2nd edition)
Addison-Wesley, 2000

[Leite01] Julio Cesar Sampaio do Prado Leite
Extreme Requirements (XR)
Jornadas de Ingenieria de Requisitos Aplicadas, Sevilla, 2001
http://citeseer.nj.nec.com/leite01extreme.html

[Schwaber01] Ken Schwaber, and Mike Beedle
Agile Software Development with Scrum
Prentice Hall, 2001

[Schweitzer03] Raffael Schweitzer
Scrum, eine agile Methode zur Software Entwicklung, Seminararbeit
Institut für Informatik, Universität Zürich, 2003