Aufwandstreiber und Kostenbewertung im zeitgemäßen Software Engineering

Den "oh shit"-Berg schon frühzeitig vorhersehen können

Florian Schäffer, Andreas Lachenschmidt;
iNTENCE automotive electronics GmbH

Dieser Vortrag zeigt, welche Fallstricke zeitgemäßes Software Engineering für die Aufwands- und Kostenbewertung bereithält und welche Mühen in virtuellen und verteilten Projekten versteckt sind. In der Vergangenheit konnte man Hardware-Stückkosten addieren und auf einen transparenten Herstellungsprozess zurückgreifen, bei dem die Programmierung eine Nebentätigkeit war. Bei immer umfangreicherer, komplexerer und agil entwickelter Software muss für eine zutreffende Aufwandsabschätzung schon tiefer in die Trickkiste gegriffen werden.

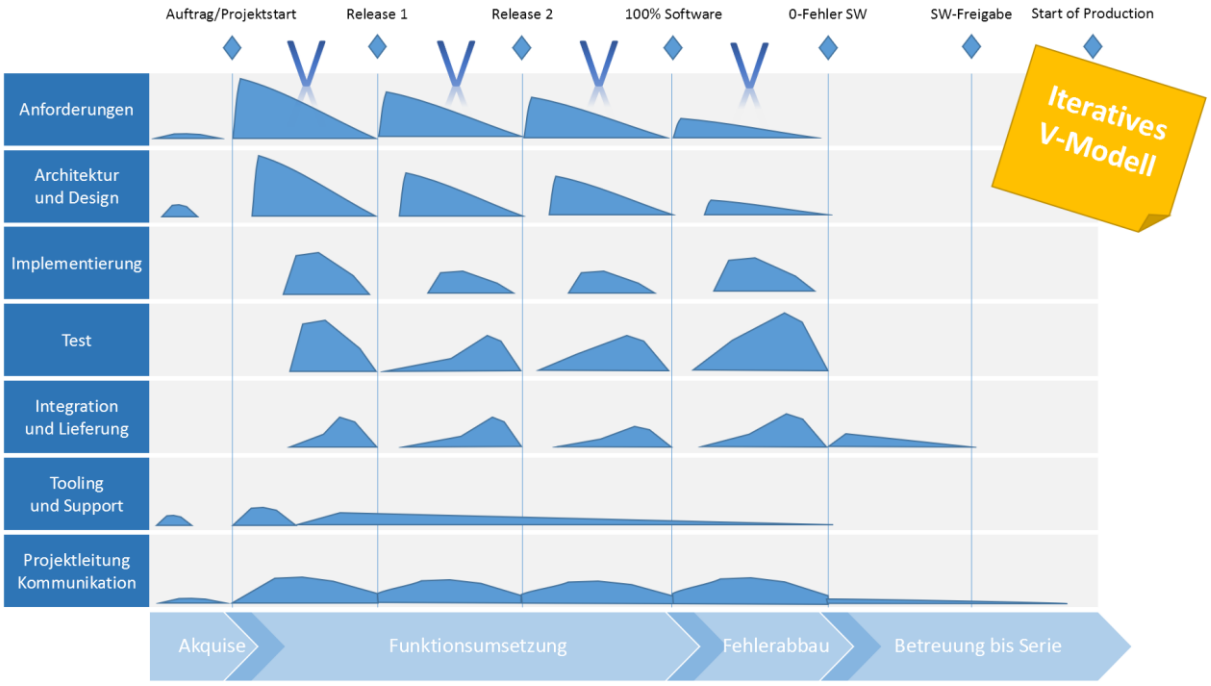
Wenn wir von Software Engineering sprechen, dann denken wir nicht an den agilen App-Entwickler, der im Alleingang alle Disziplinen ausfüllt. Seine Anforderungen formuliert er selber. Die Softwarearchitektur der App entsteht nebenher beim Finden passender Lösungen der Probleme, die auf dem Weg zum Ziel auftauchen. Die meiste Zeit der Entwicklung wird mit der Implementierung verbracht.

Aber schon das ist zu kurz gedacht: Ohne ein lauffähiges, betriebsbewährtes OS ist die beste App nicht lauffähig. Um das OS dann auch serienreif unter die Kunden zu bringen brauchen wir Software Engineering mit vielen beteiligten Spezialisten, die auch den Blick über den Disziplinen-Tellerrand nicht scheuen.

Es ist also eine Vielzahl Menschen in unterschiedlichen Rollen dafür verantwortlich, dass unser Software-Projekt den vereinbarten Zeit- und Kostenplan einhalten kann.
Ein Plan gilt aber immer nur bis zum ersten Feindkontakt. Nach Auslieferung der ersten Funktionen an den Kunden stellt sich heraus, dass die ein oder andere Anforderung falsch verstanden oder fehlerhaft umgesetzt wurde.

Wenn wir den Zeitplan nicht gefährden wollen, dann werden wir jetzt mit Überstunden und zusätzlichen Entwicklern versuchen, die Kuh wieder vom Eis zu bekommen!
Natürlich sind Mehraufwände entstanden, weil wir den "neuen Leuten", die aus anderen Projekten hinzugezogen wurden erstmal das Projekt erklären müssen. Bis die zusätzliche Hardware und alle Tools für die Entwickler eingetroffen sind vergehen ineffiziente und unproduktive Tage oder Wochen. Das einzige, was unter Volllast läuft wird die Kaffee-Maschine sein.

Die Aufwandsgebirge wachsen aber nicht nur nach oben, sondern wandern in Richtung Produktionsstart weiter. Bleiben wir zunächst bei der heilen Welt der Planung.
Beim allseits bekannten iterativen V-Modell ist die Lage recht eindeutig und auch relativ vorhersehbar, weil es das am meisten eingesetzte und damit betriebsbewährteste Entwicklungsmodell ist. Jedes einzelne V, das wir durchlaufen ähnelt seinem Nachfolger und seinem Vorgänger. Vielleicht variieren die Aufwände zwischen den einzelnen Disziplinen, aber der Gesamtaufwand ist in jedem Zyklus vergleichbar.



Das Modell bewahrt uns natürlich auch nicht, während eines V-Zyklus' in die falsche Richtung zu laufen und die Erkenntnis erst am Ende bei der Auslieferung des Arbeitsstandes an unseren Kunden zu erlangen. Der schönste Plan wäre also zu Nichte gemacht, wenn wir blauäugig auf unsere Menschenkenntnis vertraut hätten und nicht noch zusätzliche, teure Releases "vorsichtshalber" eingeplant hätten.

Agile Entwicklung mit Methoden wie SCRUM, Continuous Integration und Continuous Delivery nimmt die Hektik vor Auslieferungen und Verteilt die Aufwände ungleichmäßiger, dafür in kleineren Abschnitten. Ramp-Up- und Ramp-Down-Phasen sind weniger schwerwiegend, weil seltener nötig. Eine Projektschieflage kann früher erkannt und mit kleinen Korrekturen wieder auf Linie gebracht werden.

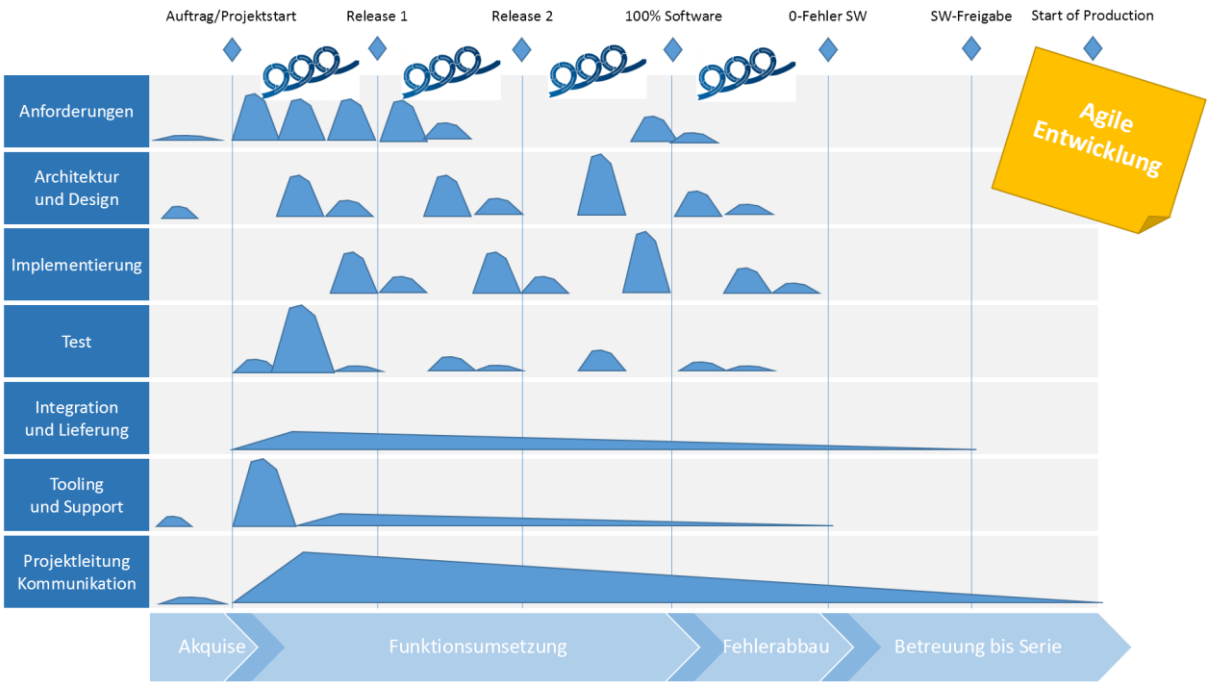
Langfristige Planung aber ist nicht das Ziel, der Vorteil liegt hier vor allem darin, keine Lieferungen verschieben zu müssen, weil es praktisch zu jeder Zeit eine ausreichend reife, aber vor allem getestete und abgestimmte Software gibt. Nicht nur beim Kundenrelease. Aber statt Zeitpläne über den Haufen zu werfen, gibt es natürlich immer das Risiko, dass Funktionen die nicht als hoch prior bewertet wurden auf der Strecke bleiben.

Was ist also unser Mittel der Wahl? Konservatives und vermeintlich sicheres Vorgehen nach dem iterativen V-Modell oder der moderne, agile Ansatz mit Platz für Korrekturen, falls der Kunde seine Meinung ändert?

Im Rahmen der mitgeltenden Vorgaben und einzuhaltenden Normen (beispielsweise ISO26262) können wir uns also aussuchen, welchen Tod wir sterben wollen. Die größten Unterschiede der zwei Pole ergeben sich beim Test, Dokumentation und vor allem der Kommunikation.

Test- und Toolaufwände sind sicher initial bei Agiler Entwicklung höher, weil wir eine Tool-Landschaft etablieren müssen, die Continuous Integration überhaupt erst möglich macht. Testfälle werden per Test Driven Development schon zu Anfang formuliert und mindestens

vorbereitet. Dafür sparen wir uns dedizierte Testläufe vor jedem Kundenrelease inkl. zehrender Bugfix-Schleifen.



Folgen wir dem Prozess des V-Modells ganz strikt, gibt es nur sehr wenige Punkte, an denen wir mit dem Kunden und späteren Nutzern unseres Systems auch kommunizieren. Anders bei Agiler Entwicklung, bei der wir permanent mit dem Kunden in Kontakt stehen. Das erzeugt Aufwand, verhindert aber auch, zu lange in die falsche Richtung zu laufen.

Eine anfängliche Abschätzung der Aufwände fällt sicherlich auf den ersten Blick beim konservativen Vorgehen nach dem iterativen V-Modell leichter. Trotzdem wird in beiden Fällen Parkinsons Gesetz zuschlagen, wenn wir zu vorsichtig schätzen:

„Arbeit dehnt sich in genau dem Maß aus, wie Zeit für ihre Erledigung zur Verfügung steht.“
Apropos Kommunikation - wir leben in modernen Zeiten. Wir wollen auch ein Stück vom Digitalisierungs-Kuchen abbekommen. Die Digitalisierung bietet Projekten vielfältige Möglichkeiten, das Arbeiten an Software an neue Gegebenheiten anzupassen. Jeder Mitarbeiter kann theoretisch seine Arbeit von jedem Fleck der Welt erledigen, solange Strom und Internetanschluss verfügbar sind.

Mit verteilten Teams können wir die besten Entwickler aus aller Welt zusammenbringen, ohne dass auch nur einer seinen Wohnort wechseln müsste. Ganz abgesehen von unterschiedlichen Gehaltsgefügen jenseits der Grenze, die uns am teuren Entwicklungsstandort auch noch zu Gute kommen.

So einfach ist es natürlich nicht! Vergessen wir nicht, was wir alles in der Kaffeeküche oder am Flur besprechen.
Wie hoch ist die Hürde, den Telefonhörer in die Hand zu nehmen und mit einem Entwicklerkollegen am anderen Ende der Welt in einer Sprache zu sprechen, die nicht unserer beider Muttersprache ist? Die Hürde können wir natürlich abbauen, wenn wir uns regelmäßig

sehen oder Small Talk führen. Dazu braucht es aber Reisebudget - was nicht nur monetär, sondern auch zeitlich eingepreist werden muss.

Gehen wir noch weiter: Wissen Sie als Autofahrer, was Autobahnblinken oder Komfortblinken bedeutet?
Würden wir die Frage in einem Land stellen, in dem Straßen außerhalb der Mega Cities nicht einmal Straßenmarkierungen haben - und das sind die beliebten "Low Cost Countrys" – würde die Antwort anders ausfallen. Kulturelle Unterschiede führen also auch dazu, dass wir wesentlich mehr Zeit aufwenden müssen, um unsere Anforderungen präziser zu spezifizieren. Diese zusätzliche Zeit fällt aber wieder in "High Cost Countrys" an und nicht dort, wo die Arbeitsstunde am Papier günstiger ist.

Die erfahrensten Leute, denen man Funktionen nicht erklären müsste werden aber nicht ihr Entwicklerleben lang dieses Lohngefälle hinnehmen. Sie werden versuchen, aufzusteigen oder sich auf lukrativere Stellen im Ausland bewerben. Wir verlieren also die besten Leute, die wir mit viel Zeitaufwand angelernt haben wieder sehr schnell.

Weitere Überlegungen
Da Software oftmals Abhängigkeiten zu Nachbarsystemen hat ist es erforderlich Informationen hierzu einzuholen. Es werden Informationen zum Zielsystem und vieles mehr benötigt. Bleiben solche Beistellungen aus, kann das zu Verzögerungen und Mehraufwänden führen. Ein Management der Beistellungen kann daher sehr wichtig werden. Eine Festlegung von Qualität, Quantität und dem Zeitpunkt der Beistellung kann Projekte retten.

Bei einer funktionsgetriebenen Sichtweise auf das Produkt dürfen wir außerdem nicht vergessen, dass banale Dinge, wie das Aufsetzen einer Projektumgebung, Architekturerstellung und qualitätssichernde Maßnahmen eingeplant werden müssen. Hier können uns Checklisten helfen.

Ungenaue Lastenhefte können zu erheblichen Missverständnissen beim Funktionsumfang der zu entwickelnden Software führen. Eine auf diesen Missverständnissen basierende Schätzung wird sich erst im Projektverlauf als fehlerhaft herausstellen und Zeit, Geld und Nerven kosten. Die Erstellung eines Projekthandbuches während der Akquise hilft dabei, mit dem Kunden vor Angebotsabgabe Ziele, Aufgaben und Zeitschiene zu reflektieren.

Oftmals sind Lastenhefte zum Zeitpunkt der Ausschreibung eines Projektes nicht bis ins Detail ausspezifiziert. Um den Kunden die Basis der Schätzung transparent zu machen, sollten die Annahmen möglichst genau im Angebot niedergeschrieben werden. Eine persönliche Abstimmung der Annahmen mit dem Kunden vor Angebotsabgabe kann Missverständnissen vorbeugen und Annahmen früh bestätigen bzw. die Möglichkeit geben, Schätzungen anzupassen. Ein schon etabliertes Change Management System kann uns hier unterstützen.

Es zahlt sich aus während der Angebotserstellung die technischen und organisatorischen Herausforderungen oder gar Risiken des Projektes im Blick zu haben. Diesen Punkten sollte bei der Schätzung genug Zeit gewidmet werden, um komplexe Themen durchdringen und Lösungsalternativen bewerten zu können.