Lean Product Software

Lean ist mehr als agil

Dr. Jörg-Volker Müller, Systemum GmbH & Co. KG

Der Begriff „Lean Software Development“ wurde vor mehr als 15 Jahren von dem amerikanischen Ehepaar Mary und Tom Poppendieck geprägt und hat die agile Methodenwelt stark beeinflusst. Darüber hinaus eröffnen die Lean Prinzipien speziell für Embedded Software im Produktgeschäft Chancen, ein erfolgreiches Produkt am Markt zu positionieren und gleichzeitig die Qualität der Software langfristig hoch zu halten.

Aus der Praxiserfahrung des Autors in zahlreichen Modernisierungsprojekten bei Kunden unterschiedlichster Branchen zeigen sich stets die gleichen Muster: Software wird mit jedem neuen Feature immer komplexer – aufgeräumt wird praktisch nie. Wissen verbleibt in den Köpfen der Entwickler, Übersicht schaffende Dokumente fehlen. So kommt irgendwann der Wunsch auf, das System zu modernisieren, um wieder ein übersichtliches, beherrschbares und schlankes System zu erhalten. Nach Abschluss der Modernisierung verfallen die Betroffenen allerdings allzu oft wieder in die alten Handlungsmuster.

Wie lässt sich dies vermeiden?

Mary und Tom Poppendieck haben im Jahr 2003 mit dem Buch „Lean Software Development“ [1] und dem 2007 folgenden „*Implementing Lean Software Development”* [2] sieben grundlegende Prinzipien für Softwareentwicklung definiert und gezeigt, wie diese als Grundlage für funktionierende agile Entwicklungsansätze dienen können.

Im Folgenden werden die sieben Prinzipien von „Lean Software Development“ vorgestellt und mit den agilen Verfahren abgeglichen. Es wird gezeigt, dass diese Prinzipien auch im Produktgeschäft angewendet werden können – und sollten.

Die Prinzipien von Lean Software Development

Die Grundlage von Lean Software Development bilden sieben zentrale Prinzipien, jeweils untermauert mit Werkzeugen und Handlungsempfehlungen.

Prinzip 1: Eliminate Waste

Der Begriff *Waste* (Deutsch etwa „Verschwendung“) bildet die zentrale Grundlage der Lean Prinzipien. Mit *Waste* ist alles gemeint, was keinen Wert zum Ergebnis beiträgt. Ziel muss es sein, jegliche überflüssige Features, Code, Artefakte, Schnittstellen, Aktivitäten usw. zu eliminieren, um so *„lean“* zu werden.

In Bezug auf Software bedeutet *Waste* vor allem:

- **Unfertige Arbeit** – Jede nicht vollständig realisierte Softwarekomponente stellt ein Risiko für das Gesamtsystem dar, weil unklar ist, welche Probleme bei der Integration entstehen. Solange eine Komponente nicht fertig und im produktiven Einsatz ist, kann der Nutzen nicht validiert werden.
- **Zusätzliche Arbeit** – Unnötige Schritte im Prozess, insbesondere überflüssige Dokumentation ohne Wert für das Endergebnis oder die spätere Wartung sind zu vermeiden. Dokumentation, die niemand liest, hat keinen Nutzen.

- **Überflüssige Features und Overdesign** – Entwickler sind kreativ im Erfinden von neuen technischen Mechanismen und „coolen“ Features. Dies wirkt zuerst harmlos, überflüssige Features und Overdesign sind aber in hohem Maße *Waste*: Jedes Stück Softwarecode muss verfolgt, übersetzt, integriert und getestet werden – jedes Mal, wenn der Code angefasst wird und über die gesamte Lebenszeit des Systems.
- **Task Switching** – Entwickler arbeiten konzentriert an ihren Aufgaben. Jeder Wechsel der Aufgabe kostet Zeit, bis man sich in die neue Aufgabe eingedacht hat. Darum sollten Entwickler möglichst fokussiert über einen längeren Zeitraum an ihren jeweiligen Aufgaben arbeiten können.
- **Warten** – Ein erheblicher *Waste* in realen Projekten entsteht durch Warten: Warten auf den Projektstart. Warten auf Abstimmungen über Projektziel und klar definierte Anforderungen. Warten auf neue Entwicklerressourcen. Warten auf Komponenten, die eine andere Abteilung erstellt, etc. Durch Verzögerungen wird verhindert, dass das System schnell Nutzen bringt.
- **Bewegung** – Ein großer *Waste* entsteht durch Übergabe von Dokumenten zwischen Personen oder Abteilungen statt direkter Kommunikation. Dokumente können nicht alle Informationen enthalten, die eine andere Person benötigt. Es ist vielmehr wichtig, dass Entwickler, Tester und Kunden bzw. Produktmanager nah beieinandersitzen. Jede organisatorische Grenze erhöht die Gesamtkosten und verzögert das Ergebnis.
- **Fehler** – Die Auswirkung von Fehlern sind umso größer, je kritischer sie sind und je später sie entdeckt werden. Ein Fehler, der von einem Entwicklertest gefunden wird, ist schnell behoben. Ein kritischer Fehler, der erst beim Nutzer entsteht, kann sehr teuer werden.

Prinzip 2: Amplify Learning

Zu Beginn einer Softwareentwicklung sind die eigentlichen Nutzeranforderungen, technischen Lösungsmöglichkeiten, die Stärken und Schwächen des Entwicklungsteams u.v.m. unscharf – und all dies verändert sich kontinuierlich über die Zeit.

Der Weg zum Ziel ist ein kontinuierlicher Prozess des Ausprobierens, Überprüfens und Anpassens – alle Beteiligten lernen mit jedem Schritt dazu. Die Lösung wird so schrittweise verbessert und nähert sich immer stärker dem Zielzustand an.

Um zu überprüfen, ob der eingeschlagene Weg der richtige ist, braucht man häufige Feedbackschleifen. In der Softwareentwicklung nutzt man

- Frühe Tests statt Anhäufung von Fehlern
- Prototypen statt detaillierter Up-front-Planung und Dokumentation
- Mockups für den Nutzer statt umfangreicher Pflichtenhefte
- Tools ausprobieren und testen statt langer Auswahlprozesse
- Schrittweise Einführung statt Big-Bang-Ansatz

All dies erfolgt in kurzen, regelmäßigen, kurzen Iterationen, in denen in einem definierten Zeitrahmen ein Nutzen bringendes Softwareinkrement entworfen, codiert, getestet, integriert und ausgeliefert wird.

Iterationen verbessern die Kommunikation im Team und zum Kunden, erlauben das ständige Justieren des Lösungswegs und ermöglichen damit das Verzögern von Entscheidungen (vgl. Prinzip 3: Defer Commitment). Regelmäßige Synchronisation zwischen Team und Kunde machen Entscheidungsbedarfe regelmäßig transparent und treiben so Entscheidungen voran.

Prinzip 3: Defer Commitment

Softwareprojekte sind oft von hoher Unsicherheit geprägt: Welche Features bringen wirklichen Wert? Welche Architektur ist auch in der Zukunft belastbar? Halten die Fremdkomponenten was sie versprechen? Bleiben die Know-how-Träger an Bord?

Darum sollten wichtige Entscheidungen (z. B. Architekturentscheidungen) so lange aufgeschoben werden, bis genügend Informationen vorliegen. Alternativen sollten möglichst so lange offengehalten werden, bis man genug gelernt hat, um die beste Entscheidung zu treffen.

Generell ist eine flexible, offene und änderbare Architektur unabdingbar, um später Features problemlos hinzufügen zu können. Jedoch sollte gleichzeitig Overdesign vermieden werden (siehe *Waste*).

Prinzip 4: Deliver Fast

Das Prinzip der „schnellen Lieferung“ basiert auf der Idee, dass man aus dem Kundenfeedback umso schneller lernen kann, je schneller man dem Kunden Elemente von Wert liefert. Und je mehr man vom Kunden lernt, desto besser ist man in der Lage, dem Kunden genau das zu liefern, was er will.

Unternehmen, denen es gelingt, die Entwicklungszeiten zu verkürzen, liefern besser passende Lösungen mit höherer Qualität.

Prinzip 5: Empower the Team

Um komplexe Systeme von hoher Qualität zu schaffen, bedarf es kompetenter Entwickler, die sich für das Gesamtergebnis verantwortlich sehen. Gerade bei der Detailarbeit an Architektur und Software ist das Detailwissen der Entwickler in Kombination mit dem Wissen des gesamten Teams wertvoll für den Erfolg.

Effektive Teams sind mit dem notwendigen Wissen versehen bzw. eignen sich selbst die benötigten Fähigkeiten an, haben aber auch effektive Führungskräfte, die ihr Team bestmöglich unterstützen.

Effektive Teams verwenden Pull-Techniken zur Planung der Aufgaben. Lokale Signalisierungsmechanismen wie Kanban-Boards, tägliche Meetings, aber auch häufige Integration und umfassende Tests, machen sichtbar, was als Nächstes zu tun ist.

Engagierte, mitdenkende Team-Mitglieder, die zusammen mit anderen etwas gestalten möchten, bieten somit den nachhaltigsten Wettbewerbsvorteil. Sie schaffen ein optimales Endprodukt durch Stolz, Engagement, Vertrauen und Anerkennung.

Prinzip 6: Build Quality In

Qualität bedeutet einerseits, dass das Produkt eine hohe gefühlte Integrität *(Perceived Integrity)* besitzt, also über seine gesamte Laufzeit die Kundenwünsche erfüllt. Marktanteile sind ein grobes Maß für diese gefühlte Integrität, weil sie die Kundenwahrnehmung über die Zeit messen.

Andererseits muss Software auch eine hohe konzeptionelle Integrität *(Conceptual Integrity)* besitzen. Die zentralen Konzepte des Systems sollen als ein reibungsloses, kohärentes Ganzes zusammenwirken. Gute Software hat eine adäquate Architektur, punktet mit hoher Benutzerfreundlichkeit und Zweckmäßigkeit und ist wartbar, anpassungsfähig und erweiterbar.

Darüber hinaus hat Software eine weitere, zusätzliche Ebene der Integrität: Sie muss ihren Nutzen über den gesamten Lebenszyklus behalten. In der Regel wird von Software erwartet, dass sie sich mit Änderungen reibungslos weiterentwickeln lässt.

Maßnahmen für eine gute Softwarequalität sind u.a. automatisierte Unit- und Akzeptanz-Tests und Test-Driven Development. Continuous Integration und regelmäßige Synchronisation zwischen Entwicklungsbereichen sorgen für frühe Erkennung von Fehlern – der Big-Bang-Ansatz ist obsolet.

Prinzip 7: Optimize the Whole

Um bei komplexen Systemen eine hohe Qualität zu erhalten, braucht es Expertise in zahlreichen unterschiedlichen Bereichen. Eines der hartnäckigsten Probleme bei der Produktentwicklung ist die Tendenz von Entwicklern und Teams, sich auf Detailprobleme zu konzentrieren anstatt das Gesamtsystem zu optimieren.

Es gilt also, Verfahren zu implementieren, die überflüssige Detailoptimierung vermeiden und gleichzeitig den Blick auf das Gesamtergebnis schärfen.

Überaus wichtig ist der Fokus aller Aktivitäten auf den Business Value: Ist dieses Feature, dieser Mechanismus, diese Optimierung, diese Aktivität etc. von Wert für das Unternehmen?

Bild 1: Business Value

Für eine fortlaufende Optimierung eines Systems ist die kontinuierliche Messung der Qualität in allen Bereichen entscheidend: Durchlaufzeiten, Team-Performance (Velocity), Architekturbewertung, u.v.m., aber vor allem auch die Kundenzufriedenheit sollte durchgängig ermittelt werden und Verbesserungen erarbeitet und regelmäßig umgesetzt werden.

Agile Softwareentwicklung ist *lean*

Agile Methoden haben seit Ende der 1990er Jahre auf breiter Front Einzug in die Softwareentwicklung (und darüber hinaus) gehalten. Scrum ist die bei weitem meistgenutzte agile Methode, SAFe dominiert bei den Skalierungsmethoden ([3]). Architekturarbeit in agilen Projekten ist längst auf die agile Vorgehensweise adaptiert ([4]).

Wie passen diese agilen Methoden zu den Prinzipien des Lean Software Development?

Eliminate Waste: Agile Methoden eliminieren *Waste* durch flexibles Reagieren auf Kundenanforderungen (priorisiertes Product Backlog, kurze Iterationszyklen, …) und die Vermeidung von Prozessübergängen (Teamansatz, Fokus auf funktionierende Software statt Dokumente) und Prozesswechseln (No Change). Die Orientierung am Business Value und dem Minimum Viable Product (MVP) reduziert die Gefahr von Overdesign.

Amplify Learning: Agile Methoden erlauben ein kontinuierliches Lernen, indem Fehler früh gefunden werden (Continuous Integration, Continuous Testing), Feedback des Product Owners regelmäßig eingeholt wird und indem häufige Produktinkremente anstelle eines späten Big-Bang-Releases die Validierung im Feld ermöglichen.

Defer Commitment: Agile Methoden ermöglichen das Verzögern von Entscheidungen, weil das System inkrementell erweitert wird und nicht in großen, untrennbaren Arbeitsschritten. Mit jedem Sprint können Alternativen ausprobiert und validiert werden, um so schrittweise die beste Lösung zu finden. Der Umfang eines Releases kann spät festgelegt werden, wenn klar ist, welche Features den größten Business Value bringen.

Deliver Fast: Kern agiler Methoden ist die regelmäßige Auslieferung von Produktinkrementen, die bereits früh Nutzen (Business Value) schaffen und dem Team frühes Feedback der Nutzer bringt.

Empower the Team: Agile Methoden fördern die Eigenverantwortung des Teams. Das Team entscheidet selbst über die Aufgabenverteilung. Aufgaben übernimmt, wer die nötige Kompetenz und Zeit hat. Das Team arbeitet ungestört und unabhängig an der Fertigstellung der Aufgaben. Dies erhöht die Effizienz, verringert Abstimmungsaufwände *(Waste)*, vermeidet so Wartezeiten und verteilt und verbessert das Wissen im Team.

Build Quality In: Agile Methoden erhöhen die gefühlte Integrität, weil Product Owner und Nutzer durch regelmäßige, immer besser passendere Ergebnisse, Zutrauen in das System und die Fähigkeiten des Teams gewinnen. Gängige Maßnahmen wie Unit-Tests und Continuous Integration finden früh Fehler. Regelmäßiges Lernen und die inkrementelle Verbesserung des Gesamtsystems durch Refactoring ermöglichen die Aufrechterhaltung der konzeptionellen Integrität.

Optimize the Whole: Die Optimierung des Gesamtsystems bleibt ebenso wie die Schaffung und Beibehaltung einer kohärenten und belastbaren Architektur eine der großen Herausforderungen, auch oder gerade in agilen Organisationen. Das beim agilen Vorgehen gewollte Auf-Sicht-Fahren birgt die Gefahr, dass der Blick auf das große Ganze verloren geht. Agile Teams brauchen daher eine gute Führung („Leadership“), die stets das Gesamtziel im Fokus behält.

Es zeigt sich also, dass die agilen Verfahren per Design die Lean-Prinzipien erfüllen.

Grenzen agiler Methoden in der Produktentwicklung

Agile Verfahren sind in der klassischen IT wie auch in der Cloud-Entwicklung, wo es typischerweise nur eine Instanz eines Zielsystems gibt, erfolgreich und flächendeckend im Einsatz.

Die Erfahrung zeigt jedoch, dass agile Verfahren in der Entwicklung von Embedded Software im Produktumfeld nicht eins zu eins umgesetzt werden können:

Der **Business Value** ist zwar direkt messbar (Absatzzahlen des Produkts), der Wert eines einzelnen Features ist aber selten direkt ermittelbar. Da es keine direkten, bewertbaren Kundenanforderungen gibt, ist eine Priorisierung der Features im Product Backlog nach Business Value nur schwer leistbar und wird oft eher von aktuellen Eindrücken der Produktmanager bei den Kunden und deren Forderungen geprägt.

Auch in der Embedded Software ist ein **inkrementelles Vorgehen** innerhalb der Entwicklung möglich und sinnvoll, jedoch können Inkremente im Regelfall nicht ausgeliefert werden, da die Software einen komplexen Verifizierungs- und Freigabeprozess durchlaufen muss und erst dann auf den gefertigten Produkten installiert werden kann. Over-the-Air-Updates halten erst langsam Einzug, ermöglichen aber beispielsweise bei Nicht-Erreichbarkeit einzelner Produkte (etwa in abgeschirmten Umgebungen) keine flächendeckende Aktualisierung der Software. Damit ist Continuous Deployment nicht möglich, Software muss stets zu einem definierten Zeitpunkt geliefert und schrittweise installiert werden. Dies verhindert schnelle Änderungen und widerspricht der Idee, Features auszuprobieren und schnell dazuzulernen.

Produkte sind geprägt von **Variantenreichtum**: Unterschiedliche Ausbaustufen einer Produktlinie, spezifische Funktionen je nach Einsatzort (z.B. Land) und regulatorischem Umfeld spielen hier genauso eine Rolle wie kundenspezifische Funktionen. So wird aus einer anfangs noch beherrschbaren (Standard-)Software ein hochkomplexes, konfigurierbares Produkt mit unzähligen Varianten.

Die Produktwelt ist geprägt von der **Langlebigkeit** der Produkte und der Vielfalt der Produktgenerationen im Markt. Jegliche Änderung an Features und Software muss immer gespiegelt werden mit der installierten Basis: Auf- und Abwärtskompatibilität spielen eine wichtige Rolle. Retrofitting wird oft gewünscht. Damit fallen Änderungen an Features und vor allem das Abkündigen von Features schwer und unterbleiben oft. Der Funktionsumfang steigt immer weiter, die Komplexität wächst kontinuierlich.

Viele **Entscheidungen** haben langfristige Auswirkungen, weil Sie nur bedingt änderbar sind. Damit werden Strukturen stärker zementiert als in der klassischen IT, weil etwa Schnittstellen und Datenformate im Nachhinein nur mit hohem Aufwand angepasst werden können.

Dennoch zeigt die Erfahrung, dass agile Verfahren – richtig angepasst – auch für die Entwicklung von Embedded Software im Produkt entscheidende Vorteile bieten (siehe auch [5]).

Lean Product Software: Was geht?

Ausgehend von diesen Vorüberlegungen stellt sich die Frage, was man aus *Lean Software Development* für Embedded Software im Produkt ableiten und anwenden kann.

Zentrales Element der Lean Prinzipien ist die Vermeidung von *Waste*:

Waste bedeutet vor allem ein Reduzieren von überflüssigen Elementen in der Software selbst. Gerade durch die Langlebigkeit und schlechtere Änderbarkeit von Embedded Software (siehe voriger Abschnitt) muss darauf geachtet werden, dass die Software im Produkt nichts Überflüssiges enthält, was über lange Zeit Flexibilität und Wartbarkeit einschränkt und unnötig Aufwand und Kosten produziert.

Features richtig bewerten

Zentrale Eigenschaften von Produkten sind die Produkteigenschaften oder Features. Es gilt, immer die am meisten Nutzen bringenden Features zu implementieren und überflüssige oder unnötige Features gar nicht erst zu realisieren: Bei neuen Anforderungen sollte immer deren Business Value im oben genannten Sinne evaluiert werden, um dann durchaus auch einmal bewusst die Entscheidung zu treffen, ein Feature **nicht** zu implementieren.

Gerade im Produktgeschäft sollten Strategien entwickelt werden, wie man den Business Value eines Features ermitteln kann, etwa indem man das Wissen über die Nutzung der Produkte durch die Kunden in einer Datenbank erfasst und pflegt. Damit kann man Fragen beantworten wie „Welche Kunden profitieren von diesem Feature?“ oder „Kaufen die Kunden mehr Produkte, wenn dieses Feature enthalten ist?“.

Wie bei SaaS-Systemen üblich, sollte überlegt werden, wie man die Nutzung von Features im Feld messen kann. Besteht eine Online-Verbindung zum Produkt, können Feature-Statistiken leicht ausgelesen werden. Denkbar ist aber auch, dass Servicetechniker bei ihren Besuchen diese Daten auslesen – das Einverständnis des Kunden vorausgesetzt. So gewinnt man wertvolle Informationen über den tatsächlichen Kundennutzen eines Features.

In der heutigen Zeit wird es immer leichter möglich, Software im Feld durch den Hersteller zu verändern (Over-the-Air-Update, Update aus der Cloud, etc.). Diese Chance sollte genutzt werden, um auch im Embedded-Umfeld einerseits Features nachladbar zu machen und andererseits Logdaten über die Nutzung zu erhalten. So kann man lernen, wie und wie oft ein Feature genutzt wird, um schlussendlich den Kunden stets die richtigen Features anzubieten und unnötige Features gar nicht erst zu implementieren.

Softwareerosion vermeiden

Einfachheit ist eine der zentralen Grundanforderungen an jede gute Softwarearchitektur: Nur wenn eine übersichtliche, schnell zu verstehende Architektur existiert, die adäquat dokumentiert ist, sind neue Mitarbeiter schnell produktiv und das Wissen steckt nicht nur in den Köpfen weniger Spezialisten.

Auch in der Architektur gilt es daher, Overdesign zu vermeiden und die richtige Balance zwischen Flexibilität und Einfachheit zu finden und zu halten. Auch in der Embedded Software ist das schrittweise Evaluieren von Architekturentscheidungen

möglich, wenn auch in größeren Zyklen. Die Architektur sollte genauso regelmäßig überprüft und ggf. angepasst werden, damit sie adäquat und „einfach“ bleibt.

Regelmäßiges Refactoring gehört zu jeder guten Architektur und sollte auch in der Embedded Software genutzt werden, um stets eine wartbare und adäquate Basis für die Weiterentwicklung zu behalten. So bleiben teure und risikoreiche Reengineering- oder Modernisierungsprojekte die Ausnahme.

Größere Veränderungen sind auch in der Embedded Software denkbar: Datenmodelle können auch hier per Datenmigration verändert werden (angestoßen beim Update-Vorgang). Schnittstellen sollten stets Versionsnummern tragen, damit alte und neue Schnittstellen nebeneinander existieren können. Hier kann man Anleihen bei bewährten Vorgehensweisen in der IT nehmen.

Varianten richtig managen

Neben vielen Features und einer zu komplexen Architektur ist ein weiterer Komplexitätstreiber bei Embedded Software im Produkt die Vielzahl der Varianten: Varianten durch unterschiedliche Ausbaustufen von Produkten, Anforderungen der Zielmärkte, aber auch kundenspezifische Anforderungen sorgen für eine hohe inhärente Komplexität der Software.

Zahlreiche Unternehmen wählen den zunächst leicht erscheinenden Weg, alle Features in die zugrunde liegende Software durch Fallunterscheidungen im Code zu integrieren. Ohne ein stringentes Management der Varianten wächst jedoch die interne Komplexität der Software, Abhängigkeiten von Varianten werden nicht erkannt, die Testbarkeit wird zum Problem (Pfadabdeckung!) und die Verständlichkeit von Architektur und Code leidet in erheblichem Maße.

Es gilt, diesen überflüssigen *Waste* durch die Schaffung von Transparenz beherrschbar zu machen: Der Einsatz von Produktlinienkonzepten mit Feature-Modellen bringt zwar eine zusätzliche Verwaltungsebene, erlaubt aber eine fundierte Entscheidung über Features auf Produkt- und Architekturebene. Es kann gemessen werden, welche Features wie oft eingesetzt/gewählt werden. Vor allem kann klar dokumentiert werden, welcher Kunde welches Feature bekommen hat, um daraus später Schlüsse bzgl. Weiterentwicklung oder Abkündigung ziehen zu können.

Die klare Trennung von Architekturebenen und der entsprechenden Varianz sorgt für weitere Erhöhung der Wartbarkeit.

Kundenspezifische Features

Viele Produkthersteller arbeiten sehr eng mit ihren Kunden und erstellen so häufig kundenindividuelle Varianten ihrer Produkte. Oft werden diese kundenindividuellen Varianten gar nicht als solche identifiziert, weil man der Auffassung ist, dass ein vom Kunden gewünschtes Feature auch für andere Kunden von Nutzen sein könnte – dies wird aber selten überprüft. So wird das Kernprodukt immer weiter aufgebläht mit zusätzlichen Features und Varianten – auf Kosten von Wartbarkeit, Testaufwänden, Verständlichkeit usw.

Um diesen *Waste* zu vermeiden, sollte jede Kundenanforderungen auf Ihren Business Value überprüft werden:

(1) Ist die Anforderung bereits im Standard vorhanden (möglicherweise konfigurierbar), dann hat man in der Vergangenheit die richtige Entscheidung getroffen. Das Produkt erfüllt den Kundenwunsch bereits.

(2) Ist das Feature neu und tatsächlich von Wert für zahlreiche andere Kunden, dann sollte es entsprechend gut ausgestaltet und implementiert werden, damit es vielen Kunden von Nutzen ist. Das Standardprodukt wird erweitert mit dem Ziel, in Zukunft mehr Kundenwünsche mit dem Standard abdecken zu können, siehe (1).

(3) Wenn aber das Feature zunächst nur für einen Kunden von Interesse ist, sollte die entsprechende Implementierung separat gehalten werden, um so den Kern des Produkts nicht zu überfrachten. Das Verwalten dieser kundenindividuellen Erweiterungen ist zwar eine zusätzliche Verwaltungsaufgabe, aber mit zusätzlichen Chancen: Benötigt der Kunde das Feature nicht mehr, kann man es leichter abkündigen und abschalten.

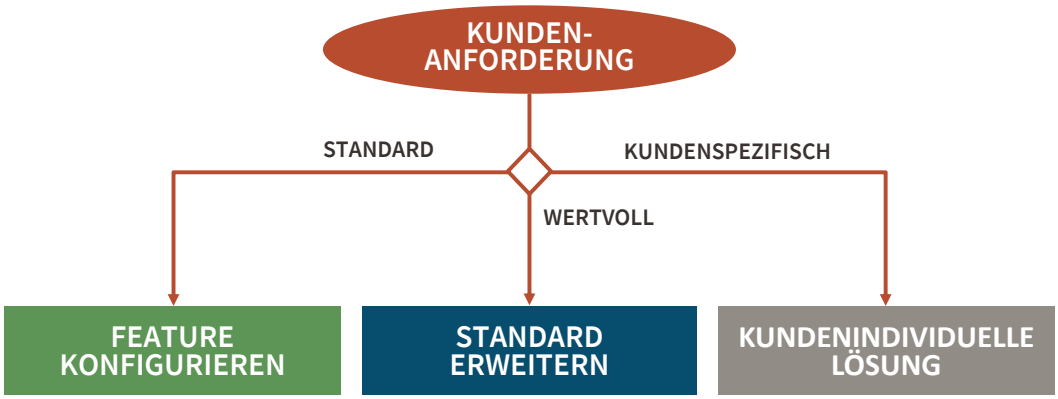

Bild 2: Umgang mit Kundenanforderungen im Produkt

Ein solches Vorgehen ist in der Softwarearchitektur üblich: Volatile Komponenten werden herausgelöst und von den stabilen Elementen getrennt. Damit wird die zugrundeliegende Software weiter stabilisiert und nicht durch kundenspezifischen Code beeinträchtigt.

Make it Simple Again!

Der Kern einer jeden guten Architektur und auch eines guten Softwaresystems ist Einfachheit. Im Sinne von Lean und dem Vermeiden von *Waste* gilt es, (a) nichts ins System einzubauen, das nicht wichtig ist, (b) verfolgen, ob die Bestandteile des Systems weiterhin von Nutzen sind und (c) die überflüssigen oder nicht mehr passenden Bestandteile zugunsten der Einfachheit wieder auszubauen.

Ein derartiges Vorgehen erlaubt es, ungenutzte oder nicht mehr genutzte Features gezielt abzukündigen und in einer Folgegeneration wieder auszubauen. Die Architektur wird regelmäßig überprüft und gewartet. So bleibt die Software schlank und wird über die Zeit von unnötig gewordenem Ballast befreit.

Zusammenfassung

Über die bewährten agilen Methoden hinaus bieten die Ansätze von Lean Software Development wichtige Impulse für die tägliche Arbeit in der Embedded Entwicklung im Produktgeschäft.

Das Eleminieren von überflüssigem Ballast *(*Waste*)* ist die zentrale Forderung von Lean: Das System sollte schlank gehalten werden, indem überflüssige Features und Overdesign vermieden werden, Variantenvielfalt durch das Transparentmachen in

Feature-Modellen beherrscht und kundenspezifische Elemente isoliert werden. Der Einsatz von agilen Methoden reduziert *Waste* im Prozess und ermöglicht eine effiziente Arbeitsweise ohne komplexe Planungs- und Steuerungsprozesse.

Auch im Produktgeschäft ist es möglich, Features auszuprobieren und durch Messen im Feld Erkenntnisse über Nutzung und Qualität zu gewinnen und nachzuregeln. Das Ziel sollte auch hier sein, stets zu lernen und so das System schrittweise zu verbessern und Fehlentscheidungen mit dem nächsten Update zu revidieren.

Nicht zuletzt ist es gerade bei Embedded Software entscheidend, die Qualität durch eine stets adäquate, schlanke Architektur hoch zu halten und durch konstantes Refactoring zu bewahren. So werden teure und risikoreiche komplexe Modernisierungsprojekte vermieden.

Verweise

[1]
Mary Poppendieck, Tom Poppendieck
Lean Software Development – An Agile Toolkit
Addison-Wesley, Boston, MA, 2003

[2]
Mary Poppendieck, Tom Poppendieck
Implementing Lean Software Development – From Concept to Cash
Addison-Wesley, Boston, MA, 2007

[3]
Ergebnisbericht Status Quo (Scaled) Agile 2019/2020
4. Internationale Studie zu Nutzen und Erfolgsfaktoren (skalierter) agiler Ansätze
Hochschule Koblenz, 2020, www.status-quo-agile.de

[4]
Stefan Toth
Vorgehensmuster für Softwarearchitektur –
Kombinierbare Praktiken in Zeiten von Agilen und Lean
Carl Hanser Verlag, München, 2015

[5]
Jens Kinzel, Dr. Jörg-Volker Müller
Erfolgsfaktoren für Scrum in der Produktentwicklung:
Praktische Tipps für erfolgreiche Embedded-Software-Projekte
Tagungsband *Embedded Software Engineering Kongress,* 2018, Seite 546

Über den Autor

Dr. Müller ist Geschäftsführer der Systemum GmbH & Co. KG. Er hat in leitender Funktion in zahlreichen Projekten erfolgreich IT-Systeme, Frameworks und Softwareprodukte realisiert. Seine fachlichen Schwerpunkte sind Software Engineering, Softwaremodernisierung, Architekturen, Plattformen, Produktlinien und Varianten in Embedded Software und IT. Dr. Müller hat Informatik in Erlangen und Braunschweig studiert und an der TU Braunschweig promoviert. Seit 2012 hält er Vorlesungen zum Thema Softwareengineering an der Hochschule Harz.