**Interim Report**

**EE5500**

Name: Wojciech Lesnianski

Student number: 1644612

Electronic and Computer Engineering

School of Engineering and Design



Dr. Ali Mousavi

Montag, Januar 01, 2018

School of Engineering & Design

Electronic & Computer Engineering

MSc Distributed Computing Systems Engineering

Brunel University West London

Simulation and Performance Analysis

of a Distributed Position Correction

Scheme for Unmanned Aerial Vehicles

Dionysios Satikidis

Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Declaration:** I have read and I understand the MSc dissertation

guidelines on plagiarism and cheating, and I certify that this

submission fully complies with these guidelines.

Abstract

Acknowledgements

Me for taking teh time to write all this stuff

List of Abbreviations

|  |  |
| --- | --- |
| VIN | **V**ehicle **I**dentification **N**umber |
| SF | **S**auberes **F**ahren |
| C# | C-Sharp |
| UMTS | **U**niversal **M**obile **T**elecommunications **S**ystem |
| XML | **E**xtensible **M**arkup **L**anguage |
| Wifi | **Wi**reless **Fi**delity |
| OBD | **O**n-**B**oard-**D**iagnose |
| OSM | **O**pen **S**treet **M**ap |
| GUI | Grafische Benutzeroberfläche (**G**raphical **U**ser **I**nterface) |
| WPF | **W**indows **P**resentation **F**oundation |
| PNG | **P**ortable **N**etwork **G**raphics |

List of Therms

|  |  |
| --- | --- |
| VIN / Vehicle Identification Number | **Vehicle Identification Number**,  ist die international genormte, 17-stellige Nummer, mit der ein Kraftfahrzeugeindeutig identifizierbar ist. |
| Fahrzeugtyp | Fahrzeugtyp wird in diesem Projekt gleichgesetzt mit von der Informationen, welche aus der VIN gewonnen werden können. Dazu gehören der Hersteller, das Modell und die Motorisierung. |
| OSM / Open Street Map | Open Street Map ist ein frei nutzbares Projekt, welches Geodaten bereitstellt. |

Content

[1 Introduction 1](#_Toc502580207)

[1.1 Die 1](#_Toc502580208)

1. Introduction

blabla

* 1. Die

Literaturverzeichnis

|  |  |
| --- | --- |
| **Bücher:** | |
| [BUN09] | Hans Joachim Bungartz, et al. „Modellbildung und Simulation“, Springer, 2009 |
| [GOL12] | Joachim Goll „Methoden des Software Engineering“, Springer Vieweg, 2012; ISBN: 978-3-8348-2433-2 |
| [MEY14] | Bertrand Meyer, „Agile! – The Good, the Hype and the Ugly“, Springer, 2014; ISBN: 978-3-319-05154-3 |
| [LOR00] | Edward Lorenz „The Chaos Avant-grande: Memories oft he Early Days of Chaos Theory“, Kapitel 7 „The Butterfly Effect“, 2000; ISBN: 978-981-02-4404-0 |
| **Internetquellen:** | |
| [TUD] | <http://ls4-www.cs.tu-dortmund.de> (Letzter Zugriff: 04.01.16) |
| [AGI1] | <http://agilemanifesto.org/iso/de/principles.html> (Letzter Zugriff: 04.01.16) |
| [AGI2] | <http://agilemanifesto.org/iso/de> (Letzter Zugriff: 04.01.16) |
| [TUD1] | <http://ls4-www.cs.tu-dortmund.de/download/LehreMaterialien/MAO2011/Kap7.pdf> (Letzter Zugriff: 04.01.16) |
| [SWI] | http://sumo.dlr.de/wiki (Letzter Zugriff: 04.01.16) |
| [SWI1] | <http://sumo.dlr.de/wiki/SUMO_User_Documentation> (Letzter Zugriff: 04.01.16) |
| [SWI2] | <http://sumo.dlr.de/wiki/Theory/Traffic_Simulations> (Letzter Zugriff: 04.01.16) |
| [SWI3] | <http://sumo.dlr.de/wiki/Networks/Building_Networks_from_own_XML-descriptions> (Letzter Zugriff: 04.01.16) |
| [SWI4] | <http://sumo.dlr.de/wiki/Definition_of_Vehicles,_Vehicle_Types,_and_Routes> (Letzter Zugriff 04.01.16) |
| [SWI5] | <http://www.sumo.dlr.de/wiki/Simulation/Output> (Letzter Zugriff: 04.01.16) |
| [CUN04] | <http://www.artima.com/intv/simplestP.html> – Ein Gespräch zwischen Bill Venners und Ward Cunningham; „A Conversation with Ward Cunningham, Part V“ (Letzter Zugriff: 04.01.16) |
| [ZAV] | www2.research.att.com/~pamela/faq.html – Quelle angegeben in [MEY14]. (Letzter Zugriff: nicht erreichbar) |
| **Weitere Arbeiten und Projekte:** | |
| [C2X] | Matthias Schulze, Timo Kosch, Ilse Kulp, Thomas Benz, Andrea Tomatis, Ilja Radusch, Gerhard Noecker, Luisa Andreone, Tanja Kessel, Carola Klessen; „Pre-Drive C2X“ – Deliverable D0.3 Final Report; 30.06. 2010 |
| [EMMA] | Sven Lahde, Michael Doering, Wolf-Bastian Pöttner, Moritz Rosin, Gerrit Lammert und Lars Wolf; „Environmental Monitoring in/of Metropolitan Areas“ |
| [GAW] | Christian Gawron, „Simulation-Based Traffic Assignment – Computing User Equilibria in Large Street Networks“, 1998; <http://www.sumo.dlr.de/pdf/GawronDiss.pdf> (Letzter Zugriff: 04.01.16) |

Anhang A

**Prinzipien [MEY14 – Kapitel 4]**

*(Übersetzung vom Englischen ins Deutsche)*

In der agilen Entwicklung bildet eine Anzahl grundlegender Prinzipien, als methodologische Regeln, die Basis für die spezifischen Praktiken und Artefakte. Diese Prinzipien ermöglichen einen Gesamtüberblick wie Software entwickelt werden sollte. Wir werden uns nun mit den Prinzipien, welche den Kern des agilen Ansatzes repräsentieren, im Detail auseinandersetzen.

**4.1 Was ist ein Prinzip?**

Um den methodologischen Kontext zu erklären, ist es erst einmal vom Vorteil, sich daran zu erinnern, was als Prinzip klassifiziert wird und was nicht. Ein gutes methodologisches Prinzip ist sowohl abstrakt als auch widerlegbar. Die Abstraktion unterscheidet Prinzipien von Praktiken. Widerlegbarkeit wiederum unterscheidet sie von Plattitüden.

**Abstraktion** bedeutet, dass das Prinzip eine Grundregel ist, nicht eine spezifische Praxis. „Baue dir eine solide finanzielle Basis für die Zukunft auf“ ist ein Prinzip; „Lege jeden Monat 10% deiner Einnahmen auf ein Sparkonto“ ist eine Praxis. Oft, wie auch in diesem Beispiel und bei den agilen Praktiken, die in einem späteren Kapitel besprochen werden, existiert eine Praktik, um ein Prinzip zu befriedigen.

**Wiederlegbarkeit** bedeutet, dass es einer vernünftigen Person möglich sein muss, mit dem Prinzip nicht einverstanden zu sein. Wenn keiner, der bei vollem Verstand ist, mit der vorgeschlagenen Regel „erreiche hohe Softwarequalität“ nicht einverstanden wäre (wer würde *nicht* versuchen, hohe Softwarequalität zu erreichen?), so könnte sie zwar richtig sein, aber auch uninteressant. Damit eine Regel ein Prinzip wird, muss man in der Lage sein – abgesehen von den eigenen Ansichten – sich jemanden vorzustellen, der die Gegenmeinung vertritt. „Teste zuerst“ erfüllt dieses Kriterium: es ist einem möglich, dafür zu argumentieren, dass Programme vor den Tests geschrieben werden sollten, oder dass lediglich die Spezifikation dem Programm vorausgehen sollte. Eine Regel, deren Negation nicht haltbar ist, wie bei „erreiche hohe Softwarequalität“, ist kein Prinzip sondern eine Plattitüde.

Die Prinzipien, welche in diesem Kapitel behandelt werden, erfüllen diese Anforderungen. Praktiken sind wichtig und haben ein eigenes Kapitel; Auch wenn Plattitüden in der agilen Literatur gelegentlich vorkommen, werden wir sie im Folgenden ignorieren.

Zusätzlich sollte ein Prinzip generell normativ und nicht beschreibend sein: statt einen Fakt oder eine Eigenschaft anzugeben, beschreibt es eine Handlung („Du sollst die Frau deines Nächsten nicht begehren“). Diese Anforderung ist für nicht-technische Bereiche optional („the best is the enemy oft he good“ [SPRICHWORT] ist ein Prinzip, obwohl es nicht durch eine Handlung beschrieben wird), aber bei Prinzipien, die die Softwareenticklungs-Methodologie beschreiben, ist es von Vorteil, eine Handlung als Beschreibung zu benutzen, so wie es in diesem Kapitel gezeigt wird.

**4.2 Die offiziellen Prinzipien**

Da die bereits in der Einleitung erwähnten zwölf Prinzipien des agilen Manifests die offizielle Sicht/Auffassung beschreiben/repräsentieren, werden diese hier zu Beginn aufgezeigt:

**Offizielle agile Prinzipien**

|  |  |
| --- | --- |
| A1 | Unsere höchste Priorität ist es, den Kunden durch frühe und kontinuierliche Auslieferung wertvoller Software zufrieden zu stellen. |
| A2 | Heisse Anforderungsänderungen selbst spät in der Entwicklung willkommen. Agile Prozesse nutzen Veränderungen zum Wettbewerbsvorteil des Kunden. |
| A3 | Liefere funktionierende Software regelmäßig innerhalb weniger Wochen oder Monate und bevorzuge dabei die kürzere Zeitspanne. |
| A4 | Fachexperten und Entwickler müssen während des Projektes täglich zusammenarbeiten. |
| A5 | Errichte Projekte rund um motivierte Individuen. Gib ihnen das Umfeld und die Unterstützung, die sie benötigen, und vertraue darauf, dass sie die Aufgabe erledigen. |
| A6 | Die effizienteste und effektivste Methode, Informationen an und innerhalb eines Entwicklungsteams zu übermitteln, ist im Gespräch von Angesicht zu Angesicht. |
| A7 | Funktionierende Software ist das wichtigste Fortschrittsmaß. |
| A8 | Agile Prozesse fördern nachhaltige Entwicklung. Die Auftraggeber, Entwickler und Benutzer sollten ein gleichmäßiges Tempo auf unbegrenzte Zeit halten können. |
| A9 | Ständiges Augenmerk auf technische Exzellenz und gutes Design fördert Agilität. |
| A10 | Einfachheit – die Kunst, die Menge nicht getaner Arbeit zu maximieren – ist essenziell. |
| A11 | Die besten Architekturen, Anforderungen und Entwürfe entstehen durch selbstorganisierte Teams. |
| A12 | In regelmäßigen Abständen reflektiert das Team, wie es effektiver werden kann und passt sein Verhalten entsprechend an. |

Diese Liste ist gut, um sich in Stimmung zu bringen, aber – obwohl sie direkt von der Quelle kommt – können wir nicht mit ihr arbeiten und unsere erste Aufgabe wird daher sein, eine passendere und brauchbarere Liste zu erstellen. Die offizielle Liste kommt dafür nicht in Frage, weil:

* Einige der Punkte sind Praktiken: A6, A12
* Andere sind Plattitüden: A5 – Wer würde ein Projekt um unmotivierte Teilnehmer errichten wollen? – und A9

Manche sind keine Handlungen, sondern Behauptungen, was kein Problem ist, weil man sie zu Handlungen umschreiben kann (A7 könnte in „Benutze funktionierende Software als Maß für den Fortschritt“ umgeschrieben werden), werden aber problematisch, wenn die Behauptung **falsch** ist. Es ist nicht wahr, dass wie es in A10 beschrieben wird, Einfachheit die Kunst ist, nicht getane Arbeit zu maximieren: Nach der Einfachheit zu suchen ist ein bedeutungsvolles Prinzip; genauso wie nicht getane Arbeit zu maximieren; aber es sind verschiedene Prinzipien.

(Es ist ein wichtiges Anliegen, welches wir uns später im Detail anschauen werden.) Die Benutzung einer Handlungs-Form hätte die Verwirrung eventuell verhindern können.

* Obwohl wir einen Satz von unabhängigen Regeln erwarten würden, sind die hier aufgeführten teilweise redundant: regelmäßige Auslieferung wird in A1 und A3 erwähnt und die Wichtigkeit von funktionierender Software in A3 und A7.
* Auf der anderen Hand sind die Regeln eindeutig unvollständig: in keiner von ihnen wird das Testen erwähnt, obwohl der Fokus auf das Testen zur Qualitätssicherung eine der Haupteigenschaften des agilen Ansatzes ist.

**4.3 Eine brauchbare Liste**

Um die offizielle Liste zu ersetzen, werden wir die Klassifizierung der agilen Prinzipien benutzen, die im Einführungskapitel vorgestellt wurde. Für einfaches Verständnis ist hier nochmal die erwähnte Liste:

**Agile Prinzipien**

|  |  |
| --- | --- |
| **Organisational** | |
| 1 | Stelle den Kunden in den Mittelpunkt. |
| 2 | Lass das Team sich selbst organisieren. |
| 3 | Arbeite im nachhaltigen Tempo. |
| 4 | Entwickle minimale Software: |
|  | |  |  | | --- | --- | | 4.1 | Produziere minimale Funktionalität. | | 4.2 | Produziere nur das, was verlangt wurde. | | 4.3 | Entwickle nur den Code und die Tests. | |
| 5 | Akzeptiere Änderungen |
| **Technisch** | |
| 6 | Entwickle Iterativ. |
|  | |  |  | | --- | --- | | 6.1 | Produziere regelmäßige funktionierende Iterationen. | | 6.2 | Friere die Anforderungen während den Iterationen ein. | |
| 7 | Behandle Tests wie eine Schlüsselressource. |
|  | |  |  | | --- | --- | | 7.1 | Starte keine neuen Entwicklungen, bis alle Tests bestanden sind. | | 7.2 | Teste zuerst. | |
| 8 | Drücke Anforderungen durch Szenarios aus. |

Wir werden uns zuerst die organisationalen Prinzipien anschauen, anschließend die Software-spezifischen technischen Prinzipien.

**4.4 Organisatorische Prinzipien**

Organisatorische Prinzipien beeinflussen das Projektmanagement, die Ablaufplanung und die Teamorganisation

**4.4.1 Stelle den Kunden in den Mittelpunkt**

„Wir sind Kunden-orientiert“ ist im Geschäftsbetrieb eine Plattitüde. Agiles Entwickeln nimmt diese Idee ernst, indem es eine tiefe Einbindung des Kunden über das gesamte Projekt fordert.

In vielen traditionellen Ansätzen wirken die Kunden nur an bestimmten Punkten mit: sie liefern den Input während den Anforderungen – nur als Teil eines strikt kontrollierten Prozesses, bei Anforderungsinterviews oder Workshops – und zeigen sich nicht, bis zu den finalen Etappen im Softwaretest. Einige Unternehmen verbieten sogar den Kontakt zwischen Kunde und Entwickler zwischen solchen Etappen, die meisten machen jedoch nicht derart extreme Einschränkungen.

In agilen Ansätzen findet die Interaktion mit dem Kunden durchgehend über das ganze Projekt hinweg statt. In Becks Worten:

*Du wirst [bessere] Resultate mit wirklichen Kunden bekommen. Sie sind es, die du versuchst zufriedenzustellen. Kein Kunde, oder „Proxy“ für einen echten Kunden führt zu Verschwendungen, weil du Features entwickelst, die nie benutzt werden, Tests spezifizierst, die nicht die richtigen Annahmekriterien widerspiegeln, und die Chance verlierst, eine echte Beziehung zwischen den Menschen aufzubauen, die die unterschiedlichsten Ansichten zum Projekt haben.*

Kunden in agilen Projekten werden regelmäßig zu Projektsitzungen eingeladen, können frei mit den Entwicklern interagieren und haben die Möglichkeit, das Produkt auszuprobieren, während es in inkrementellen Schritten veröffentlicht wird. Einige Ansätze gehen weiter und empfehlen das Einbetten des Kunden ins Entwicklerteam.

Die Betonung auf das Einbinden des Kunden adressiert eine der prinzipiellen Gefahren, welche Softwareprojekte gefährden: eine Software zu entwickeln, welche sich nicht an die Anforderungen des Benutzers richtet. Schon 1981 zitierte Boehms Klassiker *Software Engineering Economics* Softwaremisserfolge, wo die im Projekt produzierten Programme alles besaßen – gute Performance, Verlässlichkeit, … - außer einem Detail: sie lösten ein Problem, welches anders war, als das was der Kunde brauchte oder wollte. Lutzs empirische Analyse der Quellen von softwarebezogenen Sicherheitsfehlern in großen NASA-Missionen, auch ein Klassiker, berichtet:

*Die Hauptursache für sicherheitsbezogene Funktionsstörungen sind Fehler im* ***Erkennen (Verstehen) der Anforderungen*** *(62% beim Voyager, 79% beim Galileo). Sicherheitsbezogene Konditionsstörungen zum Beispiel sind so gut wie immer darauf zurückzuführen, dass die Anforderungen nicht richtig erkannt wurden.*

Solche Studien können jedoch auch benutzt werden um das Schreiben einer genaueren anfänglichen Anforderungsliste zu rechtfertigen. Genau das, was die Agilisten so ablehnen.

Die Ermutigung, Kunden einzubinden, ist ein wichtiger agiler Beitrag. Das Problem ist die Beharrlichkeit, dass solche Interaktionen die Anforderungen *ersetzen*. Solch ein Schritt ist gefährlich, denn es gibt nicht so was wie „den Kunden“. Jedes signifikante Projekt umfasst viele Kategorien von Stakeholdern (ein Überbegriff für Kunden). Dazu zählen unter anderen:

* die Benutzer des künftigen Systems selbst, sowie beispielsweise für ein online Eventreservierungssystem, die Eventverwaltung, die Besitzer der Theater und anderer Veranstaltungsorte, Veranstaltungsbesucher, Artisten, Agenten, Produzenten.
* Führungskräfte – die auch für die Kunden arbeiten, aber sich im Besonderen auf Sachen wie Integration mit der Geschäftspolitik und Weiterentwicklung konzentrieren.
* Einkauf von Agenten, Anwälten und so weiter.

Diese verschiedenen Parteien sind oft im Konflikt und brauchen Prioritäten, und genau das ist die Aufgabe eines guten Anforderungsprozesses, Widersprüche ans Licht zu bringen, sie wenn möglich aufzulösen (aus diesem Grund sind Anforderungsworkshops oft gut, um die individuellen Anforderungssitzungen zu ergänzen), und um Entscheidungen von der verantwortlichen Person zu erhalten – dem „product owner“.

Wenn man diesen formalen Prozess, mit den Stakeholdern zu sprechen, durch die Praxis ersetzt, besteht die Gefahr, dass das Programm auf die Visionen derer passen wird, die sich in den Entwicklungsprozess einbringen. Sie können jedoch nicht die beste Quelle sein: die Menschen deren Ansichten am wichtigsten sind, können auch diejenigen sein die am meisten beschäftigt sind; sie könnten Zeit finden für einen Anforderungsworkshop, würden ihre Tür jedoch nicht dauerhaft offen halten für jede Frage eines jedes Entwicklers. Die Wahrscheinlichkeit ist also groß, dass man von Leuten beeinflusst wird, die zu viel Zeit haben – möglicherweise deshalb, weil ihre Arbeit für das Unternehmen von keiner allzu großen Beteutung ist. Diese Gefahr ist besonders hoch in Projekten mit Methoden, die vorschreiben, einen Kunden in das Projekt einzubetten: wenn das Management willig ist, einen Experten aus seinem Aufgabenfeld rauszuziehen und ihn in dein Projekt integrieren, musst du dich fragen, ob diese Person wirklich die qualifizierteste ist.

Beck betont das Risiko, welches sich daraus ergibt, dass man auf nur eine Person hört:

*Der Einspruch, den ich gegen die Einbeziehung des Kunden höre ist, dass jemand genau das System bekommen wird, das er will, aber das besagte System wird für keinen anderen geeignet sein.*

Er beantwortet den Einspruch mit der Aussage:

*Es ist einfacher, ein gelungenes System zu generalisieren, als ein System zu spezialisieren, das die Probleme löst, die keiner hat.*

Dieses Argument ist fraglich. Das „gelungene System“ kann so auf den erwähnten Kunden zugeschnitten sein, dass es eine komplette Überarbeitung benötigen würde, um für jemand anderen nützlich zu sein, während ein „System, das die Probleme löst, die keiner hat“, vielleicht eine solide Basis hat, aber eine furchtbare Benutzeroberfläche, die repariert werden kann. Sogar wenn wir Becks Ansicht akzeptieren, erklärt dies nicht, wieso das Kommunizieren mit *einigen* der Kunden während des Projekts, das Konsultieren von *allen* Stakeholdern ausschließt – eine Ablehnung, die ziemlich unverantwortlich erscheint.

**4.2.2 Lass das Team sich selbst organisieren**

Der agile Ansatz nimmt den Managern traditionelle Rollen weg, wie das Zuweisen von Aufgaben an die Entwickler. Er setzt Vertrauen in die Fähigkeit des Teams, die eigene Arbeit selbst zu organisieren. Scrum ist besonders systematisch in diesem Aspekt, indem es die traditionelle Vorstellung eines Projektmanagers durch ein ermächtigtes Team ablöst, welches seine Entscheidungen eigenhändig unter der Kontrolle eines Product Owners, welcher über die Produktfunktionalität bestimmt, und eines Scrum-Masters, welcher das Entwicklerteam unterstützt und Methoden bestimmt, durchführt. Wir werden uns diese Rollen im nächsten Kapitel anschauen.

Für viele Entwickler, die unter einem schlechten Projektmanager leiden mussten, ist dies einer der großen Reize der agilen Methoden. Ein Verteidiger von XP und Scrum hat als Antwort auf einen ausführlichen Blogartikel, der agiles Entwickeln kritisiert hat, folgendes geschrieben:

*Der wichtigste Aspekt von derartigen Methoden ist es, das Management des Projekts dorthin zu stellen, wo es hingehört: hinter die Leute, die die Arbeit machen. Wenn die Leute, die tatsächlich die Arbeit machen, das Sagen haben, was gemacht wird und wann, dann werden Projekte auch wirklich innerhalb der Frist fertiggestellt.*

Die Notwendigkeit von Managern bleibt natürlich, da Unternehmen nun mal so funktionieren und weil, wie die Erfinder der originalen (nicht-Software) Scrum Methode meinten, dass *„*eine subtile Kontrolle im Einklang mit dem selbst-organisatiorischen Charakter eines Projektteams steht*“*. Schwaber und Sutherland, die Erfinder des Software-Scrums, zeigen ebenfalls Empathie für das Konzept der subtilen Kontrolle:

*Kontrolle durch gleichmäßigen Druck und „Kontrolle mit Liebe“ sind die Basis für subtile Kontrolle.* Der dynamische Fluss des Teams fördert das implizite Wissen der Gruppe zu Tage und schafft daraus explizites Wissen in Form von Software.

Wörter die einem Angst machen können, wenn dir gesagt wurde, dass dein Team selbst-organisiert ist und plötzlich erfährst du, dass du in Wirklichkeit subtil, durch stillschweigende, unterbewusste Techniken, kontrolliert wirst. Der Teil über die Liebe kann beruhigend sein; oder auch nicht.

Vielleicht ist es Ansichtssache. Persönlich, wenn ich geführt werden sollte, hätte ich lieber einen Boss, als wenn mir jemand sagen würde, ich wäre selbst-organisiert, um mich dann mit diversen Kontrolltechniken zu manipulieren.

Tatsächlich ist die Rolle des Managers in der agilen Literatur undeutlich; Kommentare diesbezüglich tendieren dazu, von empörter Natur zu sein, wie „es ist ein häufiges Missverständnis, dass in agilen Projekten …“, und mögen wahr sein, doch sie sagen uns nicht, wieso das Missverständnis überhaupt erst entstand und was noch viel wichtiger ist, was die ordnungsgemäße (nicht missverstandene) Rolle des Managers ist. Schwaber und Sutherland zum Beispiel, schreiben:

**Obwohl** Teams von Entwicklern auf sich alleine gestellt sind, sind sie **nicht** unkontrolliert.

Und Cohn:

Ein häufiges Missverständnis bezüglich der Leitung agiler Projekte ist, dass durch die selbst-organisierten Teams die Rolle eines Teamleiters wenig oder gar nicht benötigt wird. Nichts könnte der Wirklichkeit ferner sein. In „The Biology of Business“ wiederlegt Philip Anderson diese falsche Annahme: „Selbst-Organisation bedeutet nicht, dass die Arbeiter statt der Leitung für die Organisation verantwortlich sind. Es hat nicht die Bedeutung, man solle die Leute machen lassen, was sie wollen. Es bedeutet, dass das Management für die richtige Entwicklung der Verhaltensweisen, welche sich aus dem Zusammenspiel verschiedener Parteien ergeben, verantwortlich ist und nicht gleich zu Beginn entscheidet, welche die richtige Verhaltensweise ist.“

Selbst-organisierte Teams sind von einer Leitung nicht ausgeschlossen. Das Management entscheidet welches Produkt entwickelt wird und oft, wer am Projekt arbeiten wird, aber die Teams sind dennoch selbst-organisiert. Dennoch sind sie nicht frei von Einflüssen. … Dennoch, je weniger Abhängigkeiten oder Kontrollen auf ein Team hat, umso besser.

Mit anderen Worten, agile Manager „kontrollieren“, nur dass sie es nicht tun, oder vielleicht doch, aber jedenfalls nicht so viel.

Agile Texte sind reich an Projektanekdoten, welche die gewünschte Balance zwischen zu viel und zu wenig zeigen; aber die Manager, die nach unternehmensweit geltenden allgemeinen Prinzipien suchen, werden nur Regeln finden, die besagen, was Manager *nicht* tun sollten, zum Beispiel entscheiden, welche Funktionen eingebunden werden sollten (das entscheidet der Product Owner) und wer, wann und woran arbeiten sollte (das entscheidet das Team).

Cohn’s Behauptung, dass „selbst-organisieren“ nicht bedeutet „die Leute zu machen lassen was sie wollen“ bringt einen ins Grübeln. Wenn es da einen Unterschied gibt, muss er subtil sein. Derby betont in einem Artikel, welcher erneut den „Missverständnissen“ gewidmet war, dass „selbst-organisierend“ nicht „selbst-organisiert“ bedeutet. Das liegt daran, dass [selbst-Organisation] ein Vorgang und eine Charakteristik ist, nicht etwas, das ein für allemal gemacht wird. Selbst-organisierend, aus der Sicht eines sozialen Systems heißt **nur**, dass das Team neue Ansätze erschaffen kann und sich an neue Herausforderungen in seiner Umgebung anpassen kann.

„Neue Ansätze erschaffen, um sich an neue Herausforderungen anzupassen“ klingt enttäuschend. Welches gut geführte Projekt – ein traditionelles, von einem starken „befehle und kontrolliere“ Leiter eingeschlossen – würde seine Mitglieder das nicht machen lassen, oder sie sogar noch dazu ermutigen? „Selbst-organisierend“ muss ambitionierter sein. Mittal schreibt, dass selbst-organisierende Teams

„*immer noch Betreuen und Coachen erfordern, aber nicht mehr Befehlen und Kontrollieren.“*

Während Betreuen und Coachen wirklich wichtige Rollen im agilen Management spielen (wie wir im nächsten Kapitel sehen werden), ist die negative Seite der Beobachtungen wieder enttäuschend: „Befehlen und Kontrollieren“ ist das, was die Manager traditionell tun; wie von Cohn darauf hingewiesen, müssen sie dies immer noch, aber es wäre nützlich zu erfahren, was genau.

Obwohl man die Antwort in der agilen Literatur vergeblich sucht, ist sie eigentlich gar nicht so schwer, um sie nicht selbst mit ein wenig gesundem Menschenverstand herleiten zu können. Die meisten Projekte brauchen einen Manager, der sich um das „Befehlen und Kontrollieren“ kümmert. Der Nachteil eines Militär-Stil-Schemas, in dem eine Person diese Rolle übernimmt, ist dass es die Kreativität der Teammitglieder eingrenzt. Das andere Extrem ist ein talentiertes und erfahrenes Team, welches komplett selbst-organisiert, mit oder ohne einen „Mentor“ oder „Coach“ zurechtkommt.

In der Musikwelt, ist das legendäre I Musici ein bekanntes Beispiel. Dieses Orchester spielt seit 1952 und ist eines der besten Kammerorchester der Welt.Wie der Wikipedia Eintrag festhält:

„I Musici ist ein dirigentenloses Ensemble. Aber die Beziehungen zwischen den Musikern erlauben große Harmonie in ihrer Musik.“ In der Tat! Wenn man eine Gruppe von Super-Softwareentwicklern zusammen tut, können sie sich selbst managen, so wie I Musici, und es jedem „Lackaffen“ übel nehmen, der denkt, er könnte sie rumschubsen. Auf der anderen Seite – einer Gruppe von unerfahrenen Musikern zu sagen, sie sollen zusammen musizieren, wird nicht funktionieren. Nicht mal erfahrene, professionelle Musiker können für gewöhnlich auf diese Weise funktionieren; deshalb haben die meisten Orchester – kleinere Ensembles miteingeschlossen – einen Dirigenten. Die meisten Softwareentwicklungsteams brauchen dementsprechend einen Projektmanager.

Was wir am Ende aus den selbstorganisierenden Teams lernen können ist folgendes:

* Ausnahmen, wie erfahrene und gut zusammenarbeitende Teams („ I Programmatori“) können ohne einen Leiter funktionieren. Die meisten Teams jedoch benötigen einen.
* Einige der traditionellen Manager-Rollen, so wie das Auswählen der Aufgaben für die nächste Entwicklungsiteration, können auf die Teammitglieder verteilt werden.
* Der Manager sollte bei den Teammitgliedern Initiative fördern und nach und nach das Team in Richtung teilweise oder vollständig selbstorganisierend bewegen. (Hier kann Derbys Bemerkung hinzugefügt werden, dass sich das Team bis zu diesem Standpunkt auf das selbst-Management zu entwickelt.)

**4.4.3 Arbeite im nachhaltigen Tempo**

Agile Methoden betonen die zentrale Rolle der Programmierer und das Bedürfnis, ihnen eine Arbeitsumgebung zu schaffen, die ihr volles Potential fordert. Eine der Konsequenzen dieser Ansichten ist die Ablehnung von dem, was Ed Yourdon in einem bekannten und nützlichen Buch *dead marches* schrieb: Die Praxis, eine unrealistische Verpflichtung zu akzeptieren – in einem Projekt mit vagen und ständig wachsenden Anforderungen und knappen Deadlines – dann zu versuchen, das Entwicklerteam dazu zu bringen, diese durch ständigen Druck, lange Werktage und das Opfern vom Wochenende zu erfüllen.

Ein weiteres einflussreiches Buch war DeMarco’s und Lister’s PeopleWare (zum ersten Mal publiziert in 1987), welches in einfachen Worten erklärt, wie Programmierer funktionieren und wie wichtig es ist, ihnen eine ruhige, respektvolle Arbeitsumgebung zu bieten.

Cockburn war besonders betonend beim Thema Förderung der Prinzipien der „persönlichen Sicherheit“ für die Entwickler, welche ihnen erlauben, sich frei zu äußern. In seinen Worten:

„Persönliche Sicherheit bedeutet in der Lage zu sein, etwas zu sagen, wenn es einen stört, ohne Angst vor einer Vergeltungsmaßnahme haben zu müssen. Es kann beinhalten, dem Manager zu sagen, dass der Zeitplan unrealistisch ist, einem Kollegen, dass sein Entwurf Verbesserung bedarf oder gar dass er öfter duschen sollte. Mit persönlicher Sicherheit kann das Team seine Schwächen erkennen und reparieren. Ohne sie redet keiner und die Schwächen schädigen das Team weiter.

Weiter verallgemeinernd: Agilität unterstreicht – in der Tradition von PeopleWare – den nötigen Respekt den Programmierern gegenüber zu erbringen und das Bedürfnis, unter angemessenen Umständen zu arbeiten, zu erfüllen. Diese Ideen lassen sich gut mit den anderen Aspekten der Methode kombinieren: der Vorzug von persönlicher Kommunikation gegenüber geschriebenen Dokumenten; der Ratschlag (besprochen in einem anderen Kapitel), offene Flächen statt Kabinen zu nutzen. Schwaber beschreibt das Vorher-Nachher einer Firmenatmosphäre:

*Die erste Tour durch den Engineering-Raum bei Service1st war einfach deprimierend. Die Menschen wurden entweder in Büros mit geschlossenen Türen oder in Kabinen verbannt verbannt… Es gab keine Unterhaltungen, keine Geräusche von Aktivitäten, kein Gefühl, dass hier Leute bei der Arbeit sitzen und diese gerne tun.*

Das Unternehmen stellte ihn jedoch ein und beim zweiten Sprintreview hieß es dann:

*Alles fühlte sich anders an… Die Menschen sprachen, lachten und eine lebhafte Konversation füllte den Arbeitsplatz.*

Die gesellschaftspolitischen Obertöne sind interessant. Wir stellten im ersten Kapitel fest, dass die agilen Ideen eine gesellschaftspolitische Interpretation haben: „Der Aufstand der Kabinen“. Die Agile Bewegung spiegelt die Selbstbehauptung der Programmierer wieder, preist den Vorrang vom Code an, auf Kosten von „Dilbert’s boss“ – Artefakten wie Plänen Modellen und Dokumenten.

Die Debatte ist nicht neu: schon in 1977 hat ein Buch von Philip Kraft, vervollständigt durch die Analyse von Marx, die Vorreiter der heute verbreitetsten Techniken (das strukturierte Programmieren miteingeschlossen), als einen Versuch des Managements verurteilt, die Programmierer in stimmenloses Proletariat zu verwandeln. Die Analyse von Marxist ist verschwunden – wenn überhaupt, heben Agilisten ROI und andere ungenierte kapitalistische Ziele hervor – aber der Versuch, den Programmierer in den Vordergrund zu bringen, bleibt bestehen.

Es bestehen Nuancen zwischen den agilen Schulen. Zwar fördern sie alle die Ermächtigung der Programmierer, aber nicht alle aus demselben Grund. Aus den vier in diesem Buch behandelten Methoden entstehen zwei Kategorien:

* XP und Crystal sind echte Bewegungen des Programmierer-Stolzes; Cockburns oben zitierter Text ist typisch für den Schwerpunkt dieser Methoden – die Würde der Programmierer gegenüber dem Management wiederherzustellen.
* Der Kern von Scrum und Lean ist anders. Es sind Methoden, welche ihre Wurzeln in der Tradition der industriellen Ingenieurproduktion haben; ihre Autoren zitieren Deming und Toyota, vermeiden Abfall und preisen Produktivität.

Als Beispiel für die zweite Schule, erzählt Schwaber stolz, wie er als Scrum Master für ein Projekt sein Team dazu gebracht hat, die nächste Deadline zu schaffen: als sich rausstellte, dass kein Fortschritt erzielt werden kann ohne einen Schlüsselprogrammierer, der den Kontakt zur Außenwelt abgetrennt hat – oder, wie Schwaber glaubte – zu Yellowstone in seinen ersten Urlaub seit 2 Jahren (versuch das mal in Europa!) gefahren war, heuerte der fleißige Scrum Master einen Privatdetektiv an, um ihn ausfindig zu machen (versuch das mal in Europa!). Vielleicht ist das wieder die „subtile Kontrolle“, die einen nostalgisch die guten Manager vermissen lässt, mit ihrem Vorrecht, aber auch den eingebauten Grenzen ihrer Macht. Da die Anekdote den Anschein macht, nicht nur mit dem furchtlosen Managementstil des Autors angeben, sondern auch noch eine Lehre übermitteln zu wollen, wundert sich der verwirrte Leser, wie sich diese Lehre mit den Prinzipien wie Crystal’s persönlicher Sicherheit versöhnen lässt.

Die von XP empfohlene Praxis des Leerlaufs ist auch ein Teil der Betonung auf die Nachhaltigkeit, wie in einem anderen DeMarco Buch beschrieben. Beck schreibt:

*Platziere in jeden Plan einige kleine Aufgaben, die verworfen werden können, wenn du in Rückstand gerätst.*

und

*Du kannst den Leerlauf auf viele Arten strukturieren. Jede acht Wochen könnte eine „Geek Woche“ sein. Zwanzig Prozent des Budgets könnte in Aufgaben, die von den Programmierern gewählt wurden, investiert werden.*

Die 20-Prozent-Regel ist auch ein Teil von Googles Praktiken.

**4.4.4 Entwickle minimale Software**

Agile Methoden betonen die Einfachheit. Das Ziel ist es, das Feedback des Benutzers schnell zu erhalten, indem man die Software in kleinen Inkrementen abliefert, auch dann, wenn nur ein kleiner Teil der erwarteten Funktionalität vorliegt.

Der agile Sprint des Minimalismus manifestiert sich in einigen Formen: minimale Funktionalität; produziere nur das geforderte Produkt; produziere nur den Code und die Tests. Betrachten wir sie nach der Reihe, anschließend beurteilen wir ihre Tugend des Minimalismus.

***Produziere minimale Funktionalität***

Eine generelle agile Ansicht ist, dass viele Softwaresysteme unter dem „Aufblasen“ leiden: Elemente der Funktionalität, welche nicht gebraucht werden, oder nur von wenigen Benutzern verwendet werden. Während der Entwicklung nehmen sie Zeit weg von der fundamentalen Funktionalität und verspäten die Veröffentlichungen; sie stören den Fokus des Teams; sie erschaffen eine künftige Bürde für die Wartung und können können möglicherweise den künftigen Verlauf der Programmentwicklung erzwingen (weil sobald ein Feature da ist, es von irgendjemandem benutzt wird und er erwarten wird, dass es auch in späteren Versionen immer noch vorhanden sein wird).

Ein Slogan, welcher *durch das XP bekannt wurde ist „You Ain’t Gonna Need It“ oder YAGNI, welches*

*uns immer daran erinnert, an dem zuarbeiten, was wir haben, nicht an dem, was wir denken, einmal zu brauchen. Sogar dann, wenn wir wissen wir, werden es brauchen.*

Die Poppendiecks schreiben:

*Unsere Softwaresysteme enthalten viel mehr Features als sie jemals gebraucht werden. Zusätzliche Features erhöhen die Komplexität des Codes und steigern die Kosten nichtlinear. Sogar wenn die Hälfte unseres Codes unnötig ist – eine konservative Annahme – sind die Kosten nicht einfach die doppelten; sie sind vermutlich zehnfach höher als sie sein sollten.*

(Ich weiß nicht, ob der Faktor 10 eine wilde Schätzung ist, oder wörtlich genommen werden sollte und kenne keine Studie die einen genauen Wert angibt.) Sie fügen hinzu:

*Unsere beste Möglichkeit die Produktivität in der Softwareentwicklung zu verbessern ist es, aufzuhören Features einzubauen die nicht absolut notwendig sind.*

In Kürze,

*Wenn der Code jetzt nicht nötig ist, ist es eine Verschwendung ihn einzubauen. Wiederstehe der Versuchung.*

***Produziere nur das verlangte Produkt***

Softwareengineering-Weisheiten ermutigen Entwickler dazu, nach zwei Softwarequalitäten zu streben, welche langfristige, anstatt von sofortigen, kurzweiligen, Vorteilen erzielen:

* Erweiterbarkeit: entwerfe die Architektur so, dass sie zukünftige Erweiterungen unterstützt – um genau zu sein – solche die künftige Benutzer benötigen, werden.
* Wiederverwendbarkeit: verallgemeinere Softwareelemente so stark wie möglich über ihre genaue Rolle im Projekt hinaus, sodass sie an anderen Stellen in diesem und in künftigen Projekten benutzt werden können. (Wenn das passiert, werden sie zu Softwarekomponenten.)

Für agile Methoden sind das keine wichtigen Ziele, es müssen nicht einmal überhaupt Ziele sein. Das wichtige ist, Software zu entwickeln, die hier und jetzt funktioniert. Hier sind zwei typische Aussagen, die demonstrieren, wie die Agilisten allem misstrauen, das mehr verlangt als das, was im Moment benötigt wird.

Ward Cunningham schreibt:

*Dir wird immer beigebracht, so viel zu tun wie möglich ist. Setze immer Kontrollen ein. Suche immer nach Ausnahmen. Behandle immer den allgemeinsten Fall. Gib dem Benutzer immer den besten Ratschlag. Gib immer eine sinnvolle Fehlermeldung aus. Immer dies. Immer das. Man hat so viele Dinge im Hinterkopf die man tun soll, dass es gar keinen Raum mehr für das Überlegen gibt. Ich sage: vergesst das alles und frag dich selbst, „was ist die einfachste Sache die funktionieren könnte?“.*

Diese Phrase, „Mach die einfachste Sache, die funktionieren könnte“, wurde „wie YAGNI nur ohne ein Akronym“ zum agilen Mantra. Ron Jeffries beschreibt, wieso entwerfen für die Wiederverwendung es nicht wert ist:

*Wenn die Projekte nicht vom selben Team gemacht werden, ist eine effektive Wiederverwendung sehr schwer: es gibt einen großen Unterschied zwischen einem Teil des Projekts, den ich wiederverwenden kann, und dem Verpacken dieses Teils so, dass die Anderen es können. Ich muss Verpackungsarbeit erledigen, die ich ansonsten nicht hätte, sie dokumentieren, sie kugelsicherer machen, Probleme beseitigen, welche ich ansonsten automatisch umgehe, sie unterstützen, Fragen darüber beantworten, den Leuten erklären, wie sie es zu benutzen haben. Wenn ich diese Dinge mache, ist es teuer. Wenn ich sie nicht tue, ist das Benutzen meiner Komponenten für die anderen schwer und hilft nicht wirklich jemandem.*

*Ich erstelle die Abstraktion, die ich brauche. Wenn jemand wieder eine Abstraktion braucht, in einem anderen Kontext, würde ich sie verbessern. Aber wenn der Sinn meines Projekts nicht der ist, Sachen für Projekte anderer Leute zu bauen, versuche ich, keine Zeit und kein Geld dafür zu verschwenden.*

Derartige Aussagen sind Gelegenheiten, sich am Kopf zu kratzen. (Schon wieder. Dieses Buch beinhaltet so viele davon, dass es einen Discount-Coupon für Haarwiederherstellungskuren enthalten sollte.)

Wie in vielen anderen Fällen fangen sie mit korrekten, sogar aufschlussreichen Beobachtungen an: Entwerfen für die Zukunft kann in der Tat von dem Problem hier und jetzt ablenken; Entwerfen für Wiederverwendung ist schwer. Ein überzeugendes Beispiel ist für mich eine Klasse, die Punkte in einem 2-dimensionalen Raum definiert: wie kann man das verallgemeinern? Man könnte über Punkte in einem n-Dimensionalen Raum, für eine beliebige Dimension n, nachdenken; oder über jedes Objekt, welche durch zwei numerische Koordinaten (Punkte, Vektoren, komplexe Zahlen…) definiert wird; oder über eine beliebige zwei-dimensionale Figur (Punkte, Linien, Polygone…). Es gibt keine Möglichkeit, zu wissen, welches davon, wenn überhaupt, die richtige Verallgemeinerung sein wird. In solchen Fällen ist es besser, nicht zu raten wohin uns die Zukunft führen wird.

Aber aus dieser Beobachtung eines gesunden Menschenverstands abzuleiten, dass wir Generalisierungen, Kontrollen, Ausnahmen und Wiederverwendbarkeit vergessen können? Solche Anordnungen sind eine Ermutigung, schlechten Programmierstil einzusetzen. Ein einfaches Beispiel ist die Benutzung von eingebauten Konstanten. Du schreibst Software für ein kleines Unternehmen und brauchst eine Liste der Angestellten; nun, ein Array mit 1000 Elementen sollte reichen, richtig? Bevor du dich versiehst, wächst das Unternehmen und die Software hört auf eine mysteriöse Art und Weise auf zu arbeiten. Historische Katastrophen, welche Milliarden von Dollar in verschwendetem Aufwand verursachten, sind auf diese agilen, machen-wir-nur-das-was-wir-brauchen Ansätze: MS-DOS 640-K Speichergrenze, das Y2K-Schlamassel, die anfängliche Größe der IP Adressen, zurückzuführen.

Der kurzsichtige, oben zitierte Ratschlag, der dir sagt, du solltest dir nur Sorgen darüber machen, was du hier und jetzt brauchst, ist schädlich für deinen Softwareprozess. Egal was du bei den agilen Methoden als nützlich empfindest, du solltest es einfach ignorieren.

***Entwickle nur Code und Tests***

Eins der radikalsten Prinzipien der agilen Methoden missbilligt all die standardmäßig unterstützenden Produkte in der Softwareentwicklung, in genauen Dokumenten – Anforderungen, Entwürfe, Pläne, Programmdokumentation… - als Ablenkung von den Hauptvorgängen, Ausführung von Code und Tests. In Poppendiecks Worten:

*Die Dokumente, Diagramme und Modelle die als Teil eines Softwareprojekts erstellt wurden, sind oft Verbrauchsgüter, Hilfestellungen um das System zu entwickeln, aber nicht unbedingt ein Teil des endgültigen Produkts. Sobald ein funktionierendes System ausgeliefert wurde, kann es seinm dass sich der Benutzer wenig für solche Zwischenverbrauchsgüter interessiert. Lean Prinzipien suggerieren, dass jedes Verbrauchsgut ein Kandidat für die Überprüfung ist. Nicht nur die Bürde zu beweisen, dass er zum Gesamtwert des fertigen Produkts beiträgt, liegt beim Artefakt, sondern auch, dass es der effizienteste Weg ist, diesen Wert zu erreichen.*

„Verbrauchsgüter“ umschließt alles, was nicht an den Kunden ausgeliefert wird. Anders als der Code und die Tests, können die meisten Artefakte in der Softwareentwicklung als Verbrauchsgüter betrachtet werden: Durchführbarkeitsstudien, Protokolle oder Videos von Anforderungsinterviews und Workshops, Anforderungsdokumente, PowerPoint-Präsentationen über die Systemzukunft, Emails, Entwurfsdokumente, UML Diagramme…

Ähnlich dazu beschreibt Beck die Rolle der Architekten wie folgt:

Die Architekten in einem XP-Team suchen und führen Refactoring in einem großen Ausmaß durch, schreiben Tests, welche die Architektur auf der Systemebene beanspruchen und implementieren Stories.

Diese Definition ist offensichtlich eine Provokation, da die aufgeführten Aufgaben wenig mit dem zu tun haben, was traditionell von einem Architekten erwartet wird: die Architektur zu definieren. Hier wird etwas von Anfang an gebaut, „das Einfachste, das funktionieren könnte“, und der Architekt kommt nur hinzu, um Refactoring durchzuführen (also die Architektur zu verbessern, wenn sie nicht befriedigend ist), zu testen und, wie jeder andere im Team, die User Stories zu implementieren.

Einige Autoren akzeptieren widerwillig, dass es andere Leistungen geben kann als Tests. Cockburn zum Beispiel beschreibt, welche Resultate von den Entwicklern gerne gesehen werden:

***Es werden keine Elemente gewürdigt, die nicht im ausführbaren, getesteten Code resultieren****. Okay, es werden gerne* ***endgültige Abgaben*** *gesehen wie Trainingsmaterial oder Abgabedokumentation.*

Beachte das wiederwillige „Okay“: es kann einige Ausnahmen geben, aber der Code und die Tests sind die eigentlich interessanten Gegenstände.

***Minimalismus: eine Beurteilung***

Die Beharrung auf minimaler Software, in den drei eben beschriebenen Formen führt uns zu einem der absurdesten und schädlichsten Beiträge der agilen Methoden.

Wie immer gibt es auch einige Wahrheit in der agilen Kritik der traditionellen Projekte, in diesem Fall ihre Neigung zum Aufblasen. Projekte und Produkte tendieren dazu, zu viele Features einzubauen. Die Kritik an der Schreibarbeit und unnötigen Dokumenten ist auch teilweise berechtigt. Viele Dokumente, die in Unternehmen nach einem festgelegten Schema produziert werden sind bereits unnötig am Tag ihrer Veröffentlichung oder spielen nur eine unwichtige Rolle. Es ist ebenfalls wahr, dass am Ende der Code das ist was zählt, nicht UML oder GANTT-Diagramme.

Nichts davon rechtfertigt das Verzichten auf Vorplanung im Gesamten.

Zunächst, viele der Probleme die mit dem Aufblasen in Verbindung stehen, sind einfach das Resultat von schlechtem Management. Ein kompetenter Projektmanager weiß, wie man „creeping featurism“ bekämpft, und dass man ständig fragen sollte, ob das Feature wirklich benötigt wird. Die Frage, „Wollen sie alles oder wollen sie es jetzt?“, die einer Erpressung ähnlich klingt, bewirkt Wunder.

Klassische Anforderungsanalyse – Die Art von Tätigkeit, die, um den Agilisten die Angst wegzunehmen, am Anfang des Projekts stattfindet, damit wir überlegen, bevor wir schießen – ist genau dafür gedacht, die Anforderungen der vielen verschiedenen Stakeholders auf einen gemeinsamen Nenner zu bringen und Prioritäten festzulegen. Wenn du eine lange Liste mit Features vorgesetzt bekommst, und für jede von ihnen ein Stakeholder der Meinung ist, sie wäre absolut essenziell, ist ein Trick der gut funktioniert jedem virtuelle 100$ zu geben und sie auf die Features verteilen zu lassen, die sie am meisten brauchen. Die wirklich kritischen kommen dann schnell zum Vorschein.

Der Nachteil der Strategie „baue das Einfachste, das funktionieren könnte“ ist, dass es dazu verleitet, immer die am tiefsten hängende Frucht zu wählen: die Features die am einfachsten implementiert werden können und dabei einen vorführbaren Effekt erzeugen. Projekte, welche dieser Strategie folgen, funktionieren gut, *bis auf das Ende*. Während des Projekts sind alle glücklich; Die Entwickler liefern Demonstrationen mit gutem Gefühl und der Kunde fühlt sich auf der sicheren Seite. Am Ende ist es, weil ein schweres, aber unumgängliches Problem auftaucht, unmöglich, ein befriedigendes Ergebnis auszuliefern.

Als am 1 Oktober 2013 die ersten „Obamacare“ Gesundheitsbörsen in Betrieb genommen wurden, wurden sie zur Freude ihrer Gegner zum Opfer ihres eigenen Erfolgs. Es war fast unmöglich durchzukommen, geschweige denn, eine Versicherung abzuschließen. Es wurde auf den hohen Grad des Interesses geschoben, klang aber eher nach einer schwachen Ausrede für den Mangel an Engineering; schließlich haben einige kommerzielle Seiten einen viel höheren Grad an Traffic und sind auch noch um ein Vielfaches komplexer. Wenn man nicht gerade einen hohen Grad an Inkompetenz annimmt, muss man davon ausgehen, dass die ganzen User-Stories während des Testens den Anschein gemacht haben, zu funktionieren. Es war einfach nicht genug Architektur und vorausschauendes Denken vorhanden, um das System auf diesen Maßstab zu vergrößern.

Hier ist ein weiteres Beispiel in der Tradition der agilen Anekdoten, für die Gefahren, die der stückweise-Ansatz mit sich bringt - diesmal nicht aus der Softwarewelt. Der erste Einsatz der Boeing 787 „Dreamliner“ in 2013 war, wegen der gefährlichen Problemen mit den Akkus, ein Desaster; die Flugzeuge mussten für einige Monate aus dem Verkehr gezogen werden. James Surowiecki hatte in seiner Analyse für den *New Yorker* folgendes zu sagen:

*Angetrieben durch den Wunsch, den Dreamliner schnell an den Kunden zu bringen, hat Boeing schon viele von ihnen gebaut, ohne zunächst auf die Erlaubnis der Federal Aviation Administration (Luftfahrtbehörde der USA) zu warten, die den Flugzeugen überhaupt erst erlauben würde abzuheben; danach mussten sie einige Schritte zurückgehen und die Flugzeuge so überarbeiten, dass sie den Anforderungen der FAA entsprechen. Wenn es ein Sprichwort gibt: „Prüfe zweimal, baue einmal“ war das eher ein „Baue zweimal und prüfe einmal“, sagte zu mir [ein Industrieanalytiker]. „Mit all dem Zeit- / und Kostendruck war es eine chemische Formel für Probleme.“*

Das sind nur Beispiele. Aber sie bestätigen, wie naiv es ist, zu erwarten, dass „Refactoring“, sobald man etwas hat, das teilweise funktioniert, alle übrigen Probleme beheben kann. Diese Probleme können sehr schwerwiegend sein, zum Beispiel ein großes Performanceproblem, das nicht behoben werden kann, ohne dass die ganze Architektur umgebaut werden muss. Erfahrungswerte bestätigen diese Vermutung. Boehm und Turner schreiben:

*Die Erfahrung zeigt, dass während die Projekte größer werden, kein Verlass auf billiges Refactoring ist.*

und

*Die einzigen Quellen von empirischen Daten, die uns in die Hände kamen, waren von weniger erfahrenen frühen Anwendern, die rausfanden, dass sogar für eine kleine Anwendung der prozentuelle Aufwand an Refactoring und Fehlerbehebung mit der Größe der Anforderungen wächst.*

In Software, wie auch in jeder anderen Art vom Engineering, ist es gut, mit verschiedenen Arten von Lösungen zu experimentieren, aber es ist auch von großer Wichtigkeit, eine angemessene Vorbereitung durch Vordenken zu treffen, um nicht mit den falschen Entscheidungen anzufangen.

Einige der größten Projektkatastrophen kamen dadurch zustande, dass der Kunde oder der Manager etwas, das funktioniert, gleich von Anfang an sehen wollte („es macht nichts, dass noch nicht alles da ist, zeig mir einfach eine Beispielausführung“) und dann den Entwicklern Vorwürfe machte, dass sie an der Infrastruktur gearbeitet haben, statt gleich sichtbare Ergebnisse vorweisen zu können – mit anderen Worten, sie machten einen verantwortungsvollen Job eines Softwareprofessionellen. Die Manager bekamen ihre Demos und dann nichts mehr, weil der Fokus auf das Liefern sichtbarer Funktionalität lag und die großen Probleme wiederholt zur Seite geschoben wurden. Jedes Mal wenn die nächste Funktionalität oder die nächste Skalierungsebene hinzugefügt werden sollte, musste das Team das ganze Design neu machen, weil die Bemühungen eine gute Generalisierung und Infrastruktur zu erschaffen, gescheut wurden. Zwangsläufig sank die Moral, die Stakeholder verloren ihr Vertrauen und früher oder später wurde das gesamte Projekt begraben.

Für sichtbare Ergebnisse zu argumentieren ist gerechtfertigt, aber nicht wenn der Preis dafür das Verletzen einer der fundamentalen Engineering-Sorgen ist: des Risikomanagements. Es ist das Kennzeichen für ein gut geführtes Projekt, wenn gleich am Anfang die kritischen Pfade erkannt werden, welche den Untergang für das Projekt bedeuten können, wenn sie nicht richtig behandelt werden. Eine Aufgabe mit hohem Risiko, kann eine fundamentale Funktionalität sein, welche auch früh demonstriert werden könnte, oder es kann eine Skalierbarkeitsanforderung sein (die Webseite soll die Menschenmengen aushalten können, welche auf sie zukommen, wenn sie von Oprah im Fernsehen erwähnt wird), die nur durch tiefes Einsehen in den Code entdeckt und nicht schon früh demonstriert werden kann. Den Fokus auf das Sichtbare auf Kosten des essenziellen zu legen, ist unverantwortlich.

***Additive und multiplikative Komplexität: die Lasagne und die Linguini***

Trotz des Arguments für das Aufstellen einer soliden Basis am Anfang, folgen die agilen Methoden weiter dem Motto „bring etwas jetzt zum laufen“. Ein typisches Beispiel ist ein Lehrvideo über Scrum, welches einen Manager im Anzug zeigt, der zu einem Entwickler mit einem Schal und einem Flieger folgendes sagt:

*Ich kann mit etwas einfachem, das richtig funktioniert leben. Die Komplexität kann später eingebaut werden.*

Das Beste, das man hier sagen kann ist: nun, wenn es das ist was du glaubst dann viel Glück. Dieses Glück wirst du haben, wenn die Komplexität die Form von Details hat, die nach und nach hinzugefügt werden können. Das ist die erste Art von Komplexität und sie kommt vor. Wir können sie **additive Komplexität** nennen. Sie existiert dann, wenn das Grundproblem einfach ist, sagen wir zum Beispiel, berechne die Steuern als einen Teil des Preises und dann gibt es noch viele Spezialfälle die hinterher hinzugefügt werden können.

Aber es gibt auch eine andere Art, welche wir **multiplikative Komplexität** nennen können. Sie existiert dann, wenn das fundamentale Problem schon komplex ist und man keine akzeptablen Ergebnisse bekommen wird, bis man alle Schlüsselelemente in Betracht zieht. Ein Beispiel wurde früher zitiert: Unterstützung für eine mehrsprachige Benutzeroberfläche ist im Nachhinein viel schwerer hinzuzufügen, als wenn sie von Anfang an integriert worden wäre.

Komplexität kommt in allen Fällen aus der Anhäufung der Features, die integriert werden sollen. Den Unterschied macht die Art wie sie miteinander interagieren.

* Bei additiver Komplexität stapeln sich die einzelnen Features wie die Schichten einer Lasagne. Dann ist es in Ordnung, über die ersten nachzudenken und die anderen anschließend drauf zu machen.
* Bei multiplikativer Komplexität verstricken sich die einzelnen Features wie die einzelnen Nudeln in einer Schüssel Linguini (oder Spaghetti).

Pamela Zave vom AT&T, welche einen Großteil ihrer Karriere dem Studieren von Interaktionen zwischen Features gewidmet hat, angefangen bei Telekommunikationssoftware, schreibt:

*Historisch gesehen, haben die Entwickler von Telekommunikationssoftware kein effektives Verständnis für das Verwalten und Verstehen der Interaktionen zwischen den Features. Als Folge dessen waren Interaktionen zwischen den Features oft ein Fass ohne Boden, was die Komplexität, Softwarebugs, Kosten-/ und Zeitplanüberschreitung anging, ebenso wie schlechte Benutzererfahrungen. Entwickler anderer Softwaresysteme müssen realisieren, dass auch sie ein Problem haben mit der Interaktion der Features haben.*

Sie gibt ein typisches Beispiel:

Denken wir an die Behandlung von „Beschäftigt“ in der Telefonie, wie dem Weiterleiten an andere Nummern, dem Unterbrechen des Angerufenen, einem späteren Anruf oder dem Angebot, eine Sprachnachricht zu hinterlassen. Nehmen wir an, wir hätten eine Sprache zur Beschreibung der Features dieses Falles, indem diese die Handlung, Anfangsbedingung und die Priorität angeben würde. Weiter nehmen wir auch an, wir hätten einen Operator, der für die Zusammensetzung der Features verantwortlich ist, und sicherstellt, dass in einer Beschäftigt-Situation, die angewandte Handlung diejenige sein wird, welche die höchste Priorität hat.

*In einer Beschäftigt-Situation, mit zwei Behandlungsarten B1 und B2, wobei beide eingeschaltet sind und B2 die höhere Priorität hat, würden diese Features interagieren. B1 wird nicht angewandt, obwohl die eigene Beschreibung von B1 besagt, dass es eigentlich angewandt werden sollte.*

Solche Fälle zeigen, wieso wir nicht einfach annehmen können, dass wir „die einfachste Sache, die funktionieren könnte“ machen werden und dann alle Features, die wir brauchen, hinzufügen. Wenn wir es tun werden wir in Konflikt geraten mit Sachen, die wir zuvor gemacht haben und werden die Arbeit von vorne anfangen müssen. Stellen wir uns einen standardmäßigen, agilen, User-Story basierten Ansatz für das Problem vor. Eine Benutzer-Story im empfohlenen agilen Stil wäre:

(User-Story #1) Als Führungskraft möchte ich eine Weiterleitungsoption, sodass das Telefonat an meine Sekretärin weitergeleitet wird, wenn ich beschäftigt bin.

Ein wenig später, wenn wir über Prioritäten denken, können wir uns eine weitere Story zusammenbrauen:

(#2) Als System-Konfigurator möchte ich die Prioritäten der verschiedenen Behandlungsarten festlegen können.

Und wenn die Zeit vergeht kommen noch einige dazu:

(#3) Als Verkäufer möchte ich mir sicher sein, dass, sobald ich von einem Interessenten angerufen werde, während ich mich in einem anderen Gespräch befinde, das Gespräch unterbrochen wird, sodass ich sofort das Interessentengespräch annehmen kann.

(#4) Als verantwortungsvoller „Anrufbeantworter“ möchte ich sichergehen, dass ich die Option habe, Leute die mich angerufen haben, während ich beschäftigt war, zurückzurufen, sobald mein Gespräch zu Ende ist.

Andere werden folgen. Alle sind vollkommen sinnvoll, aber wie Zave hinweist, kann man sie nicht einfach einzeln betrachten. Einige Szenarien aus den vierzehn (!), die sie angegeben hat:

*Bob hat das „Weiterleiten“-Feature eingeschaltet und leitet alle Anrufe weiter an Carol. Carol hat „nicht-stören“ eingestellt. Alice ruft Bob an, der Anruf wird an Carol weitergeleitet und Carols Telefon klingelt, weil „nicht-stören“ nicht für weitergeleitete Telefonate gilt.*

*Alice ruft eine Verkaufsgruppe an. Ein Feature der Verkaufsgruppe wählt Bob als den Vertretenden im Dienst und leitet den Anruf, entsprechend der persönlichen Angaben, an Bob weiter. Bobs Telefon ist ausgestellt, also antwortet sein Anrufbeantworter und bietet an, eine Nachricht zu hinterlassen. Es wäre viel besser, das Verkaufsgruppenfeature zu reaktivieren, um einen neuen Vertreter zu finden.*

*Ein neuer Mobilitätsdienst wird den Büromitarbeitern angeboten. Wenn Alice sich dort anmeldet, wird ihre Büronummer an ihren Mobilitätsdienst weitergeleitet. Bei einem Anruf an Alice leitet der Dienst den Anruf auf das, wo Alices persönliche Daten hinführen. Wenn diese Daten jedoch sagen, Alice befände sich in ihrem Büro, landet man in einer Endlosschleife von Weiterleitungen.*

Das sind typische Beispiele, die zeigen, wieso der Ansatz mit einer Grundfunktion anzufangen und dann nach und nach Features hinzuzufügen, in einem Desaster enden kann. Und doch ist es das agile Mantra, ausgedrückt beispielsweise durch ein Zitat von Poppendieck, welches ein Kapitel von Cohn eröffnet:

*In heutiger Zeit programmieren wir Software nicht mehr Modul für Modul, sondern Feature für Feature.*

Bei durchschnittlicher Software – vielleicht. Bei komplexen Sachen (der multiplikativen Art) ist ein systematischer Ansatz unumgänglich. Solch ein Ansatz erfordert Überlegungen, vermutlich große, und egal, ob du es magst oder nicht, dies sollte am besten schon am Anfang geschehen.

Die agile Annahme, man könnte Programmfeatures jederzeit inkrementell hinzufügen, ist für solche komplexen Systeme nicht praktikabel. Hier stoßen wir an die Grenzen des agilen Ansatzes.

***Die Rolle der Dokumente***

Wörtlich genommen, ist Poppendiecks Zurückweisung von Dokumenten

*Sobald das System abgeschlossen ist, kümmern den Benutzer die Zwischenprodukte nicht sehr viel.*

wertlos. Sicher, ein Teenager, der eine Textnachricht auf seinem Smartphone versendet, „kümmert sich nicht viel“ um die Anforderungen und Analysedokumente, die während der Entwicklung erstellt wurden, aber das ist auch für jedes andere Zwischenartefakt der Fall, den Code selbst miteingeschlossen! Wir könnten auch sagen, dass sich der Benutzer eines Autos und der Inhaber eines Hauses, wenig für die Verbrauchsgüter des Produktionsprozesses interessieren, aber das bedeutet nicht, dass die Artefakte nutzlos waren. Die Frage ist nicht, ob sie für die Benutzer nützlich sind, sondern ob die Programmierer sie gebrauchen können, zum Beispiel für die Wartung des Systems.

Man kann sich fragen, ob Poppendieck „Kunde“ und nicht „Benutzer“ meinte. Aber dieselben Beobachtungen gelten in beiden Fällen. Die Entwickler sind der Interessentenkreis.

Die Kritik an Dokumenten muss auf besseren Argumenten basieren. Das Schlüsselproblem ist die **Veränderung**. Software, wie uns das agile Manifest erinnert, wird sich verändern. Wenn das Projekt Anforderungs- / und Entwurfsdokumente produziert, sind sie schwer mit dem Artefakt, synchron zu halten, welcher das letzte Wort hat: dem Code. Diese Beobachtung ist es auch, die den Vergleich mit anderen Disziplinen verhindert: Software ist einzigartig in der Geschwindigkeit in der wir sie ändern können und in der Abwesenheit der *Produktionskosten*. Einer der Gründe, wieso der Automobilbau nicht ohne Pläne und Dokumente funktionieren kann ist, dass sobald man ein Design hat, man viele Kopien davon produziert (von den Autos); das Design zu verändern ist eine umfangreiche Entscheidung und eine teure, da man auch den Produktionsprozess auf den neuen Stand bringen muss. Der Part „Soft“ in „Software“ befindet sich dort aus einem Grund: Wir können unser Programm nach Laune verändern. Wenn Dokumente es beschreiben, ist es immer schwer sie auf dem Laufenden zu halten. Tatsächlich versuchen es die meisten Projekte nicht einmal. Das ist das Hauptproblem mit den Anforderungen, Entwürfen und anderen Dokumenten.

Moderne Softwaretechnologie bietet Antworten, wie das „Single-Product-Principle“ (Single Product Principle: Behandle die Software wie eine einzige Entität, von welcher verschiedene Sichten abgeleitet werden können – beschrieben von Bertrand Meyer), welches (um nicht mein eigenes Buch zu beschreiben) hier nicht weiter erläutert wird. Sogar mit diesen Techniken ist das Risiko der Veränderung jedoch kein Grund, um die Dokumente zu verwerfen.

***Was ist Einfachheit?***

Ein weiteres agiles Mantra, das genauerer Analyse wert ist, ist die Einfachheit. In den vorangegangenen Unterkapiteln haben wir uns die spezifischen Konsequenzen auf der Suche nach der Einfachheit angeschaut: Die Vorschrift, minimale Funktionalität zu entwickeln, nicht mehr als für das Produkt benötigt wird und nur den Code und die Tests. Wir sahen den ständigen Druck auf „das Einfachste, das funktionieren könnte“ und „du wirst es nicht brauchen“. Um diese Rezension des agilen Minimalismus jetzt abzuschließen, ist es nötig, einen genaueren Blick auf das Konzept der Einfachheit zu werfen und Unklarheiten im Bezug auf die „offiziellen“ agilen Prinzipien zu beseitigen, die Einfachheit als „*Die Kunst - die Menge nicht getaner Arbeit zu maximieren*“ definieren.

Wie jeder, der irgendwann einmal ein Problem irgendeiner Art gelöst hat, die Lösung zu kompliziert fand, und versucht hat, diese zu vereinfachen, weiß, um Einfachheit zu erreichen, muss oft viel Arbeit investiert werden.

In einem Interview im Jahr 1998, sagte Steve Jobs der *Buisness Week*:

*Das war eines meiner Mantras – setze die Einfachheit in den Mittelpunkt. Einfach kann schwerer sein als Komplex: Man muss schwer daran arbeiten, um seine Gedanken frei zu bekommen und es einfach zu machen. Aber das ist es wert, denn wenn du es mal erreicht hast, kannst du Berge bewegen.*

Einige Generationen zuvor erzählte Antoine de Saint-Exupéry etwas ähnliches, im Bezug auf die Flugzeugherstellung:

*Es macht den Anschein, als hätte der ganze industrielle Aufwand, die ganzen Berechnungen, die ganzen Nächte, die für Entwürfe drauf gegangen sind, alle zu einem sichtbaren Ergebnis geführt haben: Einfachheit – als ob die Erfahrung mehrerer Generationen nötig war, um dieses Wissen zu extrahieren. Es sieht aus, als würde die Arbeit der Ingenieure, Zeichner und Techniker nur poliert und wieder ausradiert werden, bis sie eine perfekte Form, frei von Kruste, mit der spontanen Qualität eines Gedichts,, erreicht hat.* ***Es sieht so aus, als wäre die Perfektion nicht dann erreicht, wenn nichts mehr hinzuzufügen ist, sondern wenn es nichts mehr gibt, das man weglassen kann.***

Wenn Michelangelo Einfachheit mit nicht getaner Arbeit gleichgesetzt hätte, hätte er auch einfach den Marmorblock in Ruhe stehen lassen können, statt ihn zu schlagen, um David hervorzubringen („Der David steckte von Anfang an in dem Marmorblock. Ich habe nur entfernt, was nicht dazu gehörte“). Na gut, ich werde nun aufhören, Zitate von bekannten Menschen aus verschiedenen Jahrhunderten einzuwerfen, damit man mich nicht nach Beweisen für die Zitate fragt. Alle diese Autoren drücken jedoch eine fundamentale Beobachtung aus: Einfachheit zu erreichen, bedeutet nicht dasselbe, wie Arbeit minimieren. Bei beiden handelt es sich um lobenswerte Ziele im Softwareengineering, doch sie tauchen im verschiedenen Kontext auf und führen zu verschiedenen Prinzipien.

* Einfachheit wurde lange von Befürwortern der gründlichen, eleganten Programmiertechniken, wie Dijkstra, Wirth, Hoare, Gries und Parnas unterstützt. Sie setzen es oft gleich mit dem Einsatz einfacher mathematischer Modelle der Programme, keine Sorge der agilen Autoren.
* Unnötige Arbeit zu vermeiden ist, wie wir gesehen haben, ein Schlüsselthema in der agilen Literatur. Es führt zu solchen Prinzipien wie „vermeide Abfall“ und „entscheide so spät wie möglich“ in der Lean Software.

Diese zwei Ansichten begegnen sich, aber nicht unbedingt in der Art, wie es die agilen Autoren gerne hätten. Wirth hat im Jahr 1995 ein Plädoyer für Lean Software veröffentlicht – man bemerkt das Wort „Lean“- in dem er die Ansammlung von unnötigen Features in modernen Softwareprodukten kritisiert hat und kleine, schlüssige Systeme befürwortet hat. Aber, um zu beschreiben, wie diese Einfachheit erreicht werden könne, schrieb er einfach:

Ein erfahrener Ingenieur, der festgestellt hat, dass ein kostenloses Mittagessen nie kostenlos ist, wird sich jetzt fragen: wo ist der Preis für diese Wirtschaft versteckt? Eine vereinfachte Antwort ist: in einer klar konzeptionierten Basis und einer durchdachten, angemessenen Systemstruktur.

*Wenn der Kern – oder jedes anderen Modul – erfolgreich erweiterbar sein soll, muss sein Designer genau verstehen, wie er benutzt wird. Tatsächlich ist der aufwendigste Aspekt des Systementwurfs die Zersetzung in Module. Jedes Modul ist ein Teil mit genau definierten Schnittstellen, welches die Imports und Exports spezifiziert.*

Mit anderen Worten: man muss schon sehr früh sehr stark überlegen. So viel zum Thema, entscheide so spät wie möglich und baue ein System, indem ein Feature nach dem anderen implementiert wird.

**4.4.5 Akzeptiere Veränderungen**

Die Welt und unsere Weltanschauung verändern sich; dasselbe gilt für Software Systemanforderungen. Den Kunden direkt in das Projekt miteinzubeziehen, führt sehr wahrscheinlich zu sogar mehr Veränderungen.

Das agile Manifest spricht vom „Begrüßen“ der Veränderungen, nicht einfach nur vom Akzeptieren. Das ist eine Übertreibung. Es ist eine Sache zu sagen, dass Veränderungen ein normales Phänomen in der Softwareentwicklung sind und eine ganz andere, auf mehr Veränderungen zu *hoffen*. Schließlich verursacht es immer mehr Arbeit, wenn eine Funktionalität korrekt implementiert wurde, eine Veränderung der Anforderungen zu akzeptieren, als wenn man beim Original bleibt. Als Vergleich, bedenken wir, wie erfolgreich der Hotelbuchungsservice booking.com wurde, nachdem sie den Kunden erlaubt hatten ihre Buchungen ohne Geldstrafe zu ändern: es ist schwer vorstellbar, dass die Angestellten des Unternehmens morgens in die Arbeit mit dem Wunsch kommen, es würde mehr Kunden geben, welche heute ihre Meinung ändern! Die Geschäftspolitik kann durchaus profitabel sein, doch jede Veränderung verursacht immer Ärger.

Ein klares Zeichen dafür, dass die agilen Methoden Veränderungen viel mehr „akzeptieren“ als „begrüßen“, abgesehen davon, was das Manifest behauptet, ist, dass sie in der Praxis die Veränderungen einschränken. Scrum zum Beispiel, hat eine strenge Regel – wir werden sie die *geschlossenes Fenster Regel* nennen – welche dem Product Owner und allen anderen verbietet, die Produktanforderungen während der Projektentwicklungsphase (des Sprints) zu verändern, oder neue hinzuzufügen.

Trotz des Hasses, gegen die traditionellen Methoden und das Wasserfallmodell, scheint Scrum sich in die Seite der üblichen Softwareengineering Weisheiten zu stellen. Im Gegenteil zur Karikatur die in agilen Texten zu finden ist, hat die Softwareengineering Literatur schon lange die Notwendigkeit der Veränderungen als lebenslangen Prozess anerkannt; es sagt einfach, dass Veränderungen angemessen behandelt werden müssen. Nur ein naives Team würde ungezügelt Veränderungen zu jedem Zeitpunkt akzeptieren, geschweige denn willkommen heißen – und es würde nicht viel fertige Software liefern. Scrum benutzt seine eigenen spezifischen Regeln für den Umgang mit Veränderungen: akzeptiere Veränderungen außerhalb der Sprints. Es ist eine vernünftige Grundregel, gänzlich im Geist der traditionellen Prinzipien und Praktiken.

Die begeisterte Akzeptanz der Veränderungen ist eine erfrischende Abweichung von der Mentalität vieler Manager, die einen strickten Prozess-basierten Ansatz gewöhnt sind, in welchem die einzigen guten Anforderungen die eingefrorenen sind, und welcher Veränderungen als Plage sieht. Die Verbreitung der agilen Ideen hat eine große Rolle in der Veränderung dieser Einstellung gespielt.

Die Notwendigkeit Software zu produzieren, die einfach verändert werden kann, namentlich **Erweiterbarkeit**, ist keine neue Angelegenheit. Tatsächlich war dies seit Dekaden ein Kernthema der Softwareengineering-Diskussionen. Während das agile Manifest zurecht die veränderungs-bereite Denkweise befürwortet, ist das Hauptproblem mit der Erweiterbarkeit nicht psychologischen sondern technischen Ursprungs: welche Softwaretechniken können wir anwenden, um sicherzustellen, dass wir nicht alles von vorne anfangen müssen, wenn der Kunde seine Meinung ändert oder eine der Domäneneigenschaften sich verändert?

Der agile Ansatz hat eine wichtige Idee zur fortgeschrittenen Erweiterbarkeit in der Praxis beizutragen: die Regel des Extreme Programming, die besagt, dass **jedes Stück Funktionalität einem Testfall zugeordnet** werden sollte. Einer der Hindernisse beim Verändern ist das Risiko, eine der vorausgegangenen Funktionen des Systems kaputt zu machen, vor allem wenn das Problem spät gefunden wurde. Mit einer Regressions-Suite, welche nach jeder Veränderung getestet wird, sinkt dieses Risiko erheblich.

Abgesehen von dieser Regel, liefern jedoch die agilen Methoden wenig Beitrag zur Erweiterbarkeit und befürworten eher Techniken, die *dagegen* sind. Die agile Einstellung auf Softwareveränderungen zu analysieren zeigt, dass es, mit den Worten von Facebook, „*kompliziert ist*“. In großen Absichtserklärungen heißen die Agilisten Veränderungen „willkommen“; aber wenn es dann zu technischen Problemen kommt, nehmen sie oft eine höhnische oder feindselige Einstellung gegenüber Ideen an, welche erweiterbare Software produzieren.

Ein Beispiel für eine solche Idee ist eine ganze Softwareentwicklungsmethode, die mit dem Zweck entwickelt wurde, Erweiterbarkeit zu unterstützen, nämlich die objektorientierte Softwareentwicklung, mit ihrer Erzwingung der Abstraktion, des Information-Hiding, der Generik, des Polymorphismus, des dynamischen Bindens und anderer konkreter Mechanismen die direkt dafür entworfen wurden, um die Softwareveränderungen zu erleichtern. Die Pollendiecks behaupten, dass die Objektorientierung nicht abliefert:

*Während* ***in der Theorie*** *die Objektorientierung Code produziert, der einfach zu verändern ist, können objektorientierte Systeme in der Praxis genauso schwer zu verändern sein, wie alle anderen, vor allem wenn Information-Hiding nicht vollständig verstanden und falsch angewendet wurde.*

aber sie schlagen keinen besseren Weg vor, wie man Erweiterbarkeit erreichen könnte.

Ihr Kommentar ist rätselhaft: welche Art von „objektorientierter Entwicklung“ kann es ohne den richtigen Einsatz von Information-Hiding geben, einer der definierenden Charakteristiken dieser Methode? Des Weiteren kann jeder Ansatz abgewiesen werden, anhand von schlechten Resultaten von Leuten, die ihn nicht „tiefgreifend verstanden haben“ und ihn „nicht effektiv einsetzen“. Weisen wir die Idee des Autotransports ab, weil es schlechte Fahrer gibt? Oder sollten wir Lean Software abweisen, eine von den Autoren befürwortete Methode, wenn wir auf jemanden stoßen der sie nicht richtig verwendet? Ein weiteres Beispiel für bizarre agile Logik.

Das agile Problem mit Veränderungen ist nicht auf solche herablassende und unnötige Kommentare beschränkt. Einige agile Prinzipien und Praktiken verletzen gezielt die Erweiterbarkeit. Das markanteste Beispiel ist das Streben nach minimaler Software, welches uns dazu ermutigt, wie in vorangegangenen Kapiteln beschrieben, „nur das Produkt zu bauen, das benötigt wird“. Wir wurden YAGNI unterzogen, informiert, dass es „eine Verschwendung“ ist, Code, der „zurzeit nicht benötigt wird“ einzuschließen, und es wurde angeordnet, nicht immer „den allgemeinsten Fall“ zu behandeln, sondern nur das Programm für hier und jetzt. Dieser Ansatz ist jedoch mit dem Ziel, Veränderungen zu unterstützen, inkompatibel. Veränderungsbewusste Entwickler versuchen, vorausschauend zu denken und, wann immer möglich, mehr als verlangt zu bauen, in Erwartung wahrscheinlicher Veränderungen.

Agilisten scheinen den Widerspruch zwischen ihrer noblen Ambition der Unterstützung von Veränderungen und dem Aufzwingen von Prinzipien und dem Erzwingen von Praktiken, welche diese Veränderungen verhindern, nicht zu sehen.

Wie auch in anderen Fällen ist die agile Kritik an einigen der üblichen Praktiken korrekt: Programmierer sollten sich nicht in grenzenlose und unbegründete Generalisierungen verstricken. Aber sie rechtfertigt nicht das Ablehnen des Versuchs – wenn schon nicht „den allgemeinsten Fall“, dann wenigstens eine Ebene allgemeiner als derzeit.

Abgesehen vom negativen Effekt eines solchen übertriebenen Ratschlags, ist das fundamentale Problem bezüglich Veränderungen in der Software architektonisch. Die Einfachheit der Veränderungen kommt nicht von ungefähr: sie erfordert das Konzipieren einer Architektur für Veränderungen. Gute Bücher lehren dich wie, jedoch ist genau derartiges Vordenken genau das was Agilisten ablehnen.

Agile Befürwortung für Veränderungen ist das richtige Ziel – und nur ein Ziel.