Dr. Michael Schwarz

(Michael.Sc.Schwarz@daimler.com)
ist bei der Daimler TSS GmbH in einer zentralen Funktion für die unternehmensweite Ausgestaltung und Entwicklung des Requirements Engineering-Know-hows zuständig. Nach dem Informatikstudium in München arbeitete er 3 Jahre bei sd&m, promovierte in Ulm und ist nunmehr seit 9 Jahren für TSS in diversen klassischen und agilen Projekten beratend und operativ unterwegs.

„Welle oder Teilchen?" Zur Dualität von Anforderungen, auch in agilen Projekten

Oder: Warum gibt es immer noch so viele Missverständnisse, wenn wir von „Anforderungen" sprechen?

Zum Thema Requirements Engineering (RE) gibt es trotz unzähliger Publikationen im Projektalltag immer wieder Meinungsverschiedenheiten und Missverständnisse. Warum ist das so? Kann es dazu nicht endlich abschließende und allgemein akzeptierte Antworten geben? Der vorliegende Artikel nähert sich diesem Phänomen, indem er einige zentrale Dualitäten von Anforderungen beleuchtet, denen man im Projektalltag Rechnung tragen muss. Kein Wunder, wenn der bunte „RE-Wald" von seinen Bewohnern immer wieder anders wahrgenommen wird.

Einleitung

Zum Requirements Engineering (RE) im Kontext agiler wie klassischer Softwareentwicklungsprojekte gibt es bereits unzählige Publikationen, Konferenzen und Schulungen. Trotzdem gibt es im Projektalltag immer wieder – oft auch die gleichen – Missverständnisse und Meinungsverschiedenheiten darüber, was dieses „Ding" namens „Anforderung" nun überhaupt ist, wie es aussehen muss, wozu es in welcher Form nötig ist usw.

Warum ist das so? Gibt es dazu nicht irgendwann endgültige und allgemein akzeptierte Antworten?

Der vorliegende Artikel versucht etwas Licht ins Dunkle zu bringen, indem er einige zentrale Dualitäten von Anforderungen beleuchtet, die im Projektalltag zu beachten sind. Auf der einen Seite erklären sie die Vielfalt unserer aktuellen RE-Landschaft. Auf der anderen Seite geben sie einen Orientierungsrahmen für die nötige Kultivierungsarbeit, wie sie von jedem Softwareentwicklungsprojekt in diesem Bereich zu leisten ist.

Die Dualität des Lichts stellt hierzu eine wunderbar mächtige Metapher bereit, wie auch schon der Beitrag [FHS10] gezeigt hat.

Problem oder Lösung?

Dienen Anforderungen der reinen Problembeschreibung oder sind sie bereits Lösungskonstrukte? Diese Dualität klingt bereits in der klassischen Definition des Begriffs „Requirement" im Software Engineering-Glossar des IEEE an:

- (1) A condition or capability needed by a user to solve a problem or achieve an objective.
- (2) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents.
- (3) A documented representation of a condition or capability as in (1) or (2) (vgl. [IEEE90]).

Auf der einen Seite ist eine Anforderung also etwas, das von einem Benutzer benötigt wird. Auf der anderen Seite aber auch etwas, das ein System erfüllen oder besitzen muss. Auf der einen Seite also die Beschreibung von Bedürfnissen, auf der anderen Seite die Beschreibung eines Systems (das wiederum bestimmte Bedürfnisse erfüllt). In realen Projekten findet man naturgemäß immer beide Arten von Anforderungen:

1. *Problembeschreibungen*: Eine Beschreibung der „Problemsituation" in Form von Aussagen über die Welt des Kunden, der Anwender und sonstiger Stakeholder, ihre Ziele, Erwartungen und Bedürfnisse. Normalerweise in natürlicher Kundensprache geschrieben, wie z. B. „Ich möchte Produkte mit Kreditkarte bezahlen können". Hier tritt ein Requirements Engineer als Autor in Erscheinung, der den Problemkontext zumeist rein textuell erfasst, Kontextinformationen aufnimmt und im Dialog mit den Betroffenen „Wunschlisten" im O-Ton festhält und strukturiert. Sowohl das Vorgehen bei der „Erhebung" als auch die Darstellung des Ergebnisses folgen eher Ad-hoc-Prinzipien. Weder Vollständigkeit noch Konsistenz stehen im Vordergrund.
2. *Lösungskonstruktionen*: Eine Beschreibung des zu erstellenden technischen Systems als Ergebnis eines kreativen

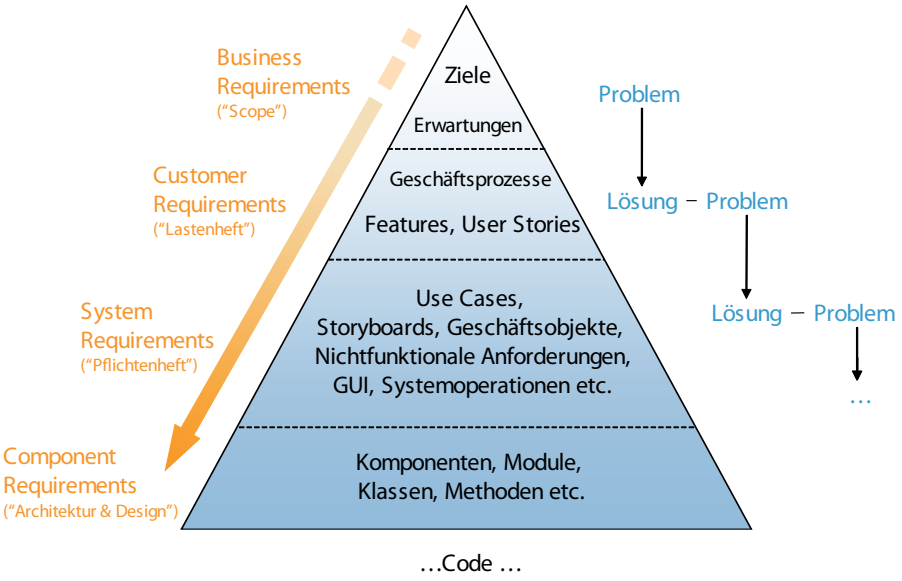

Abb. 1: Die Systementwicklung als kontinuierliche Problemlösung und Konkretisierung von Anforderungen

gemeinsamen Gestaltungsprozesses. Hier treten Requirements Engineers eher als Ingenieure auf, die ein technisches System mittels passgenauer Abstraktionsebenen und bewährten Strukturierungs-, Beschreibungs- und Modellierungstechniken aus konzeptionellen Einzelteilen zusammensetzen bzw. „konstruieren" (z. B. aus Anwendungsfällen, Geschäftsobjekten, Dialogen, Interaktionselementen). Alle erforderlichen logischen Einzelmerkmale des Softwaresystems werden dabei in geeigneter Weise dargestellt, z. B. Dialoge oder Schnittstellen zumeist textuell („Nach Bestätigung des Einkaufs werden folgende Bezahlmöglichkeiten zur Auswahl angeboten: ...") oder konzeptionelle Datenmodelle und Zustandsautomaten per Diagramm. Im Vordergrund steht immer die funktional vollständige und konsistente Beschreibung des zu erstellenden Endprodukts.

In den meisten Projekten finden sich beide Arten von Anforderungen, da man zur Komplexitätsreduktion oft unterschiedliche Abstraktionsebenen nutzt: Zuerst Klärung der Anforderungen auf einer abstrakteren Ebene, dann erst ein Abtauchen bzw. das „Entwickeln" von Anforderungsdetails.

Generell lässt sich der gesamte Entwicklungsprozess von Softwaresystemen als schrittweise Entwicklung, Konkretisierung und Detailierung von Anforderungen verstehen. Eine typische Unterscheidung von Ebenen ist z. B. in **Abbildung 1** dargestellt. Bis zur Ebene der „System Requirements" hat man es dabei ausschließlich mit Anforderungen zu tun, wie sie typischerweise in einem „Pflichtenheft" bzw. einer System Requirements Specificaton (SRS) unterschieden werden. Erst ab der Ebene der Komponentenanforderungen beginnt die Softwarerealisierung (Architektur & Design).

Jede Ebene für sich betrachtet ist sowohl Lösung der Abstraktionsebene darüber als auch Problembeschreibung für die Konkretisierungsebene darunter (siehe dazu auch [FuS10]). Eine Anforderung ist daher immer sowohl Teil einer Problemstellung (für den folgenden Konkretisierungsschritt) als auch Teil einer Lösung (für den vorhergehenden Schritt). Das sind einfach die zwei Seiten derselben Medaille.

Deskriptiv oder konstruktiv?

Zusätzlich lassen sich in der Praxis aber auch innerhalb der gleichen Abstraktionsebene oft sowohl deskriptive (das Problem beschreibende) als auch konstruktive (eine Lösung angebende) Beschreibungselemente finden. Diese lassen sich als alternative und gleichwertige Spezifikationsstile betrachten.

Beispiel Pflichtenheft: Um einen grafischen Eingabedialog zu spezifizieren kann ich das Formular konstruktiv in seinen Einzelelementen und Interaktionsmöglichkeiten Stück für Stück spezifizieren. Oder aber ich beschreibe (Detail-)Aspekte deskriptiv über seine Eigenschaften, wie sie für einen Benutzer erfahrbar sind: „Alle Kundendaten können mit einem eigenständigen Bearbeitungsdialog geändert werden", „Als Benutzer erhalte ich zu jedem Eingabefeld Hilfetexte, sobald sich der Mauszeiger darüber befindet" oder „Alle Datumsfelder werden ausschließlich über Kalender-Pop-ups gepflegt".

Anderes Beispiel „Berechnungsfunktionen": Entweder ich spezifiziere sie konstruktiv/operational mittels Pseudo-Code oder aber deskriptiv nur per Vor- und Nachbedingung (siehe [WaN95], [Mey92] zu „Design by Contract").

Die deskriptive Variante erlaubt im Allgemeinen knappere Formulierungen und wird in der Praxis daher oft auch aus Zeitersparnisgründen eingesetzt. Erkauft wird das natürlich mit Gestaltungsspielräumen, die nachgelagerte Entwicklungsphasen ausfüllen müssen. Deskriptive Spezifikationselemente einer Ebene stellen demnach bewusste Abkürzungen bzw. Unvollständigkeiten im Konstruktionsprozess dar, deren Ausarbeitung in spätere Phasen verlagert wird. Etwas, das man in fast jedem Projekt beobachten kann.

Beide Formen der Darstellung von Anforderungen sind also in der Praxis gleichberechtigte Spezifikationsstile, die sich dort sowohl in unterschiedlichen Abstraktionsebenen niederschlagen können als auch auf der gleichen Abstraktionsebene zu finden sind. Als Konsequenz ergibt sich gerade auch für Werkzeughersteller, dass beide Formen der Anforderungsformulierung geeignet zu unterstützen und zu integrieren sind. Sowohl konstruktive (wie z. B. UML-Modelle, siehe dazu auch den Abschnitt „Informell oder formalistisch" weiter unten) als auch deskriptive – zumeist natürlichsprachliche – Elemente, Darstellungsweisen und Vorgehen sind gleichberechtigt zu behandeln und zu unterstützen, gerade auch in Mischformen. Ein Anspruch, dem die meisten aktuellen RE-Tools nicht gerecht werden.

Zukunft oder Vergangenheit?

Sind Anforderungen nur Zwischenprodukte eines gemeinsamen Gestaltungs- und Entwicklungsprozesses, der als Ziel letztlich das zukünftige Softwaresystem bzw. seine Veränderung hat, oder sollen Anforderungen den Blick in die Vergangenheit ermöglichen, also dem Verständnis dienen, warum das Softwaresystem heute so

aussieht, wie es gerade ist? Zwei Aspekte, die so in Theorie und Praxis heute eher selten getrennt berücksichtigt werden.

Auf die Zukunft gerichtet: Versteht man RE als „vorwärts gerichtet", lässt sich RE auch gut auf Basis der psychologischen Tätigkeitstheorie (englisch „activity theorie", siehe [Leo12] bzw. [DaR97]) verstehen, die zur Erklärung menschlichen Verhaltens entwickelt wurde und sich vor allem aus den Erfordernissen arbeitsteiliger Lebensweisen ableitet. Bevor jemand etwas tut (z. B. codiert), muss klar geworden sein, wozu das genau dienen soll. RE lässt sich hierdurch rein prozessual als notwendiger Schritt der Zieldefinition definieren. Auch wenn im Extremfall keinerlei Anforderungen aufgeschrieben werden, findet in diesem Verständnis trotzdem zwangsläufig ein „vorwärts gerichtetes" RE statt.

Auf die Vergangenheit gerichtet: Steht der Blick zurück im Vordergrund, also das Verstehen, warum das Softwaresystem heute so aussieht, wie es ist, oder der Nachweis, ob es in einer sicheren Weise entwickelt wurde, geht es letztlich um „Wissensmanagement". Es geht dann um die fachliche Dokumentation des aktuellen Softwaresystems, z. B. um die Weiterentwicklung und Wartbarkeit abzusichern, Einarbeitungsaufwand zu minimieren, Änderungsaufwand einfacher abschätzen zu können oder darum, den Nachweis der „funktionalen Sicherheit" sicherheitskritischer Systeme zu erbringen (Stichwort „Traceability"). Hier kommt der angemessenen Darstellung des Wissens eine zentrale Bedeutung zu.

Und dies bedeutet in den meisten Fällen eben nicht, dass man jedes Schnipsel des Entstehungsprozesses aufbewahren muss, indem man z. B. die jeweiligen Change Requests oder Release-Pflichtenhefte 1:1 archiviert, sondern zu einer kompakten verständlichen Beschreibung des Ist-Systems konsolidiert und zielgruppengerecht aufbereitet. Eine effektive Beschreibung von Softwaresystemen sieht fundamental anders aus als eine „Sammlung historischer Fakten". Das Große und Ganze wird oft erst dann sichtbar, wenn man die vielen Details in ihren Zusammenhängen begreift. Und das ist wie in der Geschichtsschreibung auch oft erst im Nachhinein möglich.

Gutes Requirements Engineering muss sowohl in die Zukunft führen als auch den Blick in die Vergangenheit ermöglichen. Während man in agilen Projekten beim Weg in die Zukunft Dokumentationsaktivitäten gut „wegoptimieren" kann (Beispiel Scrum: User Stories umfassen i. d. R. in Form von Akzeptanzkriterien nur wenige Details; alles andere wird vom Product Owner später und viel effektiver mündlich vermittelt) ist auch hier mitunter ein Blick in die Vergangenheit nötig (z. B. für neue Mitarbeiter, neue Kunden, neue Dienstleister, neue Product Owner).

Informell oder formalistisch?

Wie müssen Anforderungen dargestellt werden? Was darf sich überhaupt eine „Anforderung" nennen? Welche müssen dokumentiert werden?

Dazu gibt es sehr viele Ideen und kontroverse Aussagen: Die einen favorisieren den reinen Text und verwalten ihre „Requirements" in Form von durchnummerierten „Sätzen" in einer Excel-Liste (alles andere wie z. B. UML-Diagramme wird der Einfachheit halber der „Entwicklung" zugerechnet). Andere verwenden einen modellbasierten strukturierten Ansatz (siehe Abbildung 2) und nutzen neben Text auch UML-Diagramme in ihren Softwarespezifikationsdokumenten. Dritte verwenden Storyboards, Filme oder Diagramme für die Erhebung oder eine verständlichere Darstellung von Anforderungen. Puristen setzen etwa domänenspezifische oder gar formale Sprachen (Object-Z, VDM, Pseudo-Code) zur konsistenten und mathematisch vollständigen Spezifikation ein, während Agilisten wiederum weitgehend auf die Dokumentation von Anforderungen verzichten und lieber auf die Kommunikation und das frühzeitige Feedback als sehr viel effektiveren Weg der Wissensvermittlung setzen.

Wer hat nun Recht? Die salomonische Antwort: Es kommt darauf an! Und zwar im Wesentlichen darauf,

- auf welcher Gestaltungsebene man sich befindet (geht es z. B. um die Beschreibung geschäftlicher Zielsetzungen, die abstrakte Beschreibung von Leistungsmerkmalen oder aber um eine ausschreibungsfähige Spezifikation des Softwaresystems?),
- welcher Spezifikationsstil (operational oder deskriptiv) und
- welche Darstellungsform gerade am zielführendsten ist (in Bezug auf die Zielgruppen, das gewählte Vorgehen und die gegebenen Randbedingungen) und
- welches Vorgehensmodell man vereinbart hat.

Typischerweise findet man in Projekten daher alle Formen der Beschreibung und

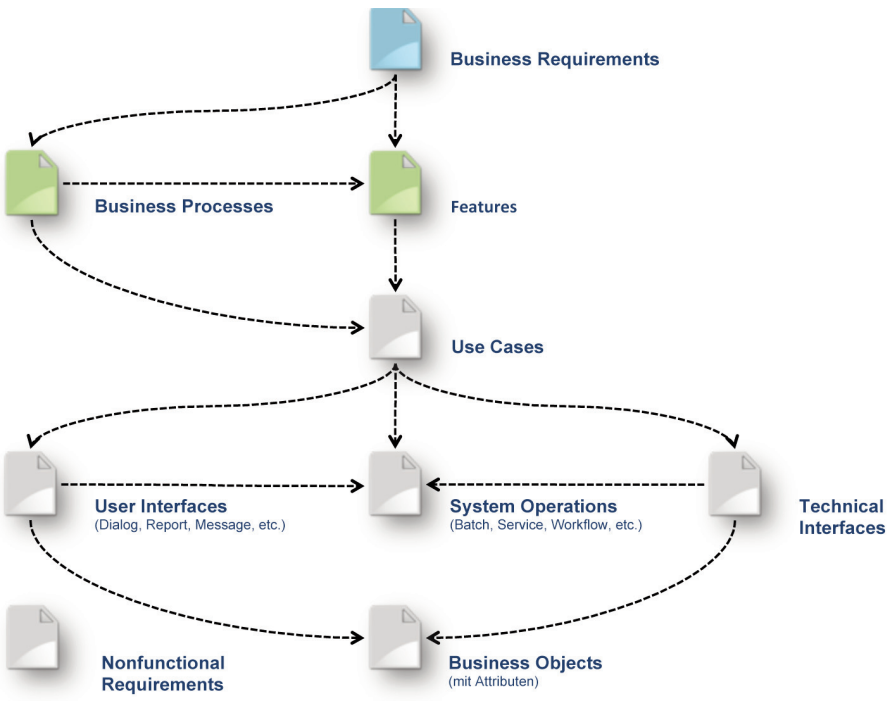

Abb. 2: Typische Anforderungskategorien und ihre logische Erstellungsfolge

das oft gemischt auch auf der gleichen Gestaltungsebene. Das ist nicht per se ein Problem. Es wird nur dann eines, wenn über das Vorgehen nicht bei allen Beteiligten Klarheit herrscht, man die Gestaltungsebenen nicht klar und bewusst trennt oder die Auswahl der Darstellungsform nicht bewusst und einheitlich trifft: Systemdetails haben in einem „Business Requirements Document" nichts zu suchen; bevor man die Systemlösung spezifiziert sollte man die eigentliche Problemstellung verstanden haben; je nach Zielgruppe verbietet oder empfiehlt es sich, mit kompakten Beschreibungsmitteln wie z. B. UML-Diagrammen zu arbeiten; wenn man den einen Ablauf mit BPMN modelliert, sollte man den nächsten nicht mit EPKs beschreiben usw.

Die Antwort auf unsere Eingangsfragen lautet also: Eine Anforderung kann sowohl informellen Charakter besitzen (Gedanke, gesprochenes Wort, Bild, Ton, Film, Prosa) als auch einen formalen (eindeutig identifiziertes und attributiertes Textobjekt, Diagramm, Modellelement). Je nachdem, was im Projekt vereinbart wurde, auf welcher Abstraktionsebene man unterwegs ist und um welche Art von Anforderung es sich handelt, empfehlen sich unterschiedliche Darstellungs- und Vorgehensweisen.

Stabil oder flexibel?

Anforderungen werden letztlich deshalb dokumentiert, damit arbeitsteilige Organisationen eine optimale und verlässliche Arbeitsgrundlage vorfinden: Ein großes verteiltes Realisierungsteam kann nur dann effizient parallel arbeiten, wenn die Anforderungen klar definiert sind; ein System kann nur dann sinnvoll getestet werden, wenn messbare Anforderungen vorliegen; belastbare Kostenaussagen erhält man nur dann, wenn die Schätzgrundlage vollständig und detailliert genug ist; ein Projekt lässt sich nur dann wirklich gut durchplanen und organisieren, wenn die Arbeitsergebnisse feststehen usw. Damit dies alles aber wirklich gut funktioniert, müssen die dokumentierten Anforderungen stabil und verlässlich sein.

Deckt sich das aber mit der Realität? Im Projektalltag hat man es mit lebendigen, d. h. sich ständig verändernden sozialen Systemen zu tun. Sowohl was das Arbeitsumfeld anbelangt, in dem das neue Softwaresystem eingesetzt werden soll, was das Kundenumfeld anbelangt, in dem sich Geschäftsziele jederzeit ändern können, oder das Projektumfeld selbst, in dem kontinuierliche Gestaltungs-, Lern- und Optimierungsprozesse stattfinden.

Die Folge davon ist, dass sich auch Anforderungen an Softwaresysteme jederzeit ändern können und werden, gerade auch während der Durchführung von Softwareprojekten. Und es geht eben nicht darum, solche Anforderungsänderungen zu verhindern, sondern darum, diese zu ermöglichen und optimal auszugestalten.

Das ist ein Paradox, auf das jedes Projekt besonders achten muss, wenn es erfolgreich sein will: Wie komme ich zu stabilen Anforderungen, ohne deren Evolution zu verhindern? Je nach Stabilität der Anforderungen und nötiger Flexibilität sind agile oder klassische Vorgehensmodelle im Vorteil.

Zweck oder Mittel?

Sind Anforderungen ein Selbstzweck oder doch nur ein Hilfsmittel? Und wie sieht das bei Softwaresystemen selbst aus?

Die Idee bzw. die Disziplin des „Software Engineerings" ist in den 60er-Jahren entstanden, einer Zeit, in der der Taylorismus großgeschrieben wurde und effiziente Produktionsprozesse im Vordergrund standen. Man wollte der „Softwarekrise" mit einem ingenieursgemäßen plan- und wiederholbaren Vorgehen mit Erfolgsgarantie begegnen. Bis heute sucht man danach vergebens (siehe z. B. [Bro86]).

Nach Tom DeMarco ist das Software Engineering mittlerweile sogar „eine Idee, deren Zeit gekommen und auch wieder gegangen ist" [DeM08]. Die eigentliche Herausforderung im Software Engineering liegt nach ihm nicht in der Beherrschung und Optimierung der technischen Konstruktion von Softwaresystemen, sondern in ihrer erfolgreichen Konzeption, gerade auch im Hinblick auf ihre Wirkung auf die reale (Arbeits-)Welt. Innovative Veränderungsprozesse lassen sich aber nicht deterministisch vorausplanen oder standardisieren.

Die Aufgabe der Entwicklung von Softwaresystemen ist in der Regel komplex, nicht nur kompliziert. Und erst recht trifft das auf die Konzeption dieser Software mit Blick auf ihre Wirkung auf die Arbeits- bzw. Lebenswelt seiner Anwender bzw. Betroffenen zu.

Der Erfolg agiler Vorgehensweisen im Vergleich zum klassischen Wasserfallvorgehen lässt sich darauf zurückführen, dass man diesem althergebrachten eher „mechanistischen" Leitbild heute in der Softwareentwicklung ein eher „humanistisch" geprägtes entgegensetzt, das den oben skizzierten Flexibilitätsanspruch oft viel besser erfüllt. Je nach Projektkonstellation können natürlich im Einzelfall aber dennoch auch klassische Vorgehen im Vorteil sein (siehe „Stabil oder flexibel?" oben).

Die beiden vorherrschenden alternativen Leitbilder der Softwareentwicklung lassen sich – bewusst polarisiert – folgendermaßen charakterisieren:

Das „mechanistische naturwissenschaftliche Leitbild"
Softwaresysteme entstehen in einem rationalisierbaren Produktionsprozess. Alle Projektergebnisse wie z. B. auch Anforderungen dienen einem fest definierten und im Voraus vereinbarten (Selbst-)Zweck: Sie stellen notwendige Zwischenprodukte in einem Produktionsprozess dar, den man systematisch optimieren kann und möglichst automatisieren sollte. Die Mitarbeiter sollen möglichst austauschbar sein, damit die „Produktionsmaschine" auch bei unvorhergesehenen „Störfällen" rund läuft. Anforderungen dienen der arbeitsteiligen Produktion, werden als Kontrollinstrument eingesetzt (Kosten, Release-Planung, Qualitätsprüfung) und unterstützen den Wettbewerb (stellen die Vergleichbarkeit von Realisierungsangeboten und -leistungen von Zulieferern oder Mitarbeiter sicher).

Das „organische humanistische Leitbild"
Anforderungen wie auch Softwaresysteme selbst sind nur Mittel zum Zweck. Sie dienen der Erreichung übergeordneter Ziele einzelner Menschen und Organisationen. Maß aller Dinge und damit Grundlage allen Handelns sind die beteiligten Menschen und ihre Bedürfnisse. Im Falle von Softwaresystemen steht deren Wertschöpfung für die Organisation bzw. deren „Endkunden" im Vordergrund. Es werden nur die Funktionen umgesetzt, die aktuell den höchsten Nutzen erbringen. Dieser Nutzen wird in einem ständigen Diskurs überprüft und so laufend an die aktuelle Wirklichkeit angepasst.

Statt der Anforderungsdokumentation stehen im RE die Implementierungsaktivitäten im Vordergrund und damit die Verhandlung und Vermittlung von Entwicklungszielen zum Entwicklungsteam hin. Mittel und Wege der optimalen Vermittlung des notwendigen Wissens sind

ständiger Diskursgegenstand und werden laufend optimiert. Das Team und seine ganz spezifischen Bedürfnisse und die Charakteristiken seiner Mitglieder werden bewusst adressiert, statt sie als „Produktionsmittel" zu sehen und möglichst austauschbar zu machen.

Ein Projekt wird nicht als beherrschbarer „Produktionsprozess" verstanden, sondern als komplexes dynamisches soziales „System", dessen Mitglieder sich ständig in einem gemeinsamen Lernprozess befinden, sich laufend an eine lebendige Umgebung anpassen, sich fortentwickeln und „wachsen" wollen.

In der Projektwirklichkeit wird man weder das eine noch das andere Leitbild in Reinkultur vorfinden. Die Beteiligten bringen alle ihre eigenen Weltbilder mit. In jedem Projekt gilt es daher, zunächst ein gemeinsames Leitbild zu (er)finden und als Projektkultur zu etablieren. Daraus leiten sich alle notwendigen Vereinbarungen bezüglich des Requirements Engineerings im Projekt ab: Wie will man gemeinsam vorgehen? Wie will man zu den Anforderungen kommen? Welche Gestaltungsebenen will man unterscheiden? Welche Anforderungen sollen aufgeschrieben werden? In welcher Form? Wann? Wie detailliert? Wie soll das fertige System fachlich dokumentiert werden? Usw.

Fazit

Was lernen wir aus den beschriebenen Unwägbarkeiten, Paradoxien und vordergründigen Unvereinbarkeiten?

1. Anforderungen treten im RE sowohl als Welle als auch als Teilchen auf: RE hat viel mit kreativer Gestaltung, sprachlicher Kommunikation, Lernprozessen und Veränderung sozialer Systeme zu tun, nicht nur mit der Entwicklung und Verwaltung konstruktiver Zwischenprodukte.
2. Anforderungen ermöglichen sowohl den Blick in die Zukunft als auch den in die Vergangenheit. Beide Blickwinkel bringen ihre eigenen Herausforderungen mit und sind nicht austauschbar, was erfolgreiche Darstellungsmittel und -weisen anbelangt.
3. Bezüglich der Darstellung von Anforderungen erübrigt sich die Frage „Text oder Modell?", da sich beide Formen sinnvoll ergänzen. Beides sind mögliche und sinnvolle Darstellungsformen für Anforderungen. Ein gutes RE-Tool muss beide Formen gleichzeitig und vor allem im Zusammenspiel unterstützen, und das nicht nur auf unterschiedlichen Abstraktionsebenen, sondern auch innerhalb derselben.
4. Meinungsverschiedenheiten, Probleme und Diskussionsprozesse rund um das Requirements Engineering sind für Projekte kein vermeidbares Übel, das durch ein Mehr an Ausbildung oder Forschung jemals gänzlich aus dem Weg geräumt sein wird. Vielmehr ist jedes Projekt darauf angewiesen, seinen ganz spezifischen eigenen Gestaltungsprozess zu entwickeln.

Für ein optimales RE bzgl. eines Softwaresystems ist letztlich immer ein erfolgreiches RE bzgl. des spezifischen Entwicklungsprojekts nötig. Um ein optimal angepasstes RE-Vorgehen zu finden, muss immer wieder die Projektkonstellation selbst analysiert werden: Wie sehen die konkreten Randbedingungen aus, welche Bedürfnisse haben die Beteiligten und Betroffenen, wie sieht ihr „Weltbild" bzw. gemeinsames Leitbild aus? Erst mit diesem Wissen kann das für dieses spezifische Projekt optimale Vorgehen gefunden bzw. mit allen Betroffenen zusammen verhandelt, abgestimmt und laufend fortentwickelt werden.

Das ist letztlich nichts wirklich Neues. Genau das passiert in allen erfolgreichen Projekten, und zwar auf gewisse Weise jedes Mal immer wieder von Neuem.

Und genau dies ist kein irgendwie vermeidbares Übel, sondern eine menschliche Notwendigkeit! ■

Literatur

[Bro86] Frederick P. Brooks: „No Silver Bullet: Essence and Accidents of Software Engineering", 1986.

[DaR97] C. Dahme, A. Raeithel: Ein tätigkeitstheoretischer Ansatz zur Entwicklung von brauchbarer Software, Informatik-Spektrum 20: 5-12 (1997).

[DeM08] „Software-Engineering ist eine Idee, deren Zeit gekommen und auch wieder gegangen ist", Interview mit Peter Hruschka, OBJEKTspektrum 06/2008, Jubiläumsheft - 40 Jahre Software-Engineering.

[FHS10] Karol Frühauf, Thomas Haas, Dr. Helmut Sandmayr: „Dualität der Anforderungen", OBJEKTspektrum Online-Themenspecial Requirements Engineering 2010.

[IEEE90] IEEE Std 610.12-1990 „IEEE Standard Glossary of Software Engineering Terminology".

[Leo12] A. N. Leont'ev: „Tätigkeit - Bewusstsein - Persönlichkeit", Lehmanns Media, 2012.

[Mey92] Bertrand Meyer: Applying „Design by Contract". IEEE Computer, Vol. 25, No. 10 (October 1992) S. 40-51.

[WaN95] Kim Waldén, Jean-Marc Nerson: Seamless Object-Oriented Software Architecture. Analysis and Design of Reliable Systems. Prentice Hall, New York (1995).