Agile Methoden

Aus OOSEWiki

Bevor auf den Begriff der Agilität eingegangen wird, soll zuerst der Unterschied zwischen schwer- und leichtgewichtigen Prozessmodellen beschrieben werden. Die **Schwer- bzw. Leichtgewichtigkeit eines Softwareentwicklungsprozesses** bezeichnet den Formalisierungsgrad des Prozesses [Pomberger et al. 2004, S. 45]. Schwergewichtige Prozessmodelle bedeuten einen Mehraufwand im Softwareentwicklungsprozess, der sich in der Planung, im Entwurf und in der Dokumentation wieder findet. Für große Projekte mit verteilter Produktentwicklung, die sich über einen längeren Zeitraum erstrecken ist dieser Mehraufwand erforderlich. Jedoch überwiegt er den Softwareentwicklungsprozess bei kleinen bis mittelgroßen Projekten. Dieser Umstand führte zur Einführung von so genannten agilen bzw. leichtgewichtigen Prozessmodellen. Agile Methoden sind schwach formalisiert, flexibel und kennzeichnen sich durch eine iterative Vorgehensweise bei der Spezifikation, Entwicklung und Auslieferung der Software. Grundwerte agiler Methoden sind die Einbeziehung des Kunden in den gesamten Entwicklungsprozess, eine inkrementelle Auslieferung der Software, Offenheit gegenüber Änderungen in der Spezifikation und Vermeidung von Komplexität in der Software selber als auch im Entwicklungsprozess. Die Kommunikation innerhalb des Entwicklerteams und mit dem Kunden spielt hierbei eine große Rolle. Bei der inkrementellen Entwicklung wird auf eine möglichst kurze Zykluszeit und qualitative Entwicklung von einzelnen Komponenten Wert gelegt. Nach der Integration dieser Komponenten in das Gesamtsystem wird ein neues Release an den Kunden ausgeliefert. Das gewonnene Feedback soll dann in den nächsten Entwicklungszyklus einfließen. [Sommerville 2004, S. 396 ff.]
Die Grundwerte agiler Methoden wurden in dem agilen Manifest (siehe Kasten) festgehalten, welches aus einer Konferenz der Agile Alliance im Februar 2001 entstanden ist:

Das agile Manifest
Wir entdecken bessere Wege, Software zu entwickeln, indem wir es selbst tun und anderen dabei helfen. Durch diese Arbeit sind wir zu der folgenden Werteabwägung gelangt:

- *Individuen und Interaktionen sind wichtiger als Prozesse und Werkzeuge*
- *Lauffähige Software ist wichtiger als umfangreiche Dokumentation*
- *Zusammenarbeit mit dem Kunden ist wichtiger als Vertragsverhandlungen*
- *Reaktionen auf Änderungen sind wichtiger als einen Plan zu verfolgen*

Daher messen wir, obwohl die jeweils zweiten Dinge ihren Wert besitzen, den jeweils erstgenannten Dingen den höheren Wert zu.

Alle agilen Methoden zielen also auf eine inkrementelle Entwicklung und Auslieferung von Softwarekomponenten ab, unterscheiden sich aber jeweils in ihrer Ausführung um dieses Ziel zu erreichen.

Inhaltsverzeichnis

- 1 Missverständnisse
  - 1.1 XP ist wie "Just start hacking"
  - 1.2 Es dürfe keine Planung gemacht werden
  - 1.3 Es müsse immer alles getestet werden
- 2 Einzelne Methoden
  - 2.1 Extreme Programming (XP)
  - 2.2 Scrum

- 2.3 Crystal
  - 2.3.1 Crystal Clear
  - 2.3.2 Crystal Orange
- 3 Artikel
- 4 Literatur

Missverständnisse

XP ist wie "Just start hacking"

Nein, dann hat man schon die erste Phase übersprungen: das Testen. Wer mit dem Testen anfängt, wird in seinem Hacker-Drang sofort gebremst.

Es dürfe keine Planung gemacht werden

Jeden Fehler, den man durch gute Planung vermeiden kann, sollte man auch vermeiden. Bei XP geht es nicht darum, jede Planung, jedes Vordenken abzuschalten. Dann würde man nur noch willkürlich drauflos hacken. Das kann nicht gemeint sein.

Die Erfahrung mit Planungs-lastigen Methoden in der Vergangenheit hat jedoch gezeigt, dass häufig Planung schnell in **Spekulation** abgleitet. Viele Dinge werden erst im Laufe des Projektes klar. Bei Unklarheit trotzdem etwas zu Papier zu bringen ist mühsam und kontra-produktiv. Wenn die Spekulationen einmal schwarz-auf-weiß auf Papier stehen, bekommen sie per-se einen Stellenwert und müssen später gelesen werden. Dann setzt die **inverse Spekulation** ein ("Was hat sich der Schreiber dabei gedacht?").

Außerdem ist der Mensch schneller im Wünsche-generieren als im Wünsche-erfüllen. Was nützt eine wunderschöne ToDo-Liste mit 100 Punkten, wenn man nur für die ersten 3 Punkte Zeit hat?

Ein anderer Effekt ist bekannt: Ab einer bestimmten Größenordnung braucht eine Theorie keine Bestätigung in der Praxis mehr. Sie kann aus sich heraus so viele interessante Fragestellungen generieren, dass der eigentliche Zweck keine Bedeutung mehr hat. Eine Planung kann in eine Theorie degenerieren, die eine solche Eigen-Dynamik entwickeln kann. Ebenso braucht man einer Organisation mit mehr als 50 Leuten von außen keine Aufgaben mehr zu geben: Sie beschäftigt sich mit selbst-generierten Aufgaben selber.

Daher zur Orientierung als **Regel:** Alles, was sich ohne jede Spekulation planen lässt, sollte man auch planen, solange dafür nicht max. 10% der Zeit verbraucht wird, wie für die Herstellung des Produkts selber. Planen ja, aber maßvoll.

Es müsse immer alles getestet werden

Es gibt viele Missverständnisse rund um TDD: TDD ist kein 11. Gebot "Du sollst testen!". Man braucht auch kein schlechtes Gewissen zu haben, wenn man nicht testet. Deswegen ist man kein schlechter Programmierer oder gar schlechter Mensch. Es geht auch nicht um Zeitverschwendung durch Testen, sondern um die Zeitersparnis durch automatisiertes Testen. Hat man Regressionstests, so ist man auf der sicheren Seite.

Wer TDD praktiziert, merkt, dass er bessere Arbeit leistet, zufriedener ist und besser einschätzen kann, was er wirklich geschafft hat und was noch fehlt.

Informatiker nehmen Theorien über die Welt oft wichtiger als die Welt selber. Dann sind sie stärker daran interessiert, die eigene Theorie umgesetzt zu sehen. Selbst-Bestätigung und Theorie-Bestätigung gehen dann vor der Realität. Das ist Verkopfung und Distanzierung vom Tatsächlichen. Wer TDD meidet, schwelgt oft in

selbst-geschaffener Komplexität.

Ausgerechnet Einstein sagte: "Man muss die Welt nicht verstehen, man muss sich nur darin zurechtfinden." Wissenschaftlichkeit wird zu oft verstanden als Nachweis, mit hoher Komplexität umgehen zu können. Da wird erst der hoch-komplexe Drachen erschaffen, damit man als Held beweisen kann, dass man den Drachen beherrschen und töten kann. Komplexität ist aber nur die eine Schwinge der Wissenschaftlichkeit. Die andere Schwinge heißt Abstraktion und Einfachheit. Wenn die wissenschaftliche Arbeit aus einem Guss ist, hat sie eine ganz andere Qualität als eine Aufzählung und Lösung vieler komplexer Fälle.

TDD heißt "So sehr bin ich an der Wirklichkeit interessiert, dass ich hier und jetzt wissen will, ob ich das Tor treffe, ob der Algorithmus funktioniert, ob ich das richtig verstanden habe, ob der Code läuft." Nicht als Bestätigung der eigenen Theorie "Ich habe doch Recht!", sondern als unverkopfte, direkte Interaktion.

Einzelne Methoden

Extreme Programming (XP)

Unter den agilen Methoden ist das Extreme Programming, im Folgenden XP, das bekannteste und am weitesten verbreitete Modell. [Sommerville 2004, S. 398]
Der Prozessablauf beim XP besteht aus drei Teilprozessen: Release-Planung, iterative Release-Erstellung und Akzeptanztests mit anschließender Release-Veröffentlichung [Pomberger et al. 2004, S. 46]. Bei diesem Prozessablauf werden Praktiken angewendet, von denen die wichtigsten im Folgenden beschrieben werden. Praktiken beschreiben die Möglichkeiten zur Umsetzung der agilen Grundwerte und können situationsbedingt ausgewählt und angepasst werden. In [Beck 2005, S.36] wird zwischen primären Praktiken und Begleitpraktiken unterschieden. Primäre Praktiken sind unabhängig voneinander und können jederzeit angewendet werden. Dahingegen ist die Anwendung der Begleitpraktiken erst nach erfolgreicher Anwendung der primären Praktiken sinnvoll.

Primäre Praktiken [Beck 2005, S.37 - 53]
Sit Together
Die Entwicklung findet in einem Raum statt, in dem für alle genug Platz ist.

Whole Team
Das Projektteam ist dynamisch und es existiert ein „Wir-Gefühl“. Werden spezielle Kenntnisse häufiger benötigt, dann wird das Team um eine qualifizierte Person mit den erforderlichen Kenntnissen erweitert. Umgekehrt werden Personen, deren Kenntnisse nicht mehr benötigt werden, entlassen oder einer anderen Funktion zugeordnet.

Informative Workspace
An einer zentralen Stelle (z.B. an einer Tafel) wird stets der aktuelle Projektstatus übersichtlich dargestellt, so dass man auf den ersten Blick einen Eindruck vom aktuellen Projektfortschritt erhält und nach näherem Betrachten auch Details entnehmen kann.

Energized Work
Arbeitszeiten dürfen nur so lang sein, so lang es möglich ist produktiv zu arbeiten. Überstunden sollen also vermieden werden und erkrankte Entwickler sollen sich bis zur vollständigen Genesung ausruhen.

Pair Programming Das Team besteht aus Paaren von Entwicklern, die stets zusammen arbeiten und sich gegenseitig überprüfen. Dies betrifft die Analyse, das Design, die Programmierung und die Tests. Das Ziel dabei ist die Steigerung der Qualität und Reduzierung von Fehlern.

Stories
Die „User Stories“ sollen eine umfangreiche Spezifikation ersetzen und stellen Funktionalitäten aus

Anwendersicht dar (vergleichbar mit Use-Cases). Sie werden von den Kunden selbst erstellt und der Entwicklungsaufwand für jede „Story“ wird abgeschätzt. In die Release-Planung werden nur so viele „User Stories“ einbezogen, dass die Zeit bis zur nächsten Release-Veröffentlichung im Rahmen bleibt. Dabei werden die Prioritäten der „User Stories“ vom Kunden festgelegt.

Continuous Integration
Die Integration soll immer sofort nach der Fertigstellung einer Implementierungsaufgabe erfolgen und darf nicht hinaus gezögert werden. Nach der Integration muss das Gesamtsystem alle Tests bestehen. Nur wenn die Tests erfolgreich waren, darf mit der nächsten Aufgabe begonnen werden. Eine durchgehende Integration beugt Problemen mit dem Interface vor und verkürzt die Auslieferungszeit eines Releases.

Test-First Programming
Für jede Implementierungsaufgabe werden zuerst automatisierte Tests erstellt bevor mit der Implementierung begonnen wird. Durch diesen Ansatz könnten z.B. im Fehler im Design schon vor der Implementierung erkannt werden, denn Schwierigkeiten bei der Erstellung von Tests sind Indizien dafür, dass es Probleme im Design gibt. Durch den Test-First-Ansatz wird auch die Komplexität verringert, weil sich dann der Entwickler auf das konzentriert, was implementiert werden muss ohne unnütze Funktionen zu implementieren [Pressman 2005, S. 112].

Incremental Design
Das Design des Systems soll laufend in kleinen Schritten aktualisiert und angepasst werden. Hierzu gehört auch das Refactoring, also die laufende Verbesserung des Quelltextes ohne die Schnittstelle nach außen hin zu ändern. Insbesondere sollen Duplikate, also Codefragmente mit gleicher Logik, entfernt werden. Dies erleichtert die Wartbarkeit des Codes.

Begleitpraktiken [Beck 2005, S.61 - 71]
Real Customer Involvement
Auftraggeber bzw. Anwender sollen in den Softwareentwicklungsprozess einbezogen werden. Dabei steht der Anwender dem Entwicklerteam durchgehend für Fragen bezüglich der Anforderungen an das System zur Verfügung. Damit soll erreicht werden, dass nur vom Kunden gewünschte Funktionalitäten implementiert werden und die durchgeführten Tests reale Anwendungsfälle abdecken.

Incremental Deployment Bei der inkrementellen Entwicklung der Releases soll eine überschaubare Menge an Funktionalitäten pro Release implementiert werden. Dadurch ist die Zeit zwischen zwei Release-Zyklen kürzer. Bei Releases mit umfangreicher Funktionalität dauert die Vorbereitung der Auslieferung länger und ist mehreren Risiken verbunden.

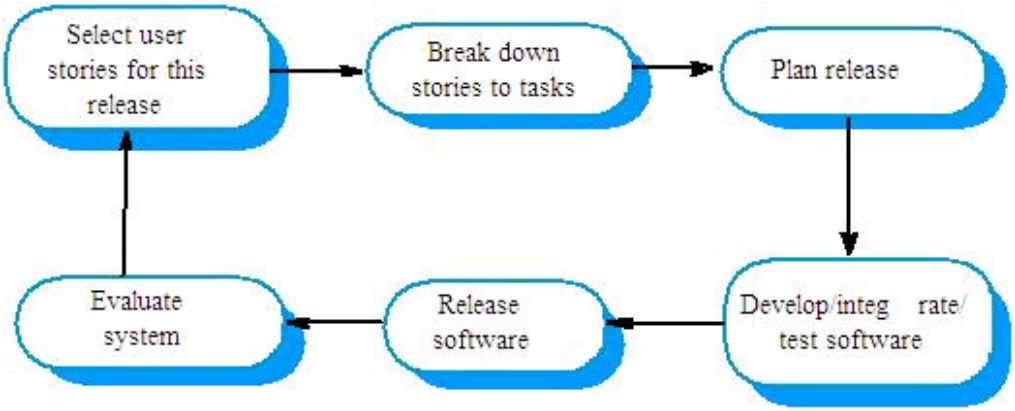
Root-Cause Analysis
Treten nach der Entwicklung in der Software Fehler auf, so sollen diese lokalisiert und behoben werden. Nach der Behebung der Fehler sollen die Ursachen analysiert und Aktionen durchgeführt werden um Fehlern gleicher Art in Zukunft vorzubeugen.

Shared Code
Für diese Praktik wird in der Literatur auch der Begriff „Collective Ownership“ verwendet, d.h. der gesamte Quelltext gehört allen und darf von jedem Entwicklerpaar jederzeit geändert werden.

Negotiated Scope Contract
Eine weitere Empfehlung ist es den Vertrag über die Erstellung des Softwaresystems in mehrere (zwei oder drei) Teilverträge aufzuteilen. Jeder Teilvertrag soll einen genau definierten Bereich im Softwaresystem abdecken, der nach jedem Abschluss des Teilvertrages neu ausgehandelt wird. Dabei haben beide Parteien die Möglichkeit die Entwicklung nicht mehr fortzusetzen.

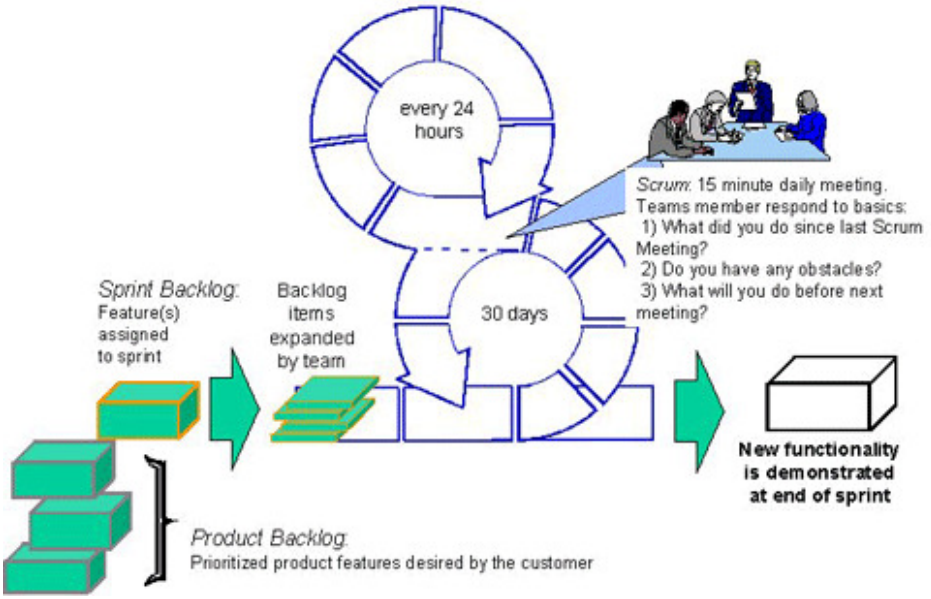
In der unten stehenden Abbildung wird der Release Zyklus dargestellt. In der Planungsphase werden die User Stories erstellt und priorisiert. Die „User Stories“ selbst werden von den Entwicklern in einzelne Implementierungsaufgaben („tasks“) zerlegt. In der nächsten Phase erfolgt die iterative Release-Erstellung. Hier empfiehlt XP den oben beschriebenen Test-First Ansatz und das Pair-Programming zu verwenden. Dabei sollen

während der Erstellung fertige Komponenten laufend in das Gesamtsystem integriert (Continuous Integration) und das Design verbessert werden (Incremental Design). Nachdem die User Stories erfolgreich implementiert wurden, erfolgt die Release-Veröffentlichung. Hier werden mit dem Kunden zusammen die Akzeptanztests durchgeführt, bei das Gesamtsystem überprüft wird.



Scrum

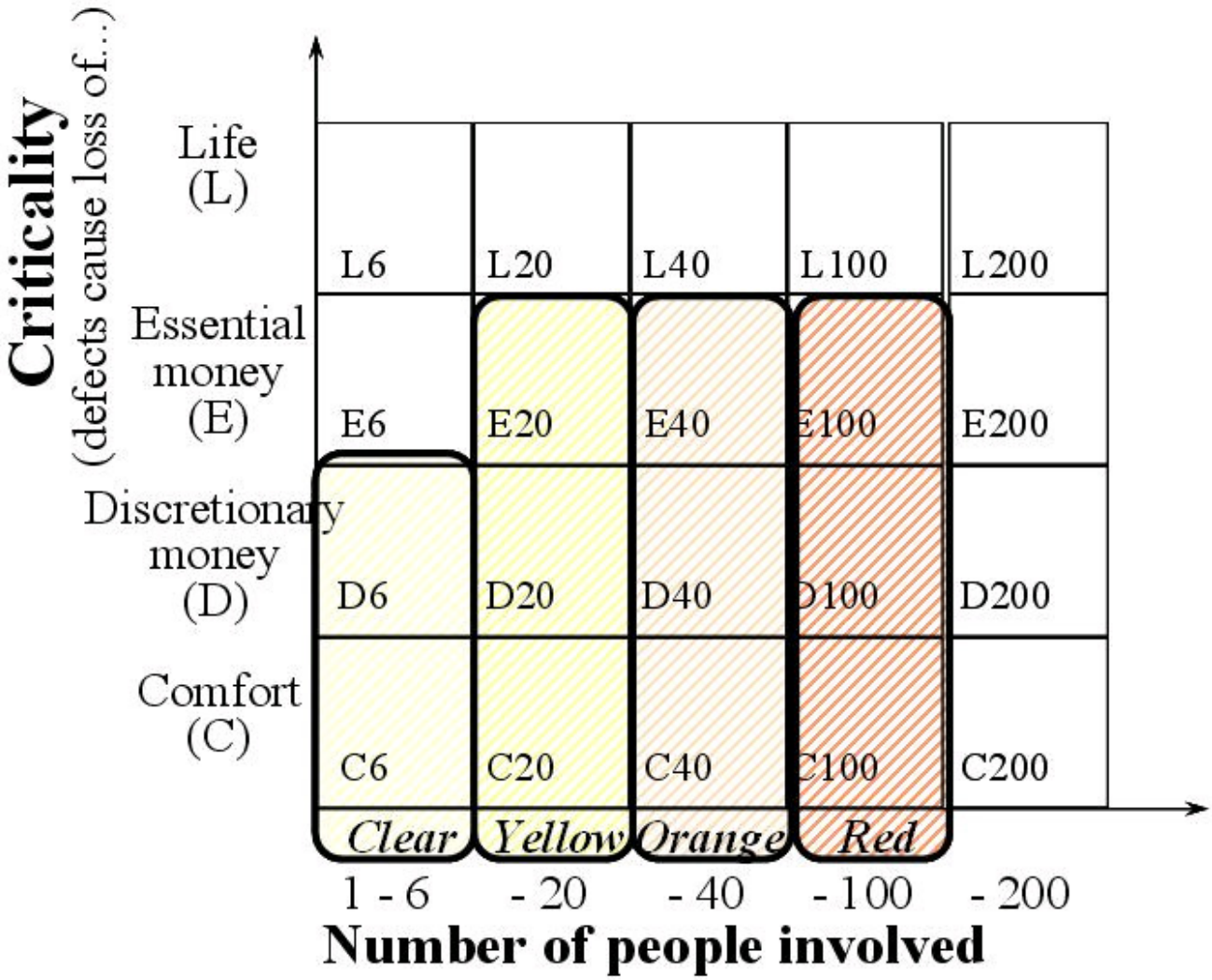
SCRUM wurde ursprünglich Anfang der 90er von Jeff Sutherland entworfen und wird in den letzten Jahren überwiegend von Ken Schwaber und Micheal Beedle weiterentwickelt [Pressman 2005, S. 117]. Bevor auf SCRUM näher eingegangen wird, werden zuerst die Begriffe Mikro und Makro im Zusammenhang mit Prozessmodellen erläutert. Die Komponenten eines Prozessmodells auf der Makroebene legen den Rahmen fest, in der die Entwicklung stattfinden soll. Dies betrifft vor allem die Projektkoordination und die Projektsteuerung, in denen z.B. der zeitliche Ablauf bestimmter Aktivitäten festgelegt wird. Auf der Mikroebene werden Regeln festgelegt, die das zu entwickelnde System selbst betreffen. Dies können z.B. bestimme Regeln beim Entwurf oder bei der Implementierung sein, die das Prozessmodell vorschreibt [Schwaber 1996, S.19].
Bei SCRUM handelt es sich im Gegensatz zum XP nur um ein Managementkonzept für die Steuerung und Koordination von Softwareentwicklungsprojekten (Makroebene). In XP gibt es neben Praktiken für das Management auch viele weitere Praktiken auf der Mikroebene, wie z.B. das Pair Programming, Test-First-Programming und weitere, die oben nicht aufgeführt wurden. SCRUM definiert also keine Praktiken auf der Mikroebene, wodurch sich SCRUM gut mit anderen Methoden kombinieren lässt (z.B. XP oder Crystal Clear). Wie auch beim XP, sind die Grundwerte bei SCRUM ein iterativer Entwicklungsprozess, eine inkrementelle Entwicklung in mehrere Release-Zyklen und die Adaptivität auf sich ändernde Anforderungen. Dabei ist das Ziel, die Software in mehreren Release-Zyklen (in der SCRUM-Terminologie Sprints genannt) mit einer Zykluszeit von maximal einem Monat zu entwickeln. Während eines Sprints finden täglich kurze Meetings (Scrum Meetings) statt, in denen der aktuelle Projektstatus besprochen wird. Die einzelnen Mitarbeiter berichten über ihren Fortschritt, über evtl. auftretende Probleme und über ihren weiteren Arbeitsablauf bis zum nächsten Meeting. Dadurch wird die Kommunikation innerhalb des Teams gefördert und Probleme werden frühzeitig erkannt. Außer den Meetings, sind während eines Sprints keine weiteren Praktiken vorgeschrieben. Nach jedem Sprint muss eine funktionierende Komponente fertig gestellt und in das Gesamtsystem integriert sein, sodass eine Demonstration mit Tests stattfinden kann. In die Demonstration wird sowohl das Management als auch der Kunde einbezogen (siehe Abb.).



Die Aufgaben bzw. die Anforderungen werden in einer Liste, dem Product Backlog, festgehalten. Diese Anforderungen sind nach Prioritäten geordnet und können jederzeit vom Kunden geändert werden. Die Anforderungen, die in einem Sprint implementiert werden, werden in einer weiteren Liste, den Sprint Backlog, festgehalten. Das Sprint Backlog enthält eine Teilmenge der Anforderungen aus dem Product Backlog und wird zu Beginn eines jeden Sprints festgelegt. Im Gegensatz zur Product Backlog ist das Sprint Backlog nicht dynamisch. Es dürfen also keine Änderungen während eines Sprints an dem Sprint Backlog vorgenommen werden [Beedle et al. 1998, S. 2 ff.].

Crystal

Mit der Crystal-Family hat Alistair Cockburn eine Familie von Methoden entworfen mit dem Ziel, für verschiedene Arten von Projekte eine passende Methode bereit zu stellen. Für ein Projekt werden zwei Dimensionen festgelegt: Die Anzahl der erforderlichen Mitarbeiter und die Kritikalität bzw. Hartnäckigkeit des zu entwerfenden Softwaresystems. Dabei klassifiziert Cockburn seine Methodikfamilie nach der Anzahl der Mitarbeiter, indem er für verschiedene Projektgrößen eine Farbe zuordnet. Der Aspekt der Kritikalität wird durch eine feinere und schärfere Anwendung der Praktiken in der jeweiligen Methodik berücksichtigt. Hieraus lassen sich die Methoden Crystal Clear, Crystal Yellow, Crystal Orange, Crystal Red … ableiten. Je größer die Anzahl der Mitarbeiter, desto dunkler die Farbe (siehe Abbildung). Lebenskritische System (Kategorie L) werden wegen unzureichender Erfahrung in der Crystal-Familie nicht berücksichtigt. [Cockburn 2003, S 264 ff.]



In der Crystal-Philosophie betrachtet Cockburn die Software-Entwicklung

> „… als ein nützliches kooperatives Erfinder- und Kommunikationsspiel, dessen Ziel darin besteht nützliche und funktionierende Software zu liefern und dessen zweites Ziel es ist, alles für das nächste Spiel vorzubereiten.“ [Cockburn 2003, S. 266 ff.]

Es handelt sich also um Methoden, in denen Menschen und deren Interaktion im Vordergrund stehen und das Resultat (das fertige Softwaresystem) im Wesentlichen vom Zusammenspiel der Individuen abhängt. Dieser Teil der Philosophie ist auch konsistent mit den agilen Grundwerten. Weiterhin fällt aber auf, dass Cockburn’s Definition auch langlebige Projekte berücksichtigt und somit der Dokumentationsaspekt bei Crystal eine andere Rolle spielt als bei XP. Ein weiterer Aspekt, an den sich Cockburn von den anderen Autoren des agilen Manifests unterscheidet ist das Problem der dynamischen Anforderungen, welches in Crystal nur eingeschränkt berücksichtigt wird. Der Schwerpunkt liegt mehr auf Effizienz und der Einhaltung von festgelegten Projektbudgets [Cockburn 2005, S.17]. In den folgenden zwei Abschnitten werden Crystal Clear, was dem XP sehr ähnelt, und Crystal Orange, welches über leichtgewichtige Prozessmodelle hinausgeht, beschrieben.

Crystal Clear

Crystal Clear ist vorgesehen für kleine Projekte mit einer Kritikalität bis Kategorie D und einer Teamgröße von bis zu 6 Personen, die in einem Raum arbeiten können oder deren Büros sehr nah beieinander sind (vgl. Abb. 12). Kritikalität von D bedeutet, dass Fehler in der Software entweder nur den Komfort beeinträchtigen oder einen relativen finanziellen Verlust verursachen würden. Bei einer Kritikalität von E bedeuten Softwarefehler einen essentiellen Verlust, z.B. die Pleite eines Unternehmens. Für Projekte der Kategorie E8 lässt sich Crystal Clear bei Optimierung der Kommunikationsstrukturen und umfangreichen Tests auch anwenden. [Cockburn 2003, S.268 ff.] Die Praktiken von XP lassen sich im Wesentlichen auf Crystal Clear übertragen ohne die

Crystal-Philosophie zu verletzen, sodass man XP durch Crystal Clear mit einer Einschränkung ersetzen könnte: Während bei XP die personalisierte Wissenssicherung die Dokumentation ersetzt, ist die Dokumentation bei der Crystal-Familie unabdingbar für Folgeprojekte, die auf das erste Projekt aufbauen. Für die Dokumentation sollten aber so wenig wie möglich Ressourcen verbraucht werden um das erste Ziel, die Software zu liefern, nicht zu beeinträchtigen. Hierzu empfiehlt Cockburn die Dokumentation so weit hinaus wie möglich zu verschieben und knapp zu halten. Im Idealfall entsteht sie aus den Notizen der Meetings (z.B. durch Whiteboards mit Druckfunktion) mit wenig Formalismus. Dabei spielt es keine Rolle, ob die Dokumentation bis ins letzte Detail vollständig und in Übereinstimmung mit dem Quelltext ist. Es geht nur darum, dass das nächste Team genug Informationen erhält um an dem Projekt weiter zu arbeiten. [Cockburn 2003, S. 235 ff.] Crystal Clear lässt sich auch als eine aufgelockerte Variante von XP beschreiben (bis auf den oben genannten Dokumentationsaspekt). Z.B. sind beim XP kontinuierliche Unit-Tests mit anschließender Integration erforderlich, während bei Crystal Clear die Integration je nach Projekt nicht kontinuierlich stattfinden muss. Die „Real Customer Involvement“-Praktik beim XP verlangt einen Kundenrepräsentanten, der dem Entwicklerteam vor Ort durchgehend zur Verfügung steht, bei Crystal Clear genügt eine normale Kontaktaufnahme mit einem Umfang von eine Stunde pro Woche [Cockburn 2005, S. 266 ff.]. Das Pair Programming ist als solches auch nicht obligatorisch, Crystal Clear schlägt dafür das Side-By-Side-Programming vor. Beim Side-By-Side-Programming haben die Entwickler immer noch ihre eigenen Aufgaben und ihr eigenes Bildschirm, sitzen aber so nah beieinander, dass sie jeweils auf den Bildschirm des anderen schauen können [Cockburn 2005, S. 126].

Crystal Orange

Crystal Orange ist für Projekte mit bis zu 40 Mitarbeitern, die sich alles in einem Gebäude befinden, geeignet. Wichtige Merkmale geeigneter Projekte sind eine Dauer von 1-2 Jahren, die Notwendigkeit der Einhaltung von Produkteinführungszeiten und die Effizienz bezüglich der Ressourcen Zeit und Geld. Bei Crystal Orange stellt sich die Frage, wie es noch möglich ist bei dieser Projektgröße agil zu bleiben. Dafür werden die Mitarbeiter erst in die Teams Systemplanung, Projektüberwachung, Architektur, Technologie, Funktionen, Infrastruktur und Externe Tests eingeteilt. Das funktionale Team, welche unter anderem für Implementierung zuständig ist und mit größter Wahrscheinlichkeit das größte Team bildet, wird in mehrere Gruppen zerlegt. Alle Teams und Gruppen arbeiten dann nach den Crystal Clear Grundsätzen. [Cockburn 2003, S. 272]

Artikel

- Bruce Tate

Literatur

- Beck, K.: Extreme Programming Explained. Addison-Wesley, Boston, 3. Auflage, 2005
- Beedle, M.; Devos, M.; Sharon, Y.; Schwaber, K.; Sutherland, J.: SCRUM: An extension pattern language for hyperproductive software development, Paper for Conference on Pattern Languages of Programs (PLoP), 1998 pdf (http://www.jeffsutherland.com/scrum/scrum_plop.pdf)
- Cockburn, A.: Agile Software-Entwicklung. mitp-Verlag, Bonn, 2003
- Cockburn, A.: Crystal Clear. mitp-Verlag, Bonn, 2005
- Pomberger, G; Pree, W.: Software Engineering. Carl Hanser Verlag, München, 3. Auflage, 2004
- Pressman, R.S.: Software Engineering. McGraw-Hill, Boston, 6. Auflage, 2005
- Schwaber, K.: SCRUM Development Process, Paper for Object-Oriented Programming, Systems, Languages & Applications (OOPSLA), 1996 pdf (http://jeffsutherland.com/oopsla/schwapub.pdf)
- Sommerville, I.: Software Engineering. Pearson/Addison-Wesley, Boston [u.a.], 7. Auflage, 2004