Agile Softwareentwicklung mit Scrum

Fabian Chanton und Sven Schumacher

chanf2@bfh.ch, schus4@bfh.ch

Zusammenfassung. Bei Softwareentwicklungsprojekten ist es oftmals der Fall, dass die Anforderungen an die Software häufigen Änderungen unterworfen sind. Da bei sequentiellen Prozessen wie dem Wasserfallmodell nur sehr schlecht auf solche Änderungen reagiert werden kann, entstehen vielfach erhebliche Probleme während eines Projekts. Aus dieser Problematik heraus entstanden die Grundprinzipien der agilen Softwareentwicklung. Diese Prinzipien wurden im sogenannten agilen Manifest festgehalten und sollen die Softwareentwicklung allgemein verbessern, insbesondere soll besser auf Änderungen reagiert werden können. Schliesslich entstanden agile Softwareentwicklungsprozesse, welche auf den Grundwerten des agilen Manifests aufbauen. *Scrum* ist ein solcher agiler Prozess. In *Scrum* wird versucht, die Produktivität zu steigern, indem man den Prozess schlank hält. Es wird bewusst auf eine aufwändige Dokumentation verzichtet, stattdessen werden täglich kurze Meetings abgehalten. Dem Entwicklerteam werden grosse Freiheiten bezüglich Organisation und Vorgehensweise gelassen. Die Produktivitätssteigerung wird bei *Scrum* vor allem durch mehr Freude an der Arbeit und weiteren positiven psychologischen Effekten erzielt.

1. Einleitung

IT-Verantwortliche sehen sich zu Beginn eines Softwareentwicklungsprojektes oft mit der Frage nach einem geeigneten Entwicklungsprozess, also einem konkreten Vorgehen zur Realisierung einer Lösung, konfrontiert.
Dabei steht ihnen eine Vielzahl möglicher Varianten zur Verfügung. Von sequentiellen Prozessen wie dem Wasserfallmodell, welches das Abarbeiten einzelner Phasen vorschreibt, bis hin zu iterativen, flexiblen Vorgehensweisen steht eine grosse Bandbreite zur Auswahl.
In den letzten Jahren zeigte der Trend immer mehr weg von statischen und durchgeplanten Prozessen, wie dem eben erwähnten Wasserfallmodell, hin zu dynamischen und flexiblen, den sogenannten agilen Entwicklungsmethoden.
Ziel dieser Arbeit ist es, den Begriff der agilen Softwareentwicklung zu definieren und sequentiellen Prozessen wie dem Wasserfallmodell gegenüberzustellen.
Anschliessend wird *Scrum* als konkrete Form agiler Softwareentwicklung umfassend vorgestellt. Der Leser sollte abschliessend in der Lage sein, zu beurteilen ob *Scrum* in Bezug auf den Entwicklungsprozess seiner Projekte einen Mehrwert bringt.

2. Agile Softwareentwicklung

2.1. Vorbemerkungen

Dieses Kapitel hat zum Ziel, den Begriff der agilen Softwareentwicklung einzuführen und zu definieren. Um dessen Vor- und Nachteile zu veranschaulichen, wird zuerst auf das Wasserfallmodell eingegangen, so dass anschliessend ein Vergleich mit der agilen Methodik angestellt werden kann.

2.2. Das Wasserfallmodell

Das Wasserfallmodell ist ein sequentieller Softwareentwicklungsprozess, bei dem jede Phase linear durchschritten wird. Die nächste Phase kann also erst begonnen werden, nachdem die vordere vollständig abgeschlossen ist. Die einzelnen Phasen und ihre Reihenfolge sind in Fig. 1 ersichtlich. Das Wasserfallmodell gehört nicht zu den agilen Vorgehensweisen, wird hier jedoch Zwecks Vergleich trotzdem erwähnt.

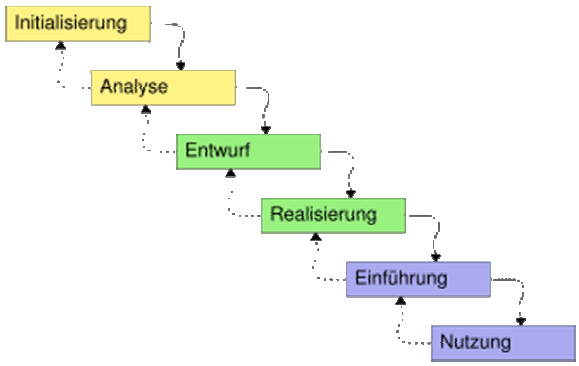

Fig. 1. Ablauf der Phasen des Wasserfallmodells
(von http://pics.computerbase.de)

Wie in Fig. 1 zu sehen ist, geht der Entwicklungsphase immer eine vollständige Analyse- und Entwurfsphase voraus. Womit wir beim Hauptkritikpunkt des Modells angelangt wären, da angenommen wird, dass bei Projektbeginn bereits alle Anforderungen bekannt sind und dass sich diese während der Entwicklung nicht mehr ändern werden. Die Realität sieht jedoch bedeutend anders aus, denn laut einer Studie von Boehm und Papaccio ändern sich in einem typischen Software-Projekt etwa 25% der Anforderungen [1]. Solch hohe Änderungsraten sind jedoch im Wasserfallmodell nicht vorgesehen. Zwar ist ein Rücksprung zur vorderen Phase, wie in Fig 1. durch die gestrichelten Pfeile ersichtlich, möglich, doch nützt dies wenig wenn beispielsweise erst bei der Einführung des Produkts gravierende Fehler zu Tage

treten. Denn dann wird modellbedingt ein erneuter Durchlauf nötig, was nicht zu unterschätzende Mehrkosten mit sich bringt. Ein grosses Problem, welches aus der Anwendung des Wasserfallmodells resultiert, ist, dass der Kunde meistens das erhält, was im Vertrag steht, jedoch nicht das, was er tatsächlich braucht. Die Begründung hierfür liegt in der Tatsache, dass strikt nach Planung vorgegangen wird und der Kunde erst in der Einführungsphase mit der bereits fertig gestellten Software in Kontakt kommt.

2.3. Agiles Manifest

Aus der Erkenntnis, dass Softwareentwicklung tendenziell einem ständigen Wandel der Anforderungen unterworfen ist und der Kunde in traditionellen Vorgehensweisen wie dem Wasserfallmodell nicht genügend in den Entwicklungsprozess miteinbezogen wird, wurde eine Reihe neuer Werte definiert, welche unter der Bezeichnung agiles Manifest bekannt sind. Das agile Manifest kann als kleinster gemeinsamer Nenner aller agilen Vorgehensweisen betrachtet werden.
Das Manifest wurde 2001 unter Federführung der Agile Alliance, der Gruppe der Ideengeber hinter agiler Softwareentwicklung, entworfen. Die Grundwerte sind in Fig. 2 ersichtlich.
Laut dem agilen Manifest stehen Menschen und die zwischen ihnen stattfindende Kommunikation über festgelegte Prozesse und den für die Entwicklung eingesetzten Tools. Weiter ist man der Auffassung, dass funktionierende Software wichtiger ist als eine vollständige und detaillierte Dokumentation. Die Kooperation mit dem Kunden geht über Verträge, diese werden lediglich als Grundlage für die Zusammenarbeit gesehen. Der vierte und letzte definierte Wert bezieht sich auf die Reaktion auf Änderungen, denn im Gegensatz zu sequentiellen Modellen sind agile Vorgehensweisen darauf angelegt, Änderungen jederzeit in den Entwicklungsprozess aufzunehmen. Wechselnde Anforderungen werden also begrüsst und über festgelegte Pläne gestellt.

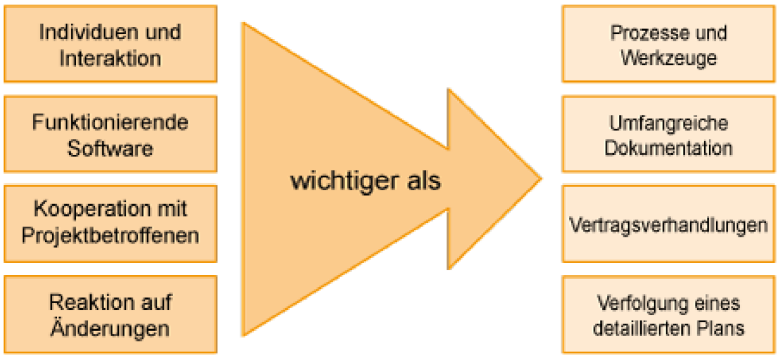

Fig. 2. Agiles Manifest und seine Grundprinzipien
(von http://www.ordix.de)

2.4. Terminologie

In der Literatur werden oft die Begriffe agiles Prinzip, agile Methoden und agiler Prozess verwendet, diese sollen im Folgenden erklärt werden.
Das agile Manifest bildet mit seinen Werten das Fundament auf welchem sogenannte agile Prinzipien aufbauen. Diese Prinzipien können als Handlungsgrundsätze verstanden werden. Ein Beispiel für einen solchen Handlungsgrundsatz ist das KISS-Prinzip. Das Akronym KISS steht für *keep it simple and stupid*, was übersetzt soviel heisst wie „halte es einfach und verständlich". Als agile Methoden werden konkrete Verfahren während der Softwareentwicklung bezeichnet, welche sich auf die definierten Werte und Prinzipien stützen. Wie zum Beispiel die testgetriebene Entwicklung, welche später noch genauer erklärt wird. Unter agilem Prozess wird schliesslich die Zusammenfassung aller angewandten agilen Methoden verstanden.

2.5. Ziel

Die agile Softwareentwicklung hat die Optimierung des Entwicklungsprozesses zum Ziel. Das Wort agil stammt übrigens aus dem lateinischen und bedeutet soviel wie flink oder beweglich. Es wird also versucht, den Entwicklungsprozess im Gegensatz zu klassischen Methoden, welche als schwergewichtig und bürokratisch gelten, flexibler und schlanker zu gestalten.
Man versucht dieses Ziel unter anderem dadurch zu erreichen, dass der Planungs- und Dokumentationsaufwand so gering wie möglich gehalten wird und menschliche sowie technische Aspekte in den Vordergrund gerückt werden. Dies soll die Konzentration auf den Kernpunkt des Projektes, nämlich die zu entwickelnde Software lenken und die Motivation der Mitarbeiter steigern.

2.6. Iterationen

Da agile Vorgehen die zeitliche Einteilung in Phasen ablehnen, stellt sich die Frage wie Projekte anders strukturierbar sind. Hier kommt der Begriff der Iteration ins Spiel. Eine Iteration ist eine kurze, zeitlich begrenzte Einheit, während der verschiedene Disziplinen der Softwareentwicklung wie Analyse, Design, Implementierung und Testen durchgeführt werden, wie dies in Fig. 3 zu sehen ist. Der gesamte Entwicklungsprozess setzt sich aus den einzelnen, aufeinanderfolgenden Iterationen zusammen. Ein wichtiges Merkmal des agilen Ansatzes ist, dass dieser im Gegensatz zum Wasserfallmodell nicht den Anspruch erhebt, bereits zu Beginn des Projektes eine komplette und vollständige Analyse und das daraus resultierende Design zu erstellen. Diese Disziplinen sind vielmehr Bestandteil jeder einzelnen Iteration, wodurch das System kontinuierlich wächst und pro Iteration jeweils nur bestimmte Aspekte verstanden werden müssen. Hier greift wie Sie evt. bereits ahnen das in Abschnitt 3.4 erwähnte KISS-Prinzip ein, welches die Bestrebung den Prozess

so einfach und verständlich wie möglich zu halten ausdrückt. Typischerweise werden zu Beginn einer Iteration die Ziele festgelegt, welche bis zum Ende des jeweiligen Durchlaufs zu erreichen sind. Dies erlaubt den Entwicklern gemäss KISS-Prinzip nur diejenigen Komponenten zu analysieren, designen, implementieren und testen, welche von Relevanz sind. Somit wird unnötige Komplexität ausgeblendet und erst in einer späteren Iteration wieder aufgegriffen. Was das Erstellen der Dokumentation angeht, so ist man der Auffassung, dass diese vielmehr dem Verständnis des Systems dienen soll als dessen detaillierter Beschreibung. Dementsprechend wird versucht, nur so viel Zeit wie nötig in diesen Bereich zu investieren und das Augenmerk auf die Fertigstellung des *Releases*, einer lauffähigen Version des bereits implementierten Systems, zu setzen, doch dazu später mehr. Ein weiterer wichtiger Begriff, welcher im Zusammenhang mit Iterationen oft auftritt, ist das Inkrement. Als Inkrement wird in der agilen Softwareentwicklung die dem System während einer Iteration neu hinzugefügte Funktionalität verstanden. Man spricht daher oft auch von inkrementeller Softwareentwicklung, da das System von Iteration zu Iteration inkrementweise wächst.

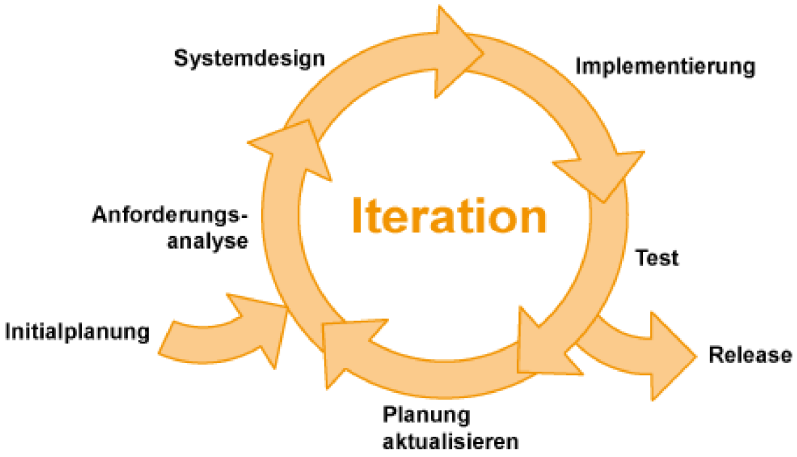

Fig. 3. Ablauf einer Iteration
(von http://www.ordix.de)

2.7. Testgetriebene Entwicklung

Die testgetriebene Entwicklung, in der Literatur oft auch als *Test Driven Development* anzutreffen, ist eine agile Methode. Wie in Fig. 3 ersichtlich, wird der während einer Iteration implementierte Code direkt im Anschluss getestet und nicht erst am Schluss nach Fertigstellung des gesamten Systems. Auf diese Weise können gravierende Fehler im Design schnell erkannt und in den folgenden Iterationen entsprechend reagiert werden.
Die testgetriebene Entwicklung geht jedoch noch einen Schritt weiter und legt fest, dass zuerst die Testfälle entwickelt werden, welche die zu entwickelnde

Softwarekomponente zu erfüllen hat und erst im Anschluss die Komponente selbst. Dies erhöht die Qualität der Software zusätzlich, da sich der Programmierer im Voraus Gedanken über kritische Szenarien machen muss und bereits während der Entwicklung der Testfälle über entsprechende Lösungen nachdenken kann. Ausserdem zwingt ihn dieses Vorgehen dazu die Spezifikation oder das *Interface* der Komponente festzulegen. Bei der Verwendung von Java werden für das Testen einzelner Komponenten oft sogenannte *Unit-Tests* mittels dem frei erhältlichen *JUnit-Framework* eingesetzt. In diesem Zusammenhang hat sich auch das erneute nächtliche Durchführen der definierten Testfälle durchgesetzt, was den Entwicklern erlaubt, Fehler im System am nächsten Arbeitstag identifizieren zu können.

2.8. Releases

Ein weiterer sehr zentraler Aspekt hinter agiler Softwareentwicklung ist der Miteinbezug des Kunden in den gesamten Entwicklungsprozess, also nicht erst zu Beginn und Ende wie beim Wasserfallmodell. Dadurch wird versucht, die genauen Bedürfnisse des Kunden fortlaufend zu ermitteln und teure Fehlentwicklungen zu vermeiden. Um ein stetiges Feedback vom Kunden erhalten zu können, sind natürlich regelmässige *Releases* nötig. Ein *Release* ist eine lauffähige Version des bereits implementierten Systems. Zu Beginn einer Iteration werden typischerweise die Funktionen festgelegt, welche am Ende des Durchlaufs mittels eines neuen *Releases* dem Kunden präsentiert werden können. Das System wird also kontinuierlich von Iteration zu Iteration wachsen und der Fortschritt mit Hilfe von *Releases* für den Kunden sichtbar gemacht. Auf diese Weise wird es möglich, den ganzen Entwicklungsprozess kundenorientierter zu gestalten und dessen Änderungswünsche in der nächsten Iteration berücksichtigen zu können.

2.9. Kritikpunkte

Agile Prozesse sind sehr kommunikationsintensiv. Dies aufgrund der Tatsache, dass keine Analyse- und Designphasen existieren, welche alle Einzelheiten des Vorgehens im Detail festlegen. Dadurch werden die Entwickler gezwungen, sich zum Grossteil selbst zu organisieren und beim Auftreten von Unklarheiten den Kontakt untereinander zu suchen. Dies limitiert die Grösse von Projekten, welche noch problemlos mittels agiler Prozesse realisiert werden können. Denn mit wachsender Anzahl Mitarbeiter, steigt auch der Kommunikationsaufwand, was eine entsprechende Koordination nötig macht. Diesem Nachteil könnte jedoch durch das Unterteilen des Projektes in kleinere Unterprojekte begegnet werden.
Eine weitere Schwierigkeit stellen Kunden dar, deren Verfügbarkeit stark eingeschränkt ist. Der enge Kontakt mit dem Kunden ist zentraler Bestandteil agiler Vorgehen und das Fehlen desselbigen kann sich negativ auf die Qualität und Stossrichtung der Entwicklung auswirken. Da diese nicht regelmässig durch den Kunden evaluiert werden kann.
Ausserdem wird die Eignung agiler Prozesse für Festpreisverträge oft bemängelt, da solche Verträge die genaue Analyse und Spezifikation des Systems voraussetzen, was

wiederum im Konflikt mit dem agilen Ansatz steht. Eine Lösung würde in diesem Fall das Festlegen fixer Preise für zeitlich kürzer gesetzte Meilensteine darstellen.
Zuletzt bleibt noch festzuhalten, dass die Methode der testgetriebenen Entwicklung nicht unproblematisch ist, da sie vom Entwickler eine gewisse Disziplin bei der Programmierung der Testfälle im Voraus fordert. Diese Testfälle sind jedoch entscheidend für den gesamten agilen Prozess. Denn aufgrund der stetig vorgenommenen Änderungen am System sind zuverlässige und möglichst alle Teile des Systems abdeckende Tests absolut erforderlich. Nur so kann sicher gestellt werden, dass durch die realisierten Änderungen die Funktionsweise der anderen Systemkomponenten immer noch gewährleistet ist.

2.10. Fazit

Trotz der zuvor erläuterten möglichen Fallstricke im Umgang mit agilen Prozessen gilt es zu erwähnen, dass diese Art Projekte zu realisieren, zumindest langfristig, klassische Prozesse wie das Wasserfallmodell verdrängen wird. Wobei festzuhalten ist, dass Projekte mit im Voraus bekannten und während der Laufzeit stabilen Anforderungen mittels sequenziellem Vorgehen evt. effizienter und günstiger realisiert werden können, da gewisse *Overheads* des agilen Vorgehens, wie beispielsweise der Kommunikationsaufwand, vermieden werden können. Nichtsdestotrotz stellt diese Art von Projekten in der Softwareentwicklung eher die Ausnahme als die Regel dar. Der Wandel ist in dieser Domäne oftmals die einzige Konstante und auf diesen Zustand weiss der agile Ansatz definitionsbedingt viel besser zu reagieren.

2.11. Agile Prozesse

Zum Abschluss des zweiten Kapitels, sind noch einige der meist verwendeten und bekannten agilen Prozesse zu nennen. Auf jede dieser einzelnen Vorgehen einzugehen, würde den Rahmen dieser Arbeit jedoch sprengen, daher ist es dem Leser überlassen, sich bei Interesse weiter zu informieren.

Folgende Prozesse finden nebst dem im nächsten Kapitel beschriebenen *Scrum* momentan Verwendung (diese Liste erhebt nicht Anspruch auf Vollständigkeit):

- Adaptive Software Development (ASD)
- Crystal
- Dynamic System Development Method (DSDM)
- Extreme Programming (XP)
- Feature Driven Development (FDD)
- Pragmatic Programming
- Software-Expedition
- Universal Application
- Usability Driven Development (UDD)

3. Scrum

3.1. Vorbemerkungen

Im vorhergehenden Kapitel wurde die agile Softwareentwicklung im Generellen dargestellt. Ziel dieses Kapitels ist es nun, einen konkreten Prozess namens *Scrum* kennen zu lernen. Zuerst soll ein grober Überblick über die in diesem Prozess verwendeten Vorgehen und die Terminologie gegeben werden. Anschliessend werden die besonders wichtigen Aspekte von *Scrum* noch einmal im Detail beleuchtet.
Wie schon in der Einleitung beschrieben, sollte nach der Lektüre dieses Kapitels klar sein, ob sich der Einsatz von *Scrum* in einem konkreten Projekt als lohnenswert erweist.

3.2. Überblick

Zu Beginn dieses Kapitels stellt sich die Frage nach dem Ursprung von *Scrum*. Aufgrund welcher Probleme und Bedürfnisse heraus ist *Scrum* entstanden? Zur Klärung dieser Frage, muss man zurück ins Jahr 1996, denn damals wurde *Scrum* zum ersten Mal bei Individual, Inc. durch Ken Schwaber eingesetzt [2].
Er stellte fest, dass die dortige Entwicklungsabteilung zwar fachlich höchst kompetent war, es jedoch nicht fertig brachte die von Management und Kunden geforderten Anpassungen innert vernünftiger Frist vorzunehmen. Nach genauerer Untersuchung dieses Zustands entdeckte er schliesslich die Quelle dieses Problems. Die Entwickler wurden täglich mit so vielen Änderungen konfrontiert, dass sie nicht mehr in der Lage waren, eine Funktion fertig zu implementieren, bevor bereits die Nächste lauffähig sein sollte. Sie konnten also nicht konzentriert und fokussiert an einigen wenigen *Features* arbeiten, da dies durch ständig neue *Inputs* verunmöglicht wurde.
Genau an diesem Punkt setzte Schwaber an und legte in einem ersten Schritt fest, dass alle Änderungswünsche und neuen Funktionen in einer einzigen Liste namens *Product Backlog* zusammengetragen werden. Eine weitere Einschränkung war, dass Anfragen nicht mehr direkt an die einzelnen Entwickler sondern neu zentral an eine einzige Person den sogenannten *Product Owner* zu richten waren. Auf diese Weise konnten sich die Entwickler voll auf ihre Arbeit konzentrieren und wurden nicht mehr ständig von Dritten gestört.
Der *Product Owner* ist zudem für die Priorisierung der Liste verantwortlich, er legt also fest, welche Funktionen demnächst implementiert werden sollen. Die in ein Projekt involvierten Personen werden in *Scrum* in 3 Rollen eingeteilt. Eine Rolle, den *Product Owner* wurde bereits erläutert. Die einzelnen Entwickler werden in der Rolle des Teams zusammengefasst. Die Dritte und letzte Rolle ist schliesslich der *Scrum Master*, dieser ist hauptsächlich für den Erhalt aller benötigten Ressourcen für das Team verantwortlich. Ebenfalls ist er für die Lösung etwaiger Probleme der Entwickler zuständig. Er fungiert also, als eine Art Manager, welcher dem Team bei Problemen und der Beschaffung fehlender Ressourcen zur Verfügung steht und den gesamten Prozess überwacht.

Wie in Kapitel 2 bereits behandelt, werden in der Agilen Softwareentwicklung Iterationen eingesetzt, um die Entwicklung zeitlich zu gliedern. Schwaber führte hierbei im Zusammenhang mit *Scrum* den Begriff *Sprint* ein. Ein *Sprint* ist von der Bedeutung her identisch mit einer Iteration, also einer zeitlich begrenzten Einheit, während der die verschiedenen Disziplinen der Softwareentwicklung durchgeführt werden. Typischerweise liegt die Dauer eines Sprints bei 30 Tagen, wobei diese je nach Bedürfnis angepasst werden kann.
Vor dem Beginn eines neuen *Sprints*, sollte festgelegt werden, welche Funktionen des *Product Backlogs* als nächstes implementiert werden. Dazu treffen sich Team, *Scrum Master*, *Product Owner* und weitere für die Entwicklung verantwortliche Personen zu einem *Meeting* namens *Sprint Planning Meeting*. Während diesem Treffen wird das *Sprint-Goal* festgelegt, welches das Team innerhalb des nächsten *Sprints* zu erreichen hat. Die für das Erreichen des Ziels benötigten Funktionen werden vom *Product Backlog* auf eine neue Liste, den *Sprint Backlog* transferiert. Diese spezielle Liste enthält alle Funktionen, welche während des *Sprints* implementiert werden sollen, im Gegensatz zum *Product Backlog*, welcher die Gesamtheit der noch zu implementierenden Funktionen enthält.
Eine weitere in *Scrum* häufig verwendete Liste ist der *Impediment Backlog*, ihr werden die während eines Sprints angetroffenen Hindernisse hinzugefügt. Der *Scrum Master* ist schliesslich für deren Beseitigung verantwortlich.
Während einem Sprint treffen sich das Team und der *Scrum Master* täglich zum so genannten *Daily Scrum*. Dies ist ein kurzes, zeitlich begrenztes *Meeting* (i.d.R. 15 min.). Je nach dem können auch *Stakeholder* [1] eingeladen werden, um sich über neueste Entwicklungen zu informieren. Laut *Scrum* dürfen diese jedoch nicht in das Gespräch eingreifen, da dies gemäss den von Schwaber gemachten Erfahrungen die *Meetings* unnötig in die Länge zieht und sich durch die Fülle von Vorschlägen und Bemerkungen eher negativ auf den Fokus des Teams auswirkt. Die *Meetings* dienen primär dem Austausch zwischen den Teammitgliedern. Folgende drei Fragestellungen sollten von allen beteiligten Entwicklern während eines *Daily Scrums* beantwortet werden:

- Was wurde seit dem letzten *Meeting* gemacht?
- Was ist bis zum nächsten *Meeting* geplant?
- Welche Probleme behindern die Arbeit?

Schwaber rät zudem, sich bei den *Daily Scrums* ausschliesslich auf diese drei Fragen zu beschränken, um den zeitlichen Rahmen einhalten zu können.
Wie in der Agilen Softwareentwicklung üblich, wird auch bei *Scrum* inkrementweise gearbeitet und nach Ende jedes *Sprints* ein neues *Release* erstellt. Dieses *Release* wird anschliessend in Anwesenheit des *Managements*, *Scrum Masters*, *Product Owners*, der Benutzer und Kunden vom Team präsentiert und von den Teilnehmern evaluiert. Diese Zusammenkunft wird in *Scrum* als *Sprint Review Meeting* bezeichnet und dient dem Aufzeigen des Verlaufs und Fortschritts des Projekts gegenüber den *Stakeholdern*. Zudem kann dadurch auf eventuelle Fehlentwicklungen und Verbesserungsvorschläge eingegangen werden.

[1] Mitglied einer Interessengruppe, Interessenvertreter

Das Team um *Scrum*-Entwickler Ken Schwaber, bei welchem er den Prozess das erste Mal einsetzte, war übrigens nach Einführung des Prozesses in der Lage auf monatlicher Basis *Releases* zu veröffentlichen. Die Masse an Änderungsvorschlägen, welche dem Team anfangs noch im Weg stand, war durch den Einsatz der gerade beschriebenen Werkzeuge erst zu bewältigen. Nach einer Durststrecke von neun Monaten ohne ein einziges *Release* waren die Entwickler nach Einführung von *Scrum* in der Lage innerhalb eines Monats ein lauffähiges *Release* zu veröffentlichen.
Abschliessend sollte noch erwähnt werden, dass *Scrum* als Agiler Prozess dem Team eine hohe Eigenverantwortung zuspricht. Das Team kann nämlich innerhalb eines *Sprints* selber entscheiden, wie es sich organisiert, um die gesetzten Ziele zu erreichen und welche Methoden es zur Erreichung des Ziels verwendet. Auch der in Analyse, Design, Testing und Dokumentation investierte Aufwand wird vom Team bestimmt. Durch diese Praxis ist *Scrum* der Definition entsprechend sehr agil und versucht, mit einem Minimum an Management und Bürokratie ein Maximum an Produktivität zu erreichen.

3.3. Anwendungsszenario

Nachdem nun die in *Scrum* verwendeten Begriffe geklärt sind und ein Überblick der einzelnen Funktionen gegeben wurde, wird in diesem Abschnitt das Vorgehen von *Scrum* anhand eines Anwendungsbeispiels verdeutlicht. Dies soll helfen die einzelnen Begriffe im Gesamtkontext zu sehen und zum besseren Verständnis des Prozesses beitragen.
Wie könnte nun ein konkretes Anwendungsszenario für *Scrum* aussehen?
Zu Beginn des Projekts treffen sich Team, *Scrum Master* und Kunden zu einem ersten *Sprint Planning Meeting* um eine Liste der zu implementierenden Funktionen zusammenzustellen, den sogenannten *Product Backlog.* Anschliessend wird die Person bestimmt, welche als *Product Owner* für die Priorisierung der Liste verantwortlich sein wird. Nachdem die Liste geordnet wurde und die wichtigsten Funktionen feststehen, kann das *Sprint-Goal* bestimmt werden und die zur Erreichung dieses Ziels benötigten Funktionen vom *Product Backlog* in den *Sprint Backlog* übertragen werden.
Nach diesen Schritten kann der erste *Sprint* begonnen werden. Während der Entwicklung, in welcher sich das Team stets selbst organisiert, wird es auf neue Funktionen treffen, welche für das Erreichen des *Sprint-Goals* nötig sind und diese dem *Sprint Backlog* hinzufügen. Begegnet das Team Hindernissen, welche es nicht selber beseitigen kann, so werden diese im *Impediment Backlog* festgehalten. Diese Liste steht unter Verantwortung des *Scrum Masters*, welcher für die Lösung der Probleme verantwortlich ist.
Um den täglichen Fortschritt bewerten zu können und Informationen untereinander auszutauschen, trifft sich das Team mit dem *Scrum Master* anlässlich des *Daily Scrums*. Nach Ende des *Sprints* treffen sich alle Projektbeteiligten zum *Sprint Review Meeting*, um das erstellte *Release* zu bewerten, dem Team ein Feedback zu geben und eventuelle Änderungswünsche anzubringen.
Nun beginnt der Zyklus mit einem *Sprint Planning Meeting* von Neuem und wird solange vorgesetzt bis der *Product Backlog* abgearbeitet ist oder der Kunde mit dem

präsentierten *Release* zufrieden ist. In Fig. 4 wird eine vereinfachte Form des gesamten *Scrum*-Prozesses dargestellt.
Das Fundament zum Verständnis von *Scrum* sollte nun gelegt sein, in den folgenden Abschnitten werden die wichtigsten Aspekte noch einmal im Detail erläutert, um fundiertere Kenntnisse zu schaffen.

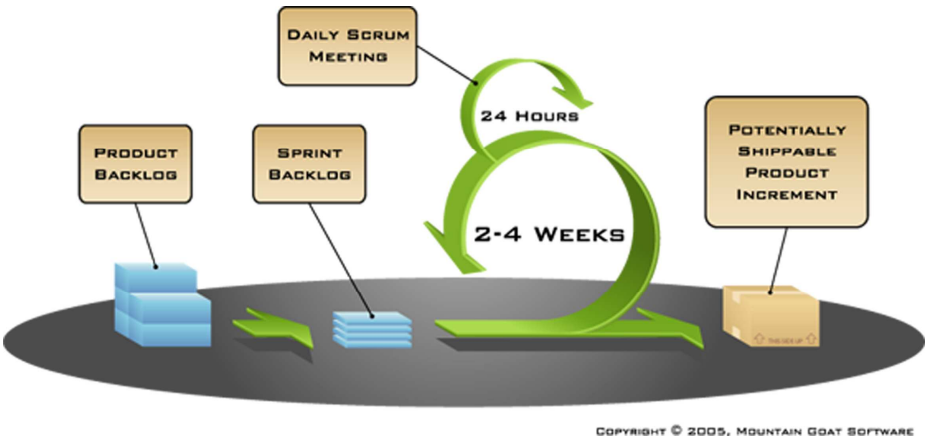

Fig. 4. Grafische Darstellung des *Scrum*-Prozesses
(von http://www.denkwerft.de)

3.4. Rollen

3.4.1. Vorbemerkungen

In *Scrum* übernimmt jeder Mitarbeiter eine von drei Rollen. Diese Rollen sind klar definiert. Jeder Rolle werden zu erledigende und zu unterlassende Aufgaben zugeteilt. Das ist ein Grundprinzip von *Scrum*. In diesem Abschnitt werden die drei Rollen vorgestellt und es wird erklärt, was in den Aufgabenbereich einer Rolle fällt und was ausdrücklich nicht.

3.4.2. Das Team

Die primäre Aufgabe des Teams ist es, das Produkt (die Software) zu entwickeln und zu testen. Wegen dieser zentralen Rolle ist es bei so gut wie jeder Sitzung anwesend. Ihre Entscheidungen und ihr Feedback bestimmen massgeblich den Verlauf des Projekts und schlussendlich über Erfolg oder Misserfolg. Das Team setzt sich

meistens aus verschiedensten Mitgliedern zusammen, die gemeinsam das ganze Projekt realisieren können (Designer, Entwickler, …). Meist besteht ein *Scrum* Team aus fünf bis neun Personen wobei sieben das Optimum darstellt.

Das besondere an einem *Scrum* Team ist, dass es selbstorganisiert ist. Es gibt keinen externen Teamleiter oder ähnliches. Vorgehensweisen werden nicht von aussen vorgegeben, sondern innerhalb des Teams bestimmt. Auch die anderen beiden *Scrum* Rollen haben nicht die Aufgabe das Team zu organisieren oder bestimmte Methoden vorzuschreiben. Es wird davon ausgegangen, dass das Team von sich aus eine optimale Organisation bildet. Im Gegensatz zu anderen Prozessen werden Praktiken wie *Pair Programming* weder vorgeschrieben noch verboten. Das Team selber kann entscheiden, wie es vorgeht.

Neben der Selbstorganisation hat das Team auch die Verantwortung darüber, wie viele und welche Features während eines *Sprints* (Iteration) implementiert werden. Auch hier entscheidet das Team selber. Es bekommt vom *Product Owner* die noch zu implementierenden Features und wählt daraus diejenigen aus, welche im nächsten *Sprint* bearbeitet werden.
Das Team ist zusätzlich dafür zuständig, die Aufwandsschätzung zu betreiben. Wenn der *Product Owner* ein neues Feature in den *Product Backlog* aufnehmen will, so lässt er den Aufwand durch das Team schätzen. Entscheidet sich das Team dann in einem *Sprint* das Feature zu implementieren, so wird der Aufwand möglicherweise neu geschätzt, da man in der Zwischenzeit möglicherweise neue Erkenntnisse gesammelt hat. Auch zur Aufwandsschätzung schreibt *Scrum* keine bestimmte Methode vor.

Zu den Aufgaben des Teams gehört:

- Implementieren und Testen der Features
- Selbstorganisation
- Aufwandschätzung für Features und Aktivitäten

<u>Nicht</u> zu den Aufgaben des Teams gehört:

- Festlegen der Priorität der Features
- Kapital zur Verfügung stellen usw.

3.4.3. Der Product Owner

Die Aufgabe des *Product Owners* ist das Vertreten der Interessen der sogenannten *Stakeholder*. Dies sind alle Personen mit einem Interesse am Projekt wie z.B. Kunden oder Geldgeber. Er stellt letztlich sicher, dass das Produkt den Vorstellungen der Kunden entspricht. Ausserdem ist er für die Einhaltung des Projektbudgets verantwortlich.

Der *Product Owner* definiert die Features, die das fertige Produkt enthalten soll und fügt diese dem so genannten *Product Backlog* hinzu. Vorschläge für neue Features können von jedem kommen, der *Product Backlog* wird aber ausschliesslich vom *Product Owner* bearbeitet. Er ordnet die Features nach ihrer Priorität und präsentiert vor jedem *Sprint* die wichtigsten Features. Aus diesen wählt das Team dann die aus, die im nächsten *Sprint* zu erledigen sind. Durch die Priorität der Features gibt der *Product Owner* grob die Richtung vor, in die weiterentwickelt wird. Der *Product Owner* schreibt dem Team aber nicht vor, wie viel es im nächsten *Sprint* zu erledigen hat. Vielmehr steht er dem Team bei der Auswahl beratend zur Seite. Der *Product Owner* behält stets das Gesamtziel im Auge und passt gegebenenfalls nach Absprache mit den *Stakeholdern* den Zeitplan oder das Budget an.

Der *Product Owner* ist der direkte Ansprechpartner für die Kunden sowie für die Teammitglieder. Sollten sich die Vorgaben der Kunden ändern, so passt der *Product Owner* den *Product Backlog* an. Er ändert gegebenenfalls die Priorität einiger Features oder fügt neue Anforderungen hinzu. Auf diese Weise kann ohne weitere Komplikationen auf veränderte Kundenwünsche eingegangen werden. Das Team wird dabei in seinem aktuellen *Sprint* nicht gestört.

Zu den Aufgaben des Product Owner gehört:

- Verwalten des *Product Backlog*
- Die Einträge des *Product Backlog* nach Priorität ordnen
- Präsentieren der aktuell wichtigsten Einträge des *Product Backlog* vor jedem *Sprint*
- Überwachen der Ziele des Projekts (Qualität, Budget, …)
- Allgemeines Vertreten der Interessen der Kunden

Nicht zu den Aufgaben des Product Owner gehört:

- Verteilen von konkreten Aufgaben an Teammitglieder
- Vorschreiben von gewissen Arbeitsmethoden
- Das Team im klassischen Sinne führen
- Bestimmen der zu erledigenden Arbeiten während eines *Sprints*

3.4.4. Der Scrum Master

Die Aufgabe des *Scrum Masters* ist es, den *Scrum*-Prozess zu überwachen und sicherzustellen, dass das Team so produktiv wie möglich arbeiten kann. Er überwacht die Einhaltung der Rollenverteilung inklusive der jeweiligen Rechte und Pflichten.

Der *Scrum Master* stellt sicher, dass der *Product Owner* nicht seine Kompetenzen überschreitet, indem er sich zum Beispiel zu sehr in die Organisation des Teams einmischt. Er beobachtet den Prozess und versucht ständig, diesen zu verbessern.

Ausserdem leitet der *Scrum Master* die verschiedenen Sitzungen, allen voran die *Daily Scrums.* Er stellt sicher, dass sich alle Teilnehmer an die Regeln halten.

Der *Scrum Master* ist kein direktes Mitglied des Teams und ist auch nicht der *Product Owner* oder ein Projektleiter im klassischen Sinne. Er ist auch nicht ein Vermittler zwischen *Product Owner* und Team, da diese direkten Kontakt pflegen sollten.
Vielmehr versucht der *Scrum Master* die Bedingungen für das Team so optimal wie möglich zu halten. Dazu meldet das Team dem *Scrum Master* fortlaufend Hindernisse, welche dieser gemeinsam mit dem Team aus dem Weg zu räumen versucht. Das Ziel des *Scrum Masters* ist eine produktive Arbeitsumgebung mit zufriedenen Teammitgliedern zu schaffen, da dies letztlich zu einem besseren Produkt führt.

Da der *Scrum Master* die Einhaltung des *Scrum*-Prozesses überwacht, muss er diesen sehr gut kennen. Er kann seine Erfahrung mit *Scrum* dann an unerfahrene Teammitglieder oder *Product Owner* weitergeben. Ausserdem sollte der *Scrum Master* fähig sein, möglicherweise skeptische Teammitglieder zu überzeugen. Oftmals entscheiden die Fähigkeiten des *Scrum Masters* darüber, ob *Scrum* von den Mitarbeitern und dem Management akzeptiert wird oder nicht.

Zu den Aufgaben des Scrum Masters gehört:

- Überwachen des *Scrum*-Prozesses
- Einhaltung der Rollenverteilung sicherstellen
- Hindernisse für das Team aus dem Weg räumen
- Produktives Arbeitsklima schaffen

Nicht zu den Aufgaben des Scrum Masters gehört:

- Zwischen Team und *Product Owner* vermitteln
- Einmischen in inhaltliche Fragen (Features usw.)
- Das Team im klassischen Sinne führen

3.5. Dokumente

3.5.1. Vorbemerkungen

In diesem Abschnitt werden die verschiedenen Dokumente näher erläutert, die für den *Scrum*-Prozess notwendig sind. Im Vergleich zu anderen Prozessen schreibt *Scrum* nur sehr wenige Dokumente vor, wobei es dem Team natürlich frei steht, ergänzende Dokumente zu führen. Obwohl *Scrum* den Schwerpunkt nicht auf das Führen von Dokumenten legt, sind diese für den reibungslosen Ablauf des Projekts von zentraler Bedeutung.

3.5.2. Product Backlog

Das *Product Backlog* ist die Liste aller bekannten Features und Eigenschaften, die das fertige Produkt besitzen sollte, die aber noch entwickelt bzw. bearbeitet werden müssen. Die Einträge sind nach Priorität geordnet und mit einer Aufwandschätzung ergänzt.

In welcher Form die Einträge in das *Product Backlog* aufgenommen werden, wird nicht von *Scrum* vorgeschrieben. Häufig werden direkt programmiertechnische Aufgaben in das *Product Backlog* aufgenommen. Diese sind für die *Stakeholder* möglicherweise nicht direkt verständlich. In vielen Fällen werden die Einträge des *Product Backlog* in Form von so genannten *User Stories* verfasst. Dies sind kurze „Geschichten" aus Sicht des Benutzers, welche die Anforderungen an das fertige Produkt beschreiben. *User Stories* kommen ursprünglich aus dem agilen Prozess *Extreme Programming* und sind meist ein bis zwei Sätze lang. Hier einige Beispiele von *Product Backlog* Einträgen:

- „EventManager Klasse persistent machen" (technisch)
- „Wenn der Anwender die Anwendung öffnet, erscheint eine Liste von Buchungen, die noch bearbeitet werden müssen." (*User Story*)

Der *Product Owner* ist für das *Product Backlog* verantwortlich. Er ordnet die Features nach ihrer momentanen Priorität. Die Features, die dem *Product Owner* momentan am dringendsten erscheinen, werden höher in der Liste platziert. Die am höchsten priorisierten Listeneinträge werden beim nächsten *Sprint* wahrscheinlich bearbeitet. Dadurch steuert der *Product Owner* die Richtung, in welche sich das Projekt entwickelt.

Neue Einträge in das *Product Backlog* dürfen nur durch den *Product Owner* getätigt werden. Somit wird verhindert, dass Änderungsvorschläge, neue Features usw. direkt an das Team gerichtet werden. Durch solche Interventionen von Aussen würde das Team in seinem aktuellen *Sprint* gestört und das gesamte Prinzip des *Backlogs* würde dadurch ausgehebelt.
Alle Vorschläge (aus anderen Abteilungen, aus anderen Teams, von den Kunden, von der Geschäftsleitung usw.) sollten direkt beim *Product Owner* vorgebracht werden. Dessen Aufgabe ist es, jeden Vorschlag und jedes Feature in den *Product Backlog* aufzunehmen, auch wenn es nicht relevant für das aktuelle Projekt erscheint. Dadurch, dass jeder Eintrag des *Product Backlogs* eine Priorität erhält, werden irrelevante Features automatisch tief eingeordnet und deswegen erst spät oder womöglich gar nicht implementiert.

Für die Aufwandsschätzung der einzelnen Einträge in das *Product Backlog* befragt der *Product Owner* das Team. Dieses erarbeitet eine grobe Schätzung, die in das *Product Backlog* eingetragen wird. Der geschätzte Zeitaufwand eines Eintrags kann auch seine Priorität beeinflussen, da ein Feature dessen Implementierung lange dauert auch ein grösseres Risiko birgt. Durch eine höhere Priorität dieses Features kann das

Risiko eher abgeschätzt werden. Wird der Eintrag dann in konkrete Aufgaben für den *Sprint* übersetzt, wird deren Aufwand erneut geschätzt.

Das *Product Backlog* ist ein sehr dynamisches Dokument. Kein Eintrag darin ist in Stein gemeisselt. Ständig kommen neue Einträge dazu und bereits implementierte Features können entfernt werden. Die Priorität für einen Eintrag kann im Verlauf des Projekts stark variieren. So ist es gut möglich, dass ein Feature, welches zu Beginn noch essentiell erschien, schlussendlich nicht im fertigen Produkt vorhanden ist, da man während des Projekts andere Features als wichtiger erkannt hat.
Auch die Zeitaufwandsschätzung ist einem ständigen Wandel unterworfen. Da das Team im Verlauf des Projekts immer mehr Erfahrungen mit dem Produkt und den eingesetzten Technologien sammelt, können immer exaktere Aufwandschätzungen getätigt werden. Zudem brauchen die meisten Teammitglieder eine gewisse Zeit, um zu lernen sich selber und das eigene Arbeitstempo richtig einzuschätzen.

Innerhalb des *Product Backlogs* unterscheidet sich der Detailgrad der Einträge teilweise massiv. Die Einträge mit hoher Priorität werden so detailliert wie möglich beschrieben, damit sie direkt in Aufgaben für den *Sprint* umgewandelt werden können. Sollte eine genaue Beschreibung wegen mangelnder Information oder Erfahrung noch nicht möglich sein, ist dies ein Anzeichen dafür, dass zunächst andere Einträge bearbeitet werden sollten. Auch die Schätzung des Zeitaufwands sollte bei Einträgen mit hoher Priorität so exakt wie möglich sein, damit eingeschätzt werden kann, wie viele Einträge während des nächsten *Sprint*s bearbeitet werden können.

Die Einträge mit niedriger Priorität hingegen, haben noch keine genaue Beschreibung, da möglicherweise viele Entscheidungen von Ergebnissen zukünftiger *Sprint*s abhängen. Diese Einträge werden in der Regel nicht im nächsten *Sprint* bearbeitet, weswegen eine detaillierte Beschreibung und eine exakte Einschätzung des Aufwands nicht nötig sind. Oftmals bestehen diese Einträge nur aus einigen Stichworten, die sich erst im Verlauf des Projekts zu einem richtig ausformulierten Feature entwickeln. Natürlich ist es auch möglich, dass diese Einträge niemals implementiert werden, da andere Einträge als wichtiger eingeschätzt werden.

In *Scrum* ist es nicht notwendig, vor Beginn des eigentlichen Projekts ausgedehnt die finalen Anforderungen an das Produkt zu ermitteln, um sie in das *Product Backlog* zu überführen. Meistens reicht es aus, zu Beginn die offensichtlichen Anforderungen in das *Product Backlog* aufzunehmen, da man ja nur genug Einträge für einen *Sprint* benötigt. Die weiteren Features werden während des ersten *Sprint*s klarer und auch von Aussen (Kunden usw.) werden wahrscheinlich neue Anforderungen gestellt.

3.5.3. Sprint Backlog

Das *Sprint Backlog* enthält alle Aufgaben, die während eines *Sprints* zum Erreichen des *Sprint-Goals* erledigt werden müssen. Ähnlich wie beim *Product Backlog* ist jeder Eintrag mit einer Aufwandsschätzung versehen.

Die Einträge des *Sprint Backlogs* leiten sich direkt von den Features und Anforderungen ab, die im *Product Backlog* definiert wurden. Die Einträge können jedoch nicht direkt übernommen werden. Während die Einträge im *Product Backlog* eine komplette Funktionalität beschreiben, die das fertige Produkt enthalten soll, beschreiben die Einträge im *Sprint Backlog* die Arbeiten, die erledigt werden müssen, um die Funktionalität zu implementieren. Das bedeutet, dass die Einträge des *Product Backlog* in sinnvolle Aufgaben unterteilt werden müssen, bevor sie in das *Sprint Backlog* eingetragen werden können.

Da es sich bei den Einträgen wirklich um einzelne Aufgaben handeln sollte, muss darauf geachtet werden, dass alle Aufgaben zwischen vier und sechzehn Stunden Arbeit ergeben. Falls eine Aufgabe auf mehr als sechzehn Stunden Arbeit geschätzt wird, sollte sie in kleinere Unteraufgaben unterteilt werden.

Auch das *Sprint Backlog* ist kein statisches Dokument, welches am Anfang des *Sprint*s verfasst wird und sich dann nicht mehr ändert. Es ist ständigen Änderungen und Aktualisierungen unterworfen. Da das Team ständig neue Erkenntnisse sammelt, kann es immer besser einschätzen, wie lange eine bestimmte Arbeit dauern wird. Während des *Sprint*s sollten die Schätzungen der verbleibenden Zeit laufend aktualisiert werden. Zudem können während des *Sprints* zusätzliche Aufgaben erkannt werden, die nötig sind, um das Ziel des *Sprint*s zu erreichen. Diese können dann hinzugefügt werden.

3.5.4. Burndown Chart

Es ist oft nicht einfach während eines *Sprint*s abzuschätzen, ob man auf gutem Wege ist, das aktuelle *Sprint-Goal* zu erreichen oder nicht. Zum Einen beruhen die Aufwandsangaben auf Schätzungen, die vor allem zu Beginn des Projekts nicht besonders präzise sind. Zum Anderen können neue Aufgaben auftauchen, die nötig sind um das *Sprint-Goal* zu erreichen. Es ist auch denkbar, dass gewisse Aufgaben aufgrund von neuen Erkenntnissen nicht mehr nötig sind.

Um den Überblick zu bewahren, wird der *Sprint Backlog* laufend aktualisiert. Dieser bietet zwar einige Informationen ist aber einerseits nicht besonders anschaulich und bietet andererseits keinen Überblick über den Verlauf des *Sprints*. Zu diesem Zweck wurde der *Burndown Chart* eingeführt. Beim *Burndown Chart* handelt es sich um einen Graphen, der auf der X-Achse die verbleibende Arbeitszeit und auf der Y-Achse die verbleibende zu erledigende Arbeit des aktuellen *Sprint*s darstellt. Der *Burndown Chart* gibt also für jeden Punkt auf der Zeitachse die verbleibende Arbeit an. Von dieser Graphik kann jederzeit der aktuelle Stand abgelesen werden. Ziel ist, dass am Ende des *Sprint*s die verbleibende Arbeit bei 0 Stunden steht.

Fig. 5. *Burndown Chart*
(von http://www.mountaingoatsoftware.com)

Der *Burndown Chart* hilft dem Team dabei, die verbleibende Arbeit im Blick zu behalten. Abweichungen vom Zeitplan können mit Hilfe dieser Grafik schnell erkannt werden und gegebenenfalls Gegenmassnahmen ergriffen werden. Unter Umständen müssen Aufgaben verändert werden oder neue Aufgaben hinzugefügt werden. Diese Massnahmen werden ebenfalls als „Sprung" in der Kurve sichtbar. Nach dem *Sprint* kann die Wirksamkeit solcher Massnahmen am *Burndown Chart* abgelesen werden.

Der *Product Owner* kann sich durch den *Burndown Chart* vergewissern, dass der *Sprint* wie geplant abläuft. Daraus kann er Schlüsse für andere Einträge im *Product Backlog* ziehen. Durch das alleinige Analysieren des *Sprint Backlog* wäre dies nicht möglich. Das Studieren von *Burndown Charts* vergangener *Sprint*s, erlaubt dem Team die Präzision seiner Schätzungen zu verbessern. Wenn verschiedene Aufgaben über- oder unterschätzt wurden, ist dies im *Burndown Chart* gut erkennbar.

Aus dem *Burndown Chart* kann ein erfahrener *Scrum Master* viele Schlüsse ziehen. Oftmals können gewisse Probleme bereits anhand des *Burndown Charts* erkannt werden. Der *Scrum Master* kann dann versuchen, diesen Problemen entgegenzuwirken und den Erfolg seiner Massnahmen wenige Tage später anhand des *Burndown Charts* ablesen. Natürlich sollte der *Scrum Master* nicht nur die *Burndown Charts* studieren sondern auch direkten Kontakt mit dem Team pflegen. Die Charts können aber ein äusserst nützliches Hilfsmittel darstellen.
Der *Burndown Chart* des vergangenen *Sprint*s kann während des *Sprint Reviews* von grossem Nutzen sein. Mit dieser Kurve lässt sich der Verlauf des Projekts anschaulich

darstellen und Probleme sind meist gut in der Kurve erkennbar. Der *Burndown Chart* erleichtert auf diese Weise die Kommunikation zwischen den einzelnen Personen.

3.5.5. Impediment Backlog

Die Hauptaufgabe des *Scrum Masters* ist das Beseitigen von Hindernissen, welche die Teammitglieder in ihrer Produktivität einschränken. Dabei kann es sich um technische Hemmnisse wie z.B. fehlende Hard- oder Software handeln oder auch um andere Probleme wie z.B. mangelndes Fachwissen oder unvollständige Informationen über eine Anforderung oder eine Aufgabe. Diese Hindernisse werden vom Team während des *Daily Scrum* angesprochen und der *Scrum Master* bemüht sich, diese zu entfernen.

Es empfiehlt sich, diese Hindernisse in eine öffentliche Liste aufzunehmen, dem sogenannten *Impediment Backlog*. Diese Liste enthält alle Hindernisse, die der *Scrum Master* noch nicht beseitigt hat. Das Führen der Hindernisse in einer öffentlichen Liste bietet einige Vorteile. Zum Einen fühlen sich die Teammitglieder eher ernst genommen, wenn ihre Probleme schriftlich in ein Dokument einfliessen. Probleme die im *Impediment Backlog* stehen, können nicht einfach ignoriert werden. Zum Andern kann der *Scrum Master* die wichtigsten Probleme jederzeit überblicken und so sein weiteres Vorgehen planen. Möglicherweise erkennt der *Scrum Master* anhand der Einträge zusätzliche Probleme oder Ursachen für die vorhandenen Hindernisse. Diese können dann gemeinsam mit dem Team aus der Welt geschaffen werden.

Wenn Einträge aus dem *Impediment Backlog* gestrichen werden können, wird es für die Teammitglieder ersichtlich, dass alles Nötige unternommen wird, um ihnen ein produktives Arbeitsumfeld zu bieten. Dies führt meistens zu mehr Freude an der Arbeit, was wiederum mit einer Produktivitätssteigerung einhergeht.

4. Schlussfolgerung

Im Laufe dieser Arbeit wurde die Problematik häufig ändernder Anforderungen mehrmals erwähnt. Erfahrungen mit klassischen Prozessen, wie dem sequentiellen Wasserfallmodell, haben gezeigt, dass durch den starken Wandel dieser Anforderungen eine komplette Analyse- und Designphase im Voraus meist nicht sinnvoll ist. Denn nicht selten weiss der Kunde zu Beginn eines Projektes nicht genau, welche Funktionalität er von der Software erwartet. Wird dann auf der Basis ungenauer oder sogar falscher Annahmen das Produkt anhand festgelegter Phasen entwickelt, so ist die Frustration am Schluss auf beiden Seiten entsprechend gross.
Diese Erkenntnisse führten schliesslich zum Entwurf des agilen Manifests und der daraus resultierenden agilen Prozesse. Durch das iterative Vorgehen und den engen Kundenkontakt kann den eben erwähnten Schwierigkeiten begegnet werden. Entspricht ein Inkrement nicht den Vorstellungen des Kunden, so geht im schlimmsten Falle die Arbeit einer Iteration verloren und nicht wie beim

Wasserfallmodell der Grossteil des Projekts.
Scrum hat sich durch seine wenigen, aber strikt umgesetzten Regeln, als sehr interessante agile Vorgehensweise herausgestellt. Durch das Setzen des Fokus auf das zu entwickelnde Produkt und nicht auf dessen vollständige Planung und Dokumentation, werden die Mitarbeiter motiviert und zeigen entsprechend bessere Leistungen. Hervorzuheben ist zudem die Bedeutsamkeit psychologischer Aspekte in *Scrum*. Wie zum Beispiel die dem Team zugesprochene Selbstorganisation oder auch die durch regelmässige, lauffähige *Releases* ausgelösten Erfolgserlebnisse.
Es gilt jedoch auch einige Schwierigkeiten im Umgang mit *Scrum* zu beachten. Der Prozess der Selbstorganisation könnte beispielsweise durch dominante Teammitglieder oder *Product Owner* gestört werden und auch die von *Scrum* geforderte Eigenverantwortung ist nicht allen Mitarbeitern gegeben. Weiter muss beim Einsatz von *Scrum* bedacht werden, dass zwei Mitarbeiter, namentlich der *Scrum Master* und *Product Owner* als Prozess-*Overhead* nicht direkt zur aktiven Entwicklung des Produkts beitragen.
Trotz dieser vermeintlichen Schwächen ist *Scrum* bei der Suche nach einem geeigneten agilen Prozess sicherlich mehr als eine Überlegung wert und kann durch seine unbürokratische und an die Bedürfnisse des Menschen gerichtete Vorgehensweise zu einer erheblichen Produktivitätssteigerung beitragen.

5. Referenzen

1. Larman, C.: Applying UML And Patterns. 24 (2007)
2. Schwaber, K., Beedle, M..: Agile Software Development with Scrum. 2 (2002)