SOFTWAREPROJEKTE QUANTITATIV MANAGEN: EIN WEG ZUM PERFORMANTEN TEAM

der autor

Matthias Bohlen
(E-Mail: mbohlen@mbohlen.de)
ist unabhängiger Coach für Softwareentwicklungsteams, die hochproduktiv werden wollen. Er ist Mitglied des OBJEKTspektrum-Redaktionsteams.

In Softwareprojekte wird viel Geld investiert. Der Auftraggeber möchte, dass diese Investition Früchte trägt, zu neudeutsch: Er erwartet ein Return on Investment (ROI). Wie kann man mit Hilfe quantitativer Methoden diesen ROI tatsächlich erwirtschaften? Dieser Artikel führt zunächst grundsätzliche Messgrößen für Softwareprojekte ein. Danach wird gezeigt, wie man ein Softwareprojekt als wissensverarbeitendes System verstehen kann – dadurch wird das Managementwissen aus den Bereichen „Lean Production" und „Theory of Constraints" anwendbar. Schließlich erklärt der Artikel, wie dieses Wissen im Projektalltag angewendet wird.

Das Ziel eines Projekts

Die Frage, warum man Softwareprojekte durchführt, ist so einfach, dass sie in der Praxis oft gar nicht gestellt wird. Es ist jedoch sehr wichtig, die Antwort auf diese Frage im Projektteam zu kennen. Der Auftraggeber eines Projekts möchte in der Regel mit dem Projekt Geld verdienen. „Verdienen" hat jedoch unterschiedliche Bedeutungen, je nach der Art des Projekts.

Unterteilen wir zunächst grob in Entwicklungs- und Wartungsprojekte, danach in verkaufbare oder intern benutzte Software (siehe Tabelle 1). Woher kommt dann das mit dem Projekt verdiente Geld? Wenn man Software für externe Kunden entwickelt, zahlen diese für die Nutzung der Software eine Lizenzgebühr oder Miete. Das ist die direkteste Form von „Geld verdienen". Ist die Software einmal in Betrieb, schließt der Kunde oft einen Wartungsvertrag ab, für den er Wartungsgebühren bezahlt. Auch das ist direkt messbar.

Schwieriger wird es bei Software für den firmeninternen Bedarf. Entwicklungsprojekte werden häufig gestartet, um durch die fertige Software Einsparungen zu erzielen, indem typische Geschäftsprozesse des Unternehmens automatisiert werden oder indem Aufwand, der bisher intern entstanden ist, im „Do it yourself"-Stil auf die internet-fähigen Kunden des Unternehmens verschoben wird. Diese Einsparungen werden geschätzt und treten erst in der Zukunft ein – d. h. sie werden erst zukünftig richtig messbar. Ist die interne Software in Betrieb, zahlen Unternehmensbereiche, die die Software nutzen, oft eine Umlage, mit der die Wartung finanziert wird – auch das ist wieder messbar. In Ihrem eigenen Unternehmen wird es sicherlich noch andere Möglichkeiten geben – fragen Sie sich einmal, woher bei Ihnen das Geld für das Projekt kommt.

Wozu Messgrößen?

Unterschiedliche Projektbeteiligte können ein ganz eigenes Interesse an Messgrößen und quantitativer Steuerung haben:

- Der *IT-Vorstand (CIO)* und das *obere Management* möchten *Return on Investment (ROI)*.
- Der *Projektleiter* strebt Voraussagbarkeit und Verlässlichkeit an.
- Das *Entwicklerteam* möchte mit gleich bleibender Schlagzahl arbeiten.

Das Management möchte Informationen, mit denen es das Geschäft gestalten und Investitionsentscheidungen fällen kann. Der IT-Vorstand und seine Kollegen möchten entscheiden, wo sie das Kapital der Anteilseigner investieren, um einen akzeptablen ROI zu erhalten. Sie möchten verschiedene Investitionsmöglichkeiten vergleichen können, um den bestmöglichen ROI für das zur Verfügung stehende Kapital zu ermitteln. In welches Projekt lohnt es sich zu investieren? Welches Projekt wird einen hohen ROI bringen, welches eher nicht? Der Projektleiter wird gefragt, wann sein Team eine bestimmte Funktionalität zur Verfügung stellen kann. Er möchte, dass das Projekt einen voraussagbaren Verlauf nimmt, indem Ergebnisse regelmäßig und verlässlich eintreffen. Das Entwicklerteam möchte mit gleich bleibender Schlagzahl arbeiten können; plötzliche Schwankungen und Stressphasen möchte es vermeiden.

Die Messgrößen, mit denen das Projekt gesteuert wird, müssen allen drei Interessengruppen nützen. Sie müssen einfach sein und für das zu erreichende Ziel (hier: Geld zu verdienen) eine Relevanz haben. Reinertsen verlangt: Messgrößen sollten idealerweise selbst-erzeugend sein, und: sie sollten die Leistung des Systems voraussagen[1]) können anstatt ihr „hinterherzuhinken" (vgl. [Rei97]).

	Entwicklung	Wartung
Software für Verkauf oder Vermietung	• Lizenzgebühren • Mieteinnahmen	Wartungsgebühren
Software für firmeninternen Bedarf	• Ermöglichen neuer Geschäftsmodelle • Einsparungen durch Automatisierung	Umlage aus anderen Unternehmensbereichen

Tabelle 1: Woher kommt das mit dem Projekt verdiente Geld?

T	Durchsatz (*Throughput*)	Geld, das die Fabrik durch Verkäufe generiert.
I	Inventar (*Inventory*)	Geld, das die Fabrik investiert hat, um Dinge zu kaufen, die sie verarbeiten und verkaufen will.
oE	Operative Ausgaben (*operative Expenses*)	Geld, das die Fabrik ausgibt, um aus Inventar Durchsatz zu machen.

Tabelle 2: Wichtige Messgrößen für eine Fabrik.

Messen am Beispiel einer Fabrik

Stellen wir uns ein Projekt einmal als Fabrik vor (in welchem Sinne es wirklich eine Fabrik ist, klären wir im nächsten Abschnitt). Die Fabrik kauft Rohmaterial, verarbeitet dieses zu interessanten Produkten, die sie am Markt verkauft. Solch eine Fabrik kann man mit drei Messgrößen beobachten:

- Durchsatz
- Inventar
- Operative Ausgaben

In Tabelle 2 sind diese drei Messgrößen einheitlich monetär definiert.

Anders gesagt: Durchsatz ist Geld, das in die Fabrik *hereinkommt*. Inventar ist Geld, das in der Fabrik *verbleibt*. Operative Ausgaben sind Geld, das die Fabrik *verlässt* (siehe Abb. 1).

Aus diesen drei Messgrößen lässt sich für die Fabrik ein ROI ableiten: Das ist ein Faktor, der beschreibt, wie viel mehr die Fabrik verdient, als sie investiert:

$$ROI = (T - oE) / I$$

Am Anfang des Projekts sind Inventar und Investment gleich – das *I* im Nenner steht für beide Größen. Aus der Formel kann man erkennen, dass der ROI wächst, wenn man den Durchsatz steigert, die Kosten senkt und das Inventar klein hält.

Inventar in der Wissensfabrik

Das wäre alles sehr schön, wenn ein Softwareprojekt eine Fabrik wäre, denn dann wüssten wir genau, wie wir es managen.

Alistair Cockburn hat in [Coc08] gezeigt, dass die Fabrik-Metapher stimmt, wenn man als Rohstoff noch nicht validierte Entscheidungen betrachtet. Dieser Rohstoff wird durch Verarbeitungsschritte (Analyse –> Design –> Implementierung –> Test –> Produktion) in weitere Entscheidungen umgewandelt und zum Schluss durch den Verkauf validiert.

Das klingt jetzt sehr abstrakt, machen wir es konkreter: Das Rohmaterial für das Projekt sind Ideen und Businesspläne, also noch nicht validierte Entscheidungen. Daraus entstehen Anforderungen, Entwürfe, Architektur, Code usw. (siehe Kasten 1). Jedes dieser Artefakte ist wieder eine nicht validierte Entscheidung, solange bis die Benutzer das neue System akzeptieren und es im Betrieb zeigt, dass die Entscheidungen valide waren.

Inventar kann man auf verschiedene Art ausdrücken – je nachdem, welcher Lebenszyklus-Methodik man folgt. Generell be-

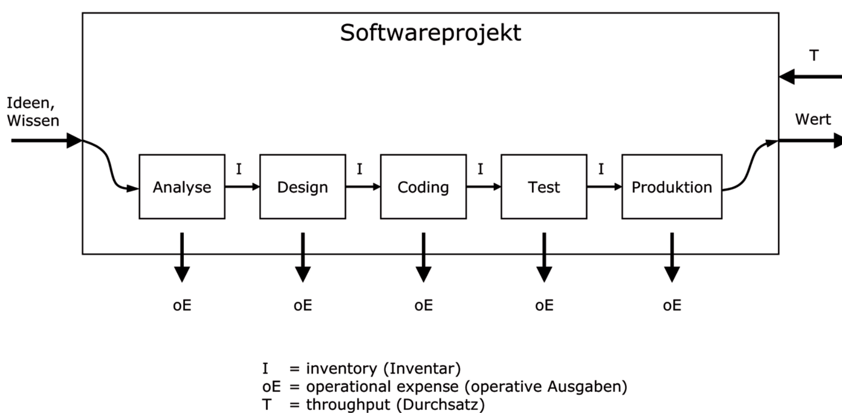

Abb. 1: Messgrößen eines Projekts.

Manche Leser werden sich fragen, warum ein Stück Code eine „nicht validierte Entscheidung" ist, bevor das System in Betrieb geht? Durch Unit-Tests zur Entwicklungszeit kann ich doch schon früher sehen, dass eine Benutzeranforderung erfüllt ist. Die Antwort lautet: Im Hinblick auf das Ziel des Projekts („Geld verdienen") sind alle noch nicht verkaufbaren Zwischenergebnisse nicht validierte Entscheidungen, denn man weiß noch nicht, ob sie Geld bringen werden. Auch Code ist eine solche Entscheidung, solange, bis er in Produktion ist und verkauft wurde (z. B. als Lizenzgebühr oder als Einsparung). Das muss man sich klar machen: Viel Code schreiben und selten in Produktion gehen, ist Geldverschwendung, weil man damit Inventar aufbaut, das Abschreibungskosten verursacht.

Der Unit-Test hilft dabei überhaupt nicht, denn er verifiziert nur, er validiert nicht: Er verifiziert, dass die zu testende Unit sich so verhält, wie ihre Spezifikation es angibt, und dass die Unit die Spezifikation auch heute noch so gut einhält wie gestern. Leider ist die Spezifikation selbst auch wieder eine noch nicht validierte Entscheidung.

Kasten 1: Warum ist Code eine nicht validierte Entscheidung?

[1]) *Bei Softwareentwicklungsprojekten handelt es sich um hoch vernetzte, rückgekoppelte, wissensverarbeitende Systeme, die gerade noch geordnet arbeiten. Die Chaostheoretiker würden sagen: „Das System arbeitet am Rande zum Chaos", weil sich durch die im Projekt vorhandenen Rückkopplungsschleifen nur schwer voraussagen lässt, wie es auf kleine Änderungen der Eingaben (sprich: Anforderungen, Ereignisse und Rahmenbedingungen) reagieren wird. Unter Rückkopplungsschleifen verstehen wir das Wiedereinspeisen von im System entstehender Information in den Prozess. Zum Beispiel werden Fehler aus dem Unit-Test in die Codierung, Fehler im Systemtest in Architektur oder Design und Fehler im Akzeptanztest in die Anforderungsanalyse wieder eingespeist. Man versucht also, ein nahezu chaotisches System mittels Messgrößen zu beobachten und sein Verhalten vorauszusagen.*

steht eine Einheit Inventar aus einer Idee für eine Funktion, die der Kunde wertschätzt, beschrieben in einem Format, das Softwareentwickler verstehen und benutzen können. Im *Unified Process* (vgl. [Kru03]) ist eine Einheit Inventar der Use-Case. Im *eXtreme Programming (XP)* ist sie ein *Story-Point*. Im *Feature-Driven-Development (FDD)* ist sie ein Feature aus der Feature-Liste. In traditionellen *SDLC*-Methoden (*Software Development Life Cycle*) ist der Funktionspunkt die Einheit des Inventars.

Ideen sind Eingaben für das System „Softwareprojekt" – je mehr Ideen man hineingibt, desto größer ist das Investment. Das Investment wandert als Inventar durch das System, bis es zu Durchsatz wird. Der Pegel an Inventar innerhalb des Systems kann die Stelle anzeigen, an der sich das Investment im Moment befindet. Er zeigt auch, wie weit es schon darin vorangekommen ist, Durchsatz zu werden.

Wir können also noch nicht validierte Entscheidungen (bzw. vom Kunden wertgeschätzte Ideen für Funktionen des Produkts) als *Inventory (I)* managen. Wie sieht es nun mit den beiden anderen Größen *operative Expenses (oE)* und *Throughput (T)* aus?

Operative Ausgaben

oE erscheint zunächst einfach: Es handelt sich um die operativen Ausgaben für Mitarbeiter, Miete, Strom, Wasser, Papier usw. Hinzu kommen jedoch die Abschreibungskosten auf Inventar, über die in der Softwareentwicklung selten nachgedacht wird.

Hier ein Beispiel: Man schreibt ein Anforderungsdokument und investiert dafür 60.000 Euro – dann ist das Dokument ein Inventar im Wert von 60.000 Euro. Das Wissen über die Inhalte und Zusammenhänge des Dokuments beginnt zu verfallen, weil Projektbeteiligte Menschen sind, die im Laufe der Zeit Dinge vergessen. Es dauert etwa drei Monate, bis keiner mehr weiß, was mit dem Dokument gemeint war. Danach hat das Dokument also den Wert 0 Euro. Wir kommen bei 60.000 Euro Wertverlust in drei Monaten auf eine Abschreibung von 20.000 Euro pro Monat. Demnach trägt dieses Dokument *ab Fertigstellung* mit knapp 1.000 Euro pro Arbeitstag zu den operativen Ausgaben des Projekts bei. Es wäre also gut, wenn wir das Dokument vor Ablauf des Mindesthaltbarkeitsdatums in lauffähigen Code verwandeln. Das ist bei großen Dokumenten schwierig.

Als Alternative dazu nehmen wir ein agiles Vorgehen. Man investiert in die Anforderungen erst einmal nur 10.000 Euro. Danach entstehen Kosten von 10.000 Euro für Code und 20.000 Euro für Tests. Schließlich bringt man das Ganze für 20.000 Euro in Produktion. Die Kunden zahlen für die neuen Features 75.000 Euro Lizenzgebühren. Damit erhält man:

- *T = 75.000 EUR*
- *oE = 10.000 + 20.000 + 20.000 EUR = 50.000 EUR* für Code, Tests und Produktion und *10.000 EUR* für Abschreibung auf Anforderungen, also insgesamt *60.000 EUR*.
- *I = 10.000 EUR* für Investitionen in Anforderungen

Setzen wir das in die Formel für den ROI ein:

- *ROI = (T - oE) / I*

Dann bekommen wir einen ROI von:

- *(75 - 60) / 10 = 150%*

Ist das gut genug oder kann man noch optimieren? Entscheiden Sie!

Durchsatz

T ist von den dreien die am schwierigsten zu messende Größe. Um *T* zu messen, müssen wir den Wert eines Features *F* der Software beurteilen. Dabei kann uns die Unterteilung in Projekttypen helfen.

Fangen wir mit den Entwicklungsprojekten für externe Kunden an. Der Vertrieb für diese Software könnte zum Beispiel durch regelmäßige Benutzertreffen wissen, wie viele Benutzer bereit sind, Lizenzgebühren für ein Upgrade der Software zu bezahlen – vorausgesetzt, sie bekommen das neue Feature *F*. Daraus kann der Vertrieb einen Wert für *F* ableiten.

Software für die betriebsinterne Verwendung – also nicht für externe Kunden – kann man nicht verkaufen. Hier muss man

Eine Durchsatzermittlung erfordert es, den Wert eines zukünftigen Features zu bestimmen. Dazu muss man fast schon eine *NPV*-Betrachtung (*Net Present Value*) anstellen. Das bedeutet: Was wäre es heute wert, wenn wir dieses Feature in drei Monaten hätten? NPV-Betrachtungen sind sehr schwierig – deshalb tendieren alle Projekte, die ich bisher gesehen habe, dazu, nur eine relative Bewertung von Features vorzunehmen, wie zum Beispiel:

- Welches Feature ist eine Ware (*Commodity*), weil es alle Konkurrenten auch haben?
- Welches Feature ist ein Alleinstellungsmerkmal (*Differentiator*), d. h. eines, das sonst keiner hat, sodass neue Kunden begeistert zu uns kommen?
- Welches Feature ist ein Störer (*Spoiler*), weil es besser ist als das, was die Konkurrenz hat, und diese damit aus dem Geschäft drängt?
- Welches Feature ist lediglich *nice to have*?

Jetzt kann man sagen: „Eine Ware muss ich haben, um überhaupt Kunden zu bekommen, sonst bin ich sofort aus dem Geschäft". Also ist der Wert eines Waren-Features sehr hoch. „*Differentiators* muss ich haben, damit ich neue Kunden bekomme, sonst bin ich langfristig aus dem Geschäft. *Spoiler* sind weniger oder mehr wert als *Differentiators* – je nachdem, wie wahrscheinlich es ist, dass ich meiner Konkurrenz damit Kunden abjage. Mit *Spoilern* versuche ich lediglich, im Geschäft zu bleiben und *Nice-to-haves*-Features sind schließlich am wenigsten wert."

Wie rechnet man das in Euro um? Eine Möglichkeit besteht darin, den total erwarteten Durchsatz *T* eines Release zu nehmen und durch die gewichtete Feature-Zahl zu teilen: Beispielsweise zählt man eine Ware als 4, ein Alleinstellungsmerkmal als 3, einen Störer als 2 und ein *Nice to have* als 1. Dann summiert man die Punkte der Feature-Liste für das Release und kommt z. B. auf 100 Punkte. Es könnte sein, dass der Produktmanager sagt: „Das neue Release bringt uns voraussichtlich Lizenzgebühren von 20 Mio. Euro." Also ist jeder Punkt 20 Mio./100 = 200.000 Euro wert. Eine Ware bringt demnach 800.000 Euro, ein Alleinstellungsmerkmal 600.000 Euro usw.

Kasten 2: *Wertermittlung für ein Feature.*

mit den Einsparungen durch ein neues Feature *F* argumentieren:

- Um wie viel näher würde man dem Einsparungsziel des Projekts kommen, wenn die Software das neue Feature *F* enthielte?
- Wie hoch ist deshalb der Wert des Features *F*?

Wartungsprojekte werden meist durch pauschale Umlagen oder durch Wartungsgebühren finanziert, die nichts mit der Zahl der Features (also Fehlerkorrekturen, kleine Verbesserungen usw.) zu tun haben. Hier ist der Wert eines einzelnen Features schwer zu messen. Denkbar wäre eine Messung der Benutzerzufriedenheit bei gut durchgeführter Wartung oder ein Ansatz über Abschreibung: Eine gut gewartete Software hat einen längeren Lebenszyklus, d. h. die Abschreibungskosten für sie sind geringer. Eine dritte Möglichkeit wäre es, die Kosten für die letzten zwei oder drei Release-Wechsel der Software zu nehmen und zu überlegen, wie viel man davon einsparen kann, wenn in der Wartung ein bestimmtes Feature *F* geliefert würde. Eine weitere Möglichkeit finden Sie in **Kasten 2**.

ROI in unserer Zunft heute

Die meisten Unternehmen messen nur die Größe *oE*, also die operativen Ausgaben, weil sie am einfachsten herauszufinden sind. Das Inventar *I* wird in der Regel gar nicht betrachtet (oder wissen Sie, was Ihr letzter, mühsam erstellter Projektplan gekostet hat und wie lange er gültig war, bis Sie das Investment in seine Erstellung abschreiben konnten?). Der Durchsatz *T* wird in den meisten internen Projekten nicht betrachtet; in der Produktentwicklung wird meist erfasst, wie viele Lizenzen man verkauft oder wie viel Wartungsgebühr man einnimmt. Nur selten wird ermittelt, welche Features dafür verantwortlich waren. Hier gibt es noch viel Verbesserungspotenzial.

In vielen heutigen Unternehmen heißt es noch:

ROI = (Unbekannt(T) – schwer zu schätzen(oE)) / nicht gemessen (I)

Machen Sie es in Ihrem Projekt besser!

ROI morgen

Ersten sollten Sie den ROI über das gesamte Projekt hin kontinuierlich verfolgen, also über alle drei Variablen Buch führen. Zweitens sollten Sie danach streben, dass das Projekt den Durchsatz kontinuierlich und nicht sprungweise erzeugt, um sich den Auftraggebern gegenüber als voraussagbar und zuverlässig zu zeigen. Dazu ist es sinnvoll, mit deutlich weniger als 100 % der Anforderungen in einen Durchlauf durch den Entwicklungs-Workflow (Analyse, Design, Implementierung, Test, Produktion) zu gehen. Andernfalls würde der gesamte Wert, den das Projekt erzeugt, erst am Ende entstehen. Das wäre eine sehr risikoreiche Vorgehensweise und – nebenbei gesagt – kaum erfolgreich zu managen.

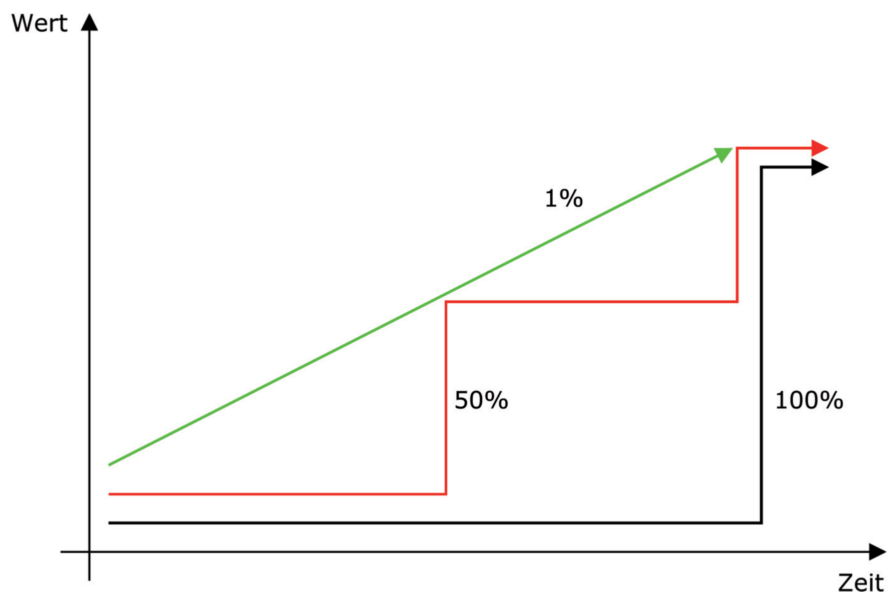

***Abb. 2:** Wertschöpfung über die Zeit, anhand verschiedener Losgrößen.*

Die etwas sehr pointierte **Abbildung 2** zeigt den Verlauf der Wertschöpfung, je nachdem, ob wir mit 100 % (schwarz), mit 50 % (rot) oder mit 1 % (grün) der Anforderungen durch den Workflow gehen. Es scheint, als sei es günstig, mit möglichst kleiner Losgröße durch den Workflow zu gehen. Wie in der Fertigung kommt es auch hier in der Softwareentwicklung darauf an, die richtige *Losgröße* zu wählen: Das ist die Größe des Stapels, mit der man auf einmal durch die Stufen des Prozesses geht. So einfach ist es jedoch nicht. Betrachtet man die Kosten pro entwickelte Einheit (Feature, Komponente, Klasse – je nach Workflow-Schritt), ergibt sich ein anderes Bild. Man könnte mit einer einzelnen Anforderung einmal durch den Workflow gehen, doch vielleicht steht die nächste Anforderung ganz konträr dazu, und man muss die entwickelte Architektur völlig verändern (die Rüstkosten für diese Anforderung wären immens hoch). Also ist es für die *Rüstkosten* günstig, mit *mehr* Anforderungen durch den Workflow zu gehen.

Mit steigender Losgröße steigt allerdings auch die Komplexität der Zwischenergebnisse (Inventar), deren Zusammenhang in den Köpfen der Beteiligten sukzessive verfällt. Mit der Losgröße steigen, wenn man so will, die Lagerkosten für Zwischenergebnisse. Für die *Lagerkosten* ist es günstig, mit *weniger Anforderungen* durch den Workflow zu gehen: Je weniger man lagert, desto geringer sind die Lagerkosten. Hier liegt also ein Widerspruch vor – was tun? **Abbildung 3** zeigt den Verlauf der Rüstkosten (rote Kurve, exponentiell fallend mit der Losgröße) und den Verlauf der Lagerkosten (blaue Kurve, linear steigend mit der Losgröße) sowie die Summe beider Kosten, die offenbar irgendwo ein Minimum hat. Das Minimum liegt – nach den Erfahrungen der Agilen Community mit Projekten der letzten 15 Jahre – bei einer Anforderungsmenge, die das Entwicklungsteam in zwei bis drei Wochen umsetzen kann, nicht mehr. Folglich sollten Sie die Losgröße in diesem Bereich wählen.

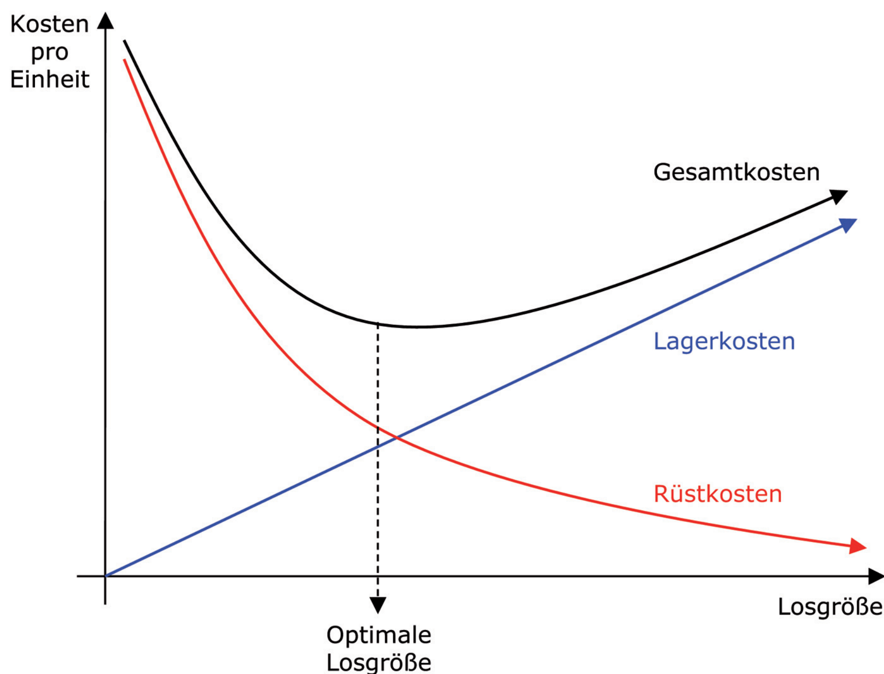

Abb. 3: Kosten pro Einheit in Abhängigkeit von der Losgröße.

Engpässe und Durchsatz-Buchführung

Auf den Durchsatz *T* zu achten, hat sicherlich hohe Priorität. Die Kosten *oE* muss man jedoch auch im Blick behalten. Je nach Art der Buchführung (Durchsatz- oder Kosten-Buchführung) verhalten sich die Kosten unterschiedlich.

Die Kosten-Buchführung (*Cost Accounting*) misst die *Kosten, die entstehen*, während jemand oder etwas arbeitet. Da die Kosten im Wesentlichen aus dem Lohn für die Projektmitarbeiter bestehen, kann man sie pro Zeiteinheit als fix annehmen. Das ist die traditionelle Betrachtungsweise.

Die Durchsatz-Buchführung (*Throughput Accounting*) betrachtet zusätzlich den *Durchsatz, der ausfällt*, wenn jemand oder etwas nicht arbeitet. Wenn eine besondere Kraft im Projekt (eine so genannte Engpass-Kraft) nicht arbeiten kann, weil ihr Voraussetzungen fehlen oder weil sie gerade mit Dingen beschäftigt ist, die nicht zu ihren Kernaufgaben gehören, fällt Durchsatz aus. Daher sieht man in der Durchsatz-Buchführung die Kosten für diese Kraft als höher an als die für eine Nicht-Engpass-Kraft.

Definieren wir zunächst, was ein Engpass eigentlich ist. Es gibt dazu zwei sich ergänzende Definitionen:

1. Engpass = Ressource (Mensch, Maschine oder Verfahren), deren Durchsatz kleiner ist als die Nachfrage des Marktes.
2. Engpass = Ressource, deren Durchsatz den Durchsatz des Gesamtprojekts wesentlich bestimmt, z. B., weil alle Ergebnisse durch diese Ressource hindurch müssen.

Ein Engpass ist aber nichts Negatives. Denn ein Unternehmen, das keinen Engpass hat, wird in Kürze ein finanzielles Problem bekommen, weil bei ihm die Nachfrage des Marktes kleiner ist als die Kapazität jeder (!) Ressource (siehe erste Definition).

Aus diesen Definitionen lässt sich folgern: Die Kosten für eine Engpass-Ressource müssen ganz anders berechnet werden als die für eine Nicht-Engpass-Ressource. Hier ein Beispiel: Nehmen wir an, das Testteam sei der Engpass. Aus Sicht der Kosten-Buchführung kostet dieses Team so viel wie die Gehälter seiner Mitglieder pro Stunde. Aus Sicht der Durchsatz-Buchführung kostet es jedoch möglicherweise hundert Mal so viel, denn wenn es nur eine Stunde lang ausfällt, kostet es so viel wie der Durchsatz des gesamten Projekts in dieser Stunde – gemessen anhand der verzögert verkauften Softwarelizenzen. Bei dieser Art der Buchführung steht also der Durchsatz im Vordergrund und nicht mehr die operativen Ausgaben. (Das Beispiel liefert übrigens eine völlig neue Sichtweise auf die Kosten für Personalfluktuation.)

Apropos Buchführungsregeln: In der heute üblichen Kosten-Buchführung schreibt man Inventar auf die Haben-Seite. In der Durchsatz-Buchführung schreibt man es auf die Soll-Seite. Warum? Inventar hindert uns daran, Durchsatz zu erzielen, weil es schwerfällig reagiert. Inventar erzeugt Kosten durch Verfall, wie bei einem leicht verderblichen Lebensmittel. So ist z. B. ein großes, aufwändig begutachtetes und verabschiedetes Anforderungsdokument eben *kein* Haben, sondern ein Lagerbestand, der Abschreibungskosten (Soll) verursacht und in Kürze nichts mehr wert sein wird. Dasselbe gilt für großes Design oder großen Code, der noch nicht in Produktion ist – das alles ist noch kein Durchsatz, sondern Inventar.[2]). Das Entscheidende am Inventar ist, dass es sich mit steigender Größe nur langsam in Durchsatz verwandeln lässt. Versuchen Sie also, nur so viel Inventar aufzubauen, wie Ihr Projekt in absehbarer Zeit in produktionsfähigen Code umwandeln kann.

Alle drei Variablen optimieren

Ein Softwareprojekt ist ein Gesamtsystem, bei dem sich die lokale Optimierung einer einzigen Variablen nicht lohnt, weil sonst die anderen Variablen aus dem Ruder laufen und die Optimierung zunichte machen. Hier als Beispiel ein traditionelles Offshore-Vorgehen: Dieses optimiert lediglich *oE*, weil die Kosten pro Mitarbeiter gesenkt werden, indem man die Arbeit in ein Land mit günstigen Löhnen verlagert. Der Durchsatz *T* wird dabei häufig aufgrund von Kommunikationsproblemen klein. Traditionell versucht man, diese Kommunikationsprobleme mit großem Inventar zu bekämpfen: Offshore-Projekte arbeiten mit großen Anforderungsdokumenten und Spezifikationen, die vor dem Projekt entwickelt und mit großen

[2]) *Das sage ich vor allem, um klar zu machen, dass ich nicht gegen Dokumente im Besonderen, sondern gegen Inventar im Allgemeinen argumentiere.*

Vertragswerken abgesichert werden, damit auch ganz bestimmt genau nach Spezifikation gearbeitet wird. Damit ergibt sich ein schlechter ROI: *oE* wird klein, *T* jedoch auch, und das große *I* tut sein Übriges.

Strategie der stetigen Verbesserung

Eine einseitige Optimierung bringt also nicht den gewünschten Erfolg. Achten Sie daher darauf, alle drei Variablen systematisch und ganzheitlich zu optimieren. Dafür hat sich die folgende Vorgehensweise bewährt:

1. Machen Sie Ihr Team arbeitsfähig. Das bedeutet, dieses muss Software von hoher Qualität entwickeln und freigeben können, denn Qualität minimiert die Nacharbeit und das wiederum senkt die Kostenvariable *oE*. Tun Sie nichts anderes, ehe dieser Zustand nicht erreicht ist. Nehmen Sie sich wenigstens drei Monate Zeit dafür. Wenn Ihr Team das Entwickeln beherrscht, seine Konfigurationen sauber managt und auftretende Fehler in kurzer Zeit behebt, dann gehen Sie zum nächsten Schritt.
2. Achten Sie auf Voraussagbarkeit (vgl. [Dem00]) und verbessern Sie diese kontinuierlich. Ihr Team sollte in der Lage sein, Ergebnisse innerhalb einer voraussagbaren Zeit zu produzieren. Beispielsweise könnte es Anforderungen nach Größe schätzen, z. B. mit Hilfe von T-Shirt-Größen (S, M, L, XL). Für jede Kategorie messen Sie nach, wie lange die Realisierung einer solchen Anforderung dauert (z. B. S = 1 Tag, M = 3 Tage, L = 9 Tage, XL = 27 Tage) und wie groß die Standardabweichung vom Mittelwert dieser Messungen ist (z. B. M = 2-5 Tage, L = 7-12 Tage). Versuchen Sie, die Anforderungen in kleine Stücke zu zerlegen – das verringert die Standardabweichung und fördert die Voraussagbarkeit. Wenn die Standardabweichung der Durchlaufzeiten klein wird, haben Sie einen voraussagbaren Prozess etabliert. Rechnen Sie auch dafür einige Monate Zeit ein und nehmen Sie Schritt 3 gleich mit zu Hilfe.
3. Verringern Sie das Inventar *I* systematisch. Versuchen Sie, Ihre Leute an möglichst wenigen Aufgaben gleichzeitig arbeiten zu lassen. Begrenzen Sie den *Work in Progress* rigoros per Order von oben: Jeder sollte nur an einer Sache arbeiten – es sein denn, diese Sache ist blockiert, z. B. wegen ausstehender Antworten, oder fehlender Ressourcen. In diesem Fall darf ein Mitarbeiter sich ausnahmsweise einer zweiten Aufgabe widmen. Das Inventar an gleichzeitiger Arbeit auf dem Schreibtisch und im Kopf sollte minimiert werden. Lassen Sie Ihre Mitarbeiter das *Pull-Prinzip* anwenden (das bedeutet, ich bekomme keine Arbeit zugewiesen, sondern hole ich sie mir) oder etablieren Sie gleich ein Kanban-System (vgl. [And10]). Das Inventar bzw. den *Work in Progress* zu begrenzen, geht relativ schnell: Es dauert zwischen einigen Tagen und einem Monat.
4. Ihr Team ist jetzt leistungsfähig und arbeitet voraussagbar? Dann versuchen Sie jetzt, den Engpass Ihres Teams zu finden. Wenden Sie die *Theory of Constraints (ToC)* (vgl. [Gol90], [Gol04]) auf diesen Engpass an: Ordnen Sie ihm zunächst alles andere unter, beschützen Sie ihn und nutzen Sie ihn sinnvoll aus. Versuchen Sie dann, den Engpass aufzulösen, z. B. indem Sie ihm unnötige Arbeit abnehmen oder mehr gleichartige Kapazität hinzufügen. Damit steigern Sie den Durchsatz *T*.
5. Wenn Sie einen Engpass aufgelöst haben, wird sich der nächste zeigen. Versuchen Sie, diesen ebenfalls aufzulösen – achten Sie dabei aber darauf, alte Regeln, die Sie zum Auflösen von Engpässen vereinbaren, regelmäßig auf ihre Notwendigkeit zu überprüfen und alte Regeln auch wieder abzuschaffen. Sonst wird Ihr Unternehmen vor lauter Regeln träge oder baut neue Engpässe auf.
6. Gehen Sie nun wieder zu Schritt 1.

Dieser Verbesserungsprozess kann (und sollte) Sie und Ihr Unternehmen ein Leben lang beschäftigen.

Fazit

In dem Artikel habe ich gezeigt, dass man die Messgrößen, mit denen man ein Projekt führt, am Ziel eines Projekts ausrichten sollte: Das Projekt muss Geld verdienen. Messgrößen müssen einfach, selbsterzeugend und dazu geeignet sein, Voraussagen zu treffen. Wir haben gesehen, dass man das Ziel „Geld verdienen" mit drei einfachen Messgrößen überprüfen kann: Durchsatz, Kosten und Inventar.

Ein Projekt, das seine Buchführung mit einem Schwerpunkt auf Durchsatz anstelle von Kosten macht, wird das Ziel viel leichter erreichen und dabei noch mehr Freude haben als andere.

Schließlich habe ich in dem Artikel eine mögliche Strategie aufgezeigt, wie eine stetige Verbesserung der Organisation erreicht werden kann: Kosten durch hohe Qualität senken, Voraussagbarkeit erreichen, Durchsatz steigern, Inventar verringern und Engpässe aufheben. Ich wünsche Ihnen allzeit viel Erfolg in Ihren Projekten! ■

Literatur & Links

[And03] D.J. Anderson, Agile Management for Software Development: Applying the Theory of Constraints for Business Results, Prentice Hall, 2003

[And10] D.J. Anderson: Kanban – Successful Evolutionary Change for Your Technology Business, Blue Hole Press, 2010

[Coc08] A. Cockburn, Effektive Softwareentwicklung im 21. Jahrhundert: Das neue Gesicht des Software-Engineering, in: OBJEKTspektrum 06/2008

[Dem00] W.E. Deming, The New Economics for Industry, Government, Education, MIT Press, Oktober 2000

[Gol04] E.M. Goldratt, J. Cox, D. Whitford, The Goal: A Process of Ongoing Improvement, 3rd revised Edition, North River Press Inc., 2004

[Gol90] E.M. Goldratt, What is this thing called Theory of Constraints and how should it be implemented?, North River Press Inc., 1990

[Kru03] P. Kruchten, The Rational Unified Process. An Introduction, Addison-Wesley Longman, 2003

[Rei97] D. Reinertsen, Managing the Design Factory: A Product Developers Tool Kit, Free Press, 1997