Unterstützung der Release-Planung mittels priorisierter Use Cases [1]

Peter Küng, Heinrich Krause, Carl F. Worms
E-Mail: peter.kueng@csfs.com
Credit Suisse, Postfach 600, 8070 Zürich

Abstract: Requirements represent essential input for software projects and, thus, need to be defined clearly. Unfortunately requirements often are not prioritised systematically. As a consequence, it is not determined in the early phases of a project which functions (or functional requirements) are going to be covered by which release. This paper suggests a technique to prioritise requirements and to assign them to software releases. Special emphasis is given on the fact that cost and benefit of each functional requirement are taken into consideration before a decision is taken.

1 Einführung

Die Verkürzung der Time-to-Market ist eine Herausforderung für sämtliche Branchen. In der Pharmaindustrie beträgt die Time to Market typischerweise mehrere Jahre; im Finanzsektor liegt die Zeitdauer für die Erstellung neuer Produkte bei wenigen Monaten bis gar Jahren. Da die Konkurrenzfähigkeit von Unternehmen zu einem beachtlichen Teil durch die Fähigkeit, neue Produkte in kurzer Zeit auf den Markt zu bringen, bestimmt wird, kommt dieser Thematik entsprechend hohe Bedeutung zu. Dabei gilt es zu berücksichtigen, dass die Kreation neuer Produkte – und dies gilt ganz besonders im Finanzdienstleistungssektor – auch eine Modifikation der IT bedeutet: neue Anwendungen werden benötigt, neue Daten müssen in eventuell neuer Struktur gespeichert werden, neue Kanäle müssen bedient werden. Dies wiederum hat zur Folge, das die unternehmensinterne Applikationsentwicklung – oder allgemein gesprochen die Softwareindustrie – in der Lage sein muss, IT-Systeme in kürzerer Zeit verfügbar zu machen.

In den letzten Jahren sind verschiedene Vorschläge entworfen worden, um die Entwicklungszeit für Software zu verkürzen. Dazu gehören unter anderem die beiden folgenden:

1 Dieser Beitrag ist erschienen in: Spitta, Thorsten; Borchers, Jens; Sneed, Harry (Hrsg.), "Software Management 2002", GI-Edition – Lecture Notes in Informatics. Gesellschaft für Informatik, Bonn 2002, S. 77-87; ISBN: 3885793520.

1. Vorgehensmodelle, bei welchen Software in Iterationen entwickelt wird. Ein Ansatz, bei welchem Iterationen einen integralen Bestandteil darstellen, ist der in der Softwareindustrie stark beachtete Rational Unified Process, welcher zum Beispiel in Kruchten [Kr00] gut beschrieben ist.
2. Extreme Programming (XP): Dies ist ein Mix "bewährter" Software-Entwicklungstechniken oder Prinzipien. Die beiden ersten XP-Prinzipien lauten gemäss Beck [Be99]: (1) Planning Game: Die Kunden bestimmen Umfang und Zeitpunkt der einzelnen Software Releases. (2) Small Releases: Die Releases enthalten wenig neue Funktionalität; sie folgen jedoch in kurzen Zeitabständen, zum Beispiel monatlich.

Es ist hier nicht der Ort um eine systematische Gegenüberstellung der beiden Ansätze vorzunehmen. Es seien lediglich zwei Elemente ins Feld geführt. Erstens, der XP-Ansatz ist primär für kleinere bis mittelgrosse Projekte geeignet. In grösseren Projekten wird die Betonung der Architektur zunehmend zu einem kritischen Faktor [Pa01]. Zweitens, die ausgeprägte Verkürzung der Release-Zyklen – wie sie in XP partiziert wird – bedingt, dass die Tests sehr gut organisiert und weitgehend automatisiert werden.

Bild 1: Die Evolution des Wasserfallmodells (in Anlehnung an [Be99])

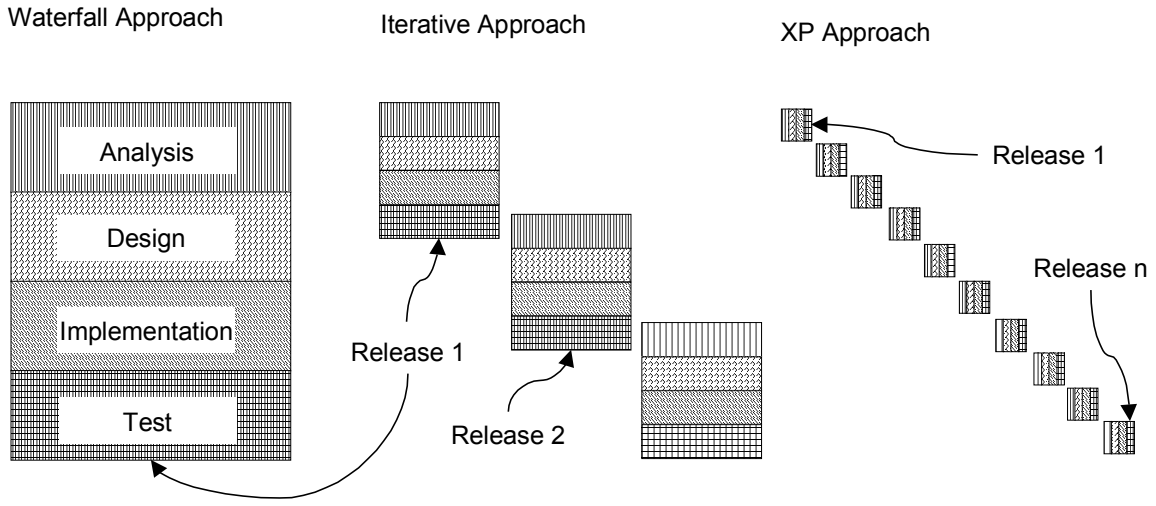


Bild 1 veranschaulicht den Sachverhalt: Während bei den traditionellen, Wasserfall-orientierten Arbeitsweisen in einem bestimmten Zeitabschnitt (z.B. zwei Jahre) eine einzige, jedoch umfassende Applikation erstellt wird, wird beim iterativen Ansatz (z.B. RUP) die Funktionalität der Applikation in mehrere Releases aufgeteilt; es werden im gleichen Zeitraum mehrere Releases eingeführt. Im XP-Ansatz wird das Konzept der kurzen Release-Zyklen noch viel stärker betont.

Mit der zunehmenden Zahl an Releases, die in einem bestimmten Zeitraum erstellt werden, wird auch die Frage, welchen Umfang ein bestimmter Release haben soll, immer wichtiger.

2 Verwandte Arbeiten und existierende Ansätze

Das Thema "Software-Release-Planung" findet in der Forschung relativ wenig Beachtung. Einige Forscher, die sich mit dem Thema befassen – z.B. [KTY99], [XH98] – konzentrieren sich auf die Fragestellung nach dem optimalen Zeitpunkt zur Freigabe einer Software (determination of software release time). Die Frage, welche Funktionalität die einzelnen Software-Releases enthalten sollen, wird in der Literatur jedoch selten behandelt. Eine partielle Quelle ist SPICE (Software Process Improvement and Capability dEtermination), eine internationale Initiative für das Assessment von Software-Prozessen. Im Prozessmodell, welches durch eine SPICE-Arbeitsgruppe im Jahre 1995 beschrieben wurde, ist die Priorisierung von Anforderungen wie folgt positioniert: Das Prozessmodell von SPICE besteht aus Kunden-/Lieferantenprozessen, Engineering-, Support-, Management- und Organisations-Prozessen. In der Kategorie Engineering-Prozesse ist ein Sub-Prozess "Determine release strategy" definiert. Die Tätigkeit, die in diesem Subprozess auszuführen ist, lautet: "Priorisiere die Anforderungen und bilde sie auf die zukünftigen Releases ab" [SP95]. Als Zusatzinformation wird lediglich folgender Hinweis gegeben: "Es ist nicht immer sinnvoll, ein einziges Produkt (Release) zu erstellen, welches sämtliche Anforderungen enthält. Manchmal ist es besser, die Anforderungen zu priorisieren und eine Reihe von Releases zu erstellen, welche die einzelnen Anforderungen stufenweise abdecken; beispielsweise um schon frühzeitig einen bestimmten Marktanteil erreichen zu können. Als möglicher Basis-Input dient das Software-Vorgehensmodell (Wasserfallmodell, Spiralmodell, etc.)." [SP95].

Eine andere Quelle ist der Rational Unified Process. Im RUP-Ansatz steht jedoch weniger die Releaseplanung als vielmehr die Iterationsplanung im Vordergrund.[2] Die Releaseplanung wird auf der Basis von priorisierten Use Cases, welche die funktionalen und nicht-funktionale Anforderungen beschreiben, vorgenommen. Hinsichtlich der Art und Weise, wie Use Cases zu priorisieren sind, macht der RUP nur sehr allgemeine Aussagen; vgl. [Kr00].

Als dritte Quelle kann der Extreme-Programming-Ansatz genutzt werden. In diesem Ansatz ist die Releaseplanung von besonders grosser Bedeutung, ist doch die Zahl der Releases bedeutend höher als in traditionellen Vorgehensweisen. Nichtsdestotrotz sind die Aussagen, wie die Releases geplant werden sollen, nicht sehr konkret. Beck (von vielen als Vater des Extreme Programming betrachtet) betont, dass die Priorisierung aus der Perspektive des Kunden

2 Um einen neuen Release zu produzieren sind eventuell mehrere Iterationen notwendig. Deshalb müssen Iterationen und Releases nicht deckungsgleich sein.

vorzunehmen ist, indem sich der Kunde die Frage stellt, was er an Funktionalität für einen bestimmten Geldbetrag erhalten kann [Be99].

Wie können Anforderungen priorisiert werden? Diese von der Software-Entwicklungsmethode weitgehend unabhängige Frage wird seit vielen Jahren diskutiert. Dabei haben sich zwei Kategorien von Ansätzen herausgeschält:

- *Relative Beurteilung:* Die einzelnen Anforderungen werden nicht absolut (für sich alleine), sondern relativ zu den anderen bewertet. Die einzelnen Anforderungen werden also bezüglich der anderen Anforderungen positioniert.
- *Absolute Beurteilung:* Die einzelnen Anforderungen werden mittels einer absoluten Grösse (z.B. einer Punktzahl) bewertet; ein expliziter Vergleich zwischen den einzelnen Anforderungen wird nicht vorgenommen.

Welches ist der am besten geeignete Ansatz? In der Literatur werden die relativen Beurteilungen den absoluten Beurteilungen vorgezogen. Karlsson und seine Mitarbeiter [KWR98] haben sechs Priorisierungsansätze evaluiert und sind zum Schluss gelangt, dass der Analytic Hierarchy Process (AHP) die meisten Vorteile auf sich vereinigen kann. Dazu äussern sich die Autoren wie folgt: "... wir sind zum Schluss gelangt, dass AHP der vielversprechendste Ansatz ist. Er führt zu den zuverlässigsten Ergebnissen: sie basieren auf Verhältniszahlen, sie sind fehlertolerant und sie beinhalten Konsistenzprüfungen." [KWR98].

Beim AHP-Ansatz werden die einzelnen Anforderungen paarweise verglichen. Dabei wird nicht nur ermittelt, ob Anforderung n wichtiger ist als Anforderung n+1, sondern es wird auch ausgedrückt, um wie viel Anforderung n wichtiger ist [Sa80].[3]

In der Softwareindustrie hat sich der AHP-Ansatz bis jetzt nicht durchsetzen können. Dies dürfte damit zusammenhängen, dass die Handhabung des Ansatzes zeitaufwändig ist. Berücksichtigt man beispielsweise, dass sich die Zahl der vorzunehmenden Vergleiche nach der Formel n*(n-1)/2 berechnet (wobei n der Anzahl Anforderungen entspricht), so zeigt sich, dass schon bei nur 20 Anforderungen 190 Vergleiche durchgeführt werden müssen.

Zusammenfassend kann festgehalten werden, dass:

- die Priorisierung von Anforderungen als wichtig erachtet wird
- der AHP-Ansatz in der Literatur mehrfach beschrieben wurde, er aber in der Softwareindustrie kaum praktiziert wird

3 Um den relativen Wichtigkeitsunterschied zwischen zwei Anforderungen auszudrücken, sind fünf Stufen vorgesehen: (1) of equal importance; (2) moderate difference; (3) essential difference; (4) major difference; (5) extreme difference

- in der Softwareindustrie die Anforderungen bei grösseren Projekten priorisiert werden, die der Priorisierung zu Grunde liegenden Kriterien jedoch häufig nicht konkretisiert werden.

Aufgrund dieser Situation haben wir ein Verfahren entworfen, welches es erlaubt, Anforderungen an neu zu erstellende IT-Systeme in nachvollziehbarer Weise zu priorisieren. Dieser wird nachfolgend beschrieben.

3 Ein Use-Case-basierter Ansatz

Der hier beschriebene Ansatz wurde im Finanzdienstleistungsunternehmen Credit Suisse entworfen. Unser Priorisierungsansatz kann durch vier Merkmale charakterisiert werden:

- Das Requirements Engineering basiert auf Use Cases; unabhängig davon, ob die Anforderungen priorisiert werden oder nicht.
- Um die einzelnen Anforderungen priorisieren zu können, werden sie mit absoluten Werten versehen. Die einzelnen Anforderungen werden also nicht paarweise verglichen (wie dies beim AHP-Ansatz gemacht wird), sondern mittels absoluten Werten eingestuft. Damit wird die Zahl der von den Evaluatoren vorzunehmenden Entscheidungen in engeren Grenzen gehalten.
- Das dritte Merkmal besteht darin, dass sich die Priorität einer einzelnen Anforderung aus dem Kosten/Nutzenverhältnis ergibt. Nebenbei sei vermerkt, dass in vielen Werkzeugen, die für das Requirements Engineering eingesetzt werden, lediglich der Nutzen, den eine einzelne Anforderung stiftet, in Betracht gezogen wird.
- Viertes und wichtigstes Merkmal: Bei jeder Anforderung werden Kosten- und Nutzenseite anhand einheitlicher Kriterien “gemessen”. Durch die Offenlegung der Kriterien wird die Objektivität und Präzision der Schätzung erhöht.

3.1 Beschreibung der Anforderungen mit Use Cases

Use Cases wurden von Jacobson und seinen Mitarbeitern [JCJ92] vorgeschlagen. Sie sind heute als Instrument zur Beschreibung von Anforderungen breit anerkannt. Dazu beigetragen haben, dass Use Cases einerseits Teil des Rational Unified Process sind, und andererseits die graphische Notation (Use-Case Diagrams) Bestandteil der Unified Modeling Language (UML) ist. Use Cases dienen dazu, die Anforderungen, welche ein IT-System zu erfüllen hat, aus der Perspektive der Systembenutzer (Aktoren) zu beschreiben. Use Cases können – in Anlehnung an Kruchten [Kr00] und Kulak /Guiney [KG00] – wie folgt charakterisiert werden:

- Ein Use Case umfasst eine oder mehrere Interaktionen zwischen Aktor(en) und IT-System. Ein Aktor ist meistens ein Anwender; ein Aktor kann aber auch ein anderes IT-System sein, welches mit dem zu erstellenden IT-System interagiert.
- Ein Use Case liefert dem Aktor ein nützliches Ergebnis; ein Use Case stellt einen vollständigen und sinnvollen Ablauf von Ereignissen dar.
- Ein Use Case beschreibt *was* das IT-System macht – und nicht *wie* das IT-System etwas macht. Ein Use Case ist somit eine Black Box.
- Die Dokumente, in denen Use Cases beschrieben sind, heissen "Use-Case Diagram" und "Use-Case Specification"

Die funktionalen Anforderungen werden einerseits in Form von Use-Case Diagrams (auch "Strichmännchen"-Diagramme genannt) beschrieben. Sie zeigen, welche Aktoren welche Use Cases ausführen und dienen der Übersicht. Ferner kommt in den Use-Case Diagrams die Struktur zum Ausdruck. So wird beispielsweise beschrieben, welche Use Cases welche Sub-Use Cases nutzen (sogenannte Inclusions), und welche Use Cases eine Erweiterung eines Basis-Use Cases darstellen (Extensions).

Die genauere inhaltliche Beschreibung der funktionalen Anforderungen erfolgt nicht in grafischer Form (Use-Case Diagram), sondern in strukturierter, verbaler Form (Use-Case Specification). Für jeden Use Case wird unter anderem beschrieben, (a) welches Ziel der Use Case verfolgt, (b) wie der normale Ablauf und die alternativen Abläufe aussehen, (c) welche Vorbedingungen (Pre-Conditions) erfüllt und welche Resultate (Post-Conditions) erbracht werden müssen, und (d) welche nicht-funktionalen Anforderungen einzuhalten sind. Die nicht-funktionalen Anforderungen, welche für sämtliche Use Cases Gültigkeit haben, werden in der Supplementary Specification beschrieben; die Use-Case-spezifischen, nicht-funktionalen Anforderungen kommen (als mehrwertiges Attribut) in der Use-Case Specification zum Ausdruck.

3.2 Priorisierung von Use Cases anhand von Kosten und Nutzen

Wie weiter vorne schon kurz erwähnt, wird die Priorität einer einzelnen Anforderung durch das geschätzte Kosten/Nutzenverhältnis bestimmt. Dieser Punkt wird in der Praxis sehr oft nicht berücksichtigt. Es ist nicht untypisch, dass die Priorisierung von Anforderungen lediglich aus der Sicht des Nutzens vorgenommen wird, den eine einzelne Anforderung (nach Realisierung des Systems) stiftet. Korrekterweise müssen sowohl der zukünftige Nutzen einer Anforderung als auch die entstehenden Kosten mit ins Kalkül einbezogen werden. Funktionen die keinen Nutzen stiften, aber dennoch realisiert werden müssen (z.B. bedingt durch gesetzli

che Vorgaben), werden als zwingend betrachtet und nicht in die Kosten/Nutzenrechnung einbezogen.

Bild 2 zeigt wie das Prinzip der Priorisierung funktioniert: Hohe Priorität erhalten jene Anforderungen (Use Cases), welche einerseits einen hohen potentiellen Nutzen (contribution) abwerfen und andererseits relativ geringe Kosten (cost) verursachen. Anforderungen mit hoher Priorität werden im ersten Software-Release umgesetzt. Mittlere Priorität erhalten jene Anforderungen, welche (a) hohe Kosten verursachen und hohen Nutzen versprechen, oder (b) geringe Kosten verursachen und geringen Nutzen versprechen. Anforderungen mit mittlerer Priorität werden je nach vorhandenen Ressourcen im ersten oder in späteren Releases implementiert. Anforderungen mit tiefer Priorität werden nochmals geprüft, bevor sie in die Releaseplanung aufgenommen werden.

Bild 2: Grundraster zur Priorisierung von Use Cases

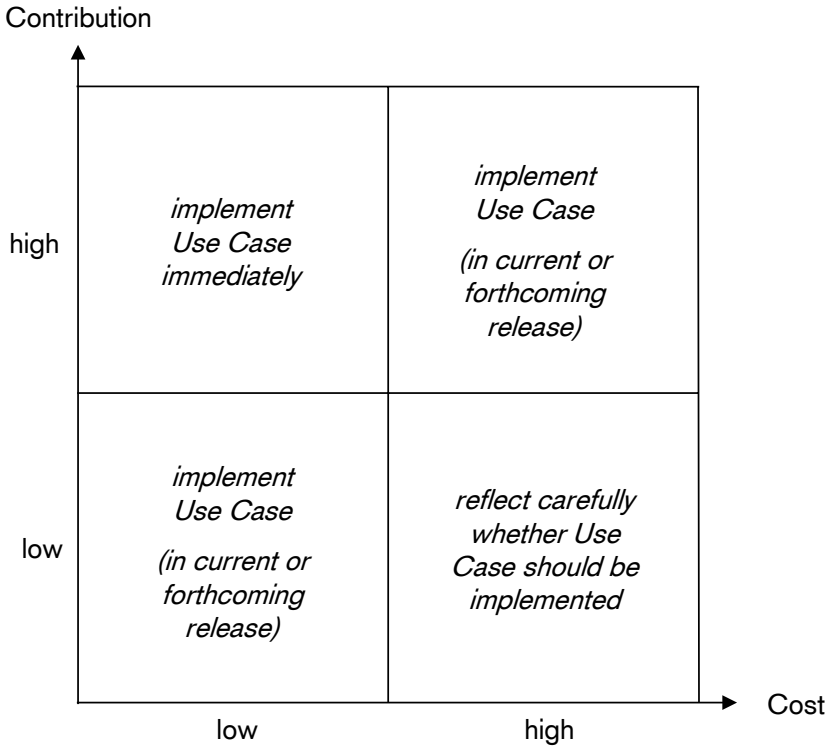


Bild 2 zeigt, welche grundsätzliche Überlegung hinter dem hier vorgestellten Priorisierungsansatz steckt. Nicht zum Ausdruck kommt darin, wie die einzelnen Prioritäten zu Stande kommen. Dieser Aspekt wird im nächsten Abschnitt dargelegt.

4 Die verwendeten Kriterien

Wie in Abschnitt 3 dargelegt, wird die Priorität der einzelnen Anforderungen (Use Cases) anhand von Kriterien ermittelt. In unserem Ansatz werden sowohl die Nutzen- als auch die Kostenseite mit *je fünf Kriterien* eingeschätzt. Die verwendeten Kriterien basieren zum Teil auf dem "Information Economics"-Ansatz von Parker und Benson [PB88]. Als Massgrösse werden nicht Euro oder Dollars verwendet, sondern Punkte. Punkte, die aus der Nutzenbetrachtung resultieren, werden als *Contribution Points*, Punkte, die aus der Kostenperspektive resultieren, werden als *Cost Points* bezeichnet.

Um die **Contribution Points** der einzelnen Use Cases zu schätzen, werden die folgenden fünf Kriterien in Betracht gezogen:

1. *Customer expectation:* Erwartet der Kunde beziehungsweise der Anwender der neuen Software, dass dieser Use Case zur Verfügung steht?
2. *Reduction of internal cost:* Trägt die Implementierung des Use Case dazu bei, dass die internen Kosten reduziert werden?
3. *Quality of information:* Bewirkt die Implementierung des Use Case, dass hochwertigere Informationen vorliegen, um besser fundierte Entscheidungen treffen zu können?
4. *Competitive advantage:* Führt die Implementierung des Use Case zu einer besseren Positionierung im Markt?
5. *Frequency:* Wird der Use Case häufig angestossen?

Funktionalitäten (Use Cases) stiften nicht nur Nutzen; sie verursachen auch Kosten. Die Kosten, die mit der Realisierung eines Use Case verbunden sind, setzen sich aus drei Kategorien zusammen. Zum einen sind dies die Softwareentwicklungskosten (Kosten für Softwareentwurf, Implementierung, Test, Installation). Ferner müssen die Kosten berücksichtigt werden, die durch den Betrieb (Operating und Support) entstehen. Und schliesslich werden in der Kategorie Kosten auch die technischen Risiken berücksichtigt, die mit der Implementierung eines Use Case verbunden sind.

Um die **Cost Points** eines einzelnen Use Case zu schätzen, werden die folgenden fünf Kriterien in Betracht gezogen:

1. *User Interface:* Bedingt der Use Case eine aufwändige Benutzerschnittstelle?
2. *Business logic:* Bedingt der Use Case, dass viel Geschäftslogik programmiert/ implementiert werden muss?

3. *Back end/ data store:* Bedingt der Use Case, dass neue Services[4] erstellt werden müssen oder dass Datenstrukturen geändert werden müssen?
4. *Cost of operation:* Bewirkt der Use Case, dass die Kosten für den laufenden Betrieb ansteigen?
5. *Technical risk:* Ist die technische Realisierung des Use Case mit Risiko behaftet?

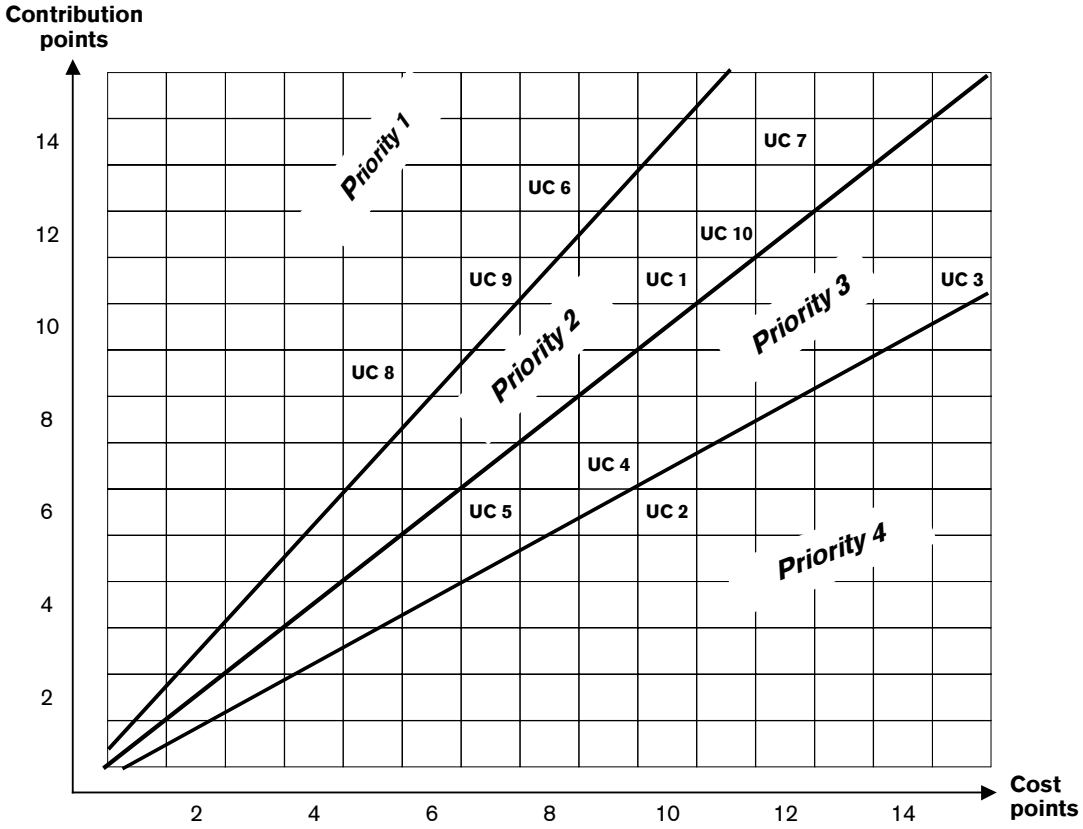
Jeder Use Case wird anhand der zuvor erwähnten zehn Kriterien bewertet. Dabei werden pro Kriterium 0 bis 3 Punkte vergeben. Ferner können die einzelnen Kriterien gewichtet werden. Höchste Priorität erhalten jene Use Cases, welche pro Contribution Point am wenigsten Cost Points verursachen oder jene, welche pro Cost Point am meisten Contribution Points generieren. Die Priorisierung kann als Formel ausgedrückt werden. So könnte beispielsweise folgender Algorithmus definiert werden:

```
IF (Sum of Contribution Points / Sum of CostPoints) < 0.7 then Priority = 4
IF (Sum of Contribution Points / Sum of CostPoints) < 1.0 then Priority = 3
IF (Sum of Contribution Points / Sum of CostPoints) < 1.4 then Priority = 2
IF (Sum of Contribution Points / Sum of CostPoints) >= 1.4 then Priority = 1
```

Um die ermittelten Prioritäten zu kommunizieren, empfiehlt sich die grafische Aufbereitung. Dadurch kommt die Positionierung jedes einzelnen Use Cases sehr schnell und klar zum Ausdruck. Bild 3 zeigt ein Beispiel, in dem zehn Use Cases in vier Prioritätsstufen eingeteilt wurden.

[4] Den Begriff Service definieren wir wie folgt: “Ein Service ist eine Leistung, die von einer Leistungseinheit (Programm/ Komponente) erbracht wird. Die Leistungseinheit erhält den Input in Form festgelegter Parameter, sie respektiert definierte Vorbedingungen und sie erfüllt die vereinbarten Nachbedingungen.” Beispiele von Services: ‘Eroeffne_Konto’, ‘Eroeffne_Kreditantrag’, ‘Liefere_Kreditertraege_Details’.

Bild 3: Visualisierung der priorisierten Use Cases anhand eines Beispiels



Die im Beispiel vorgenommene Priorisierung (siehe Bild 3) könnte wie folgt interpretiert werden: Sämtliche Use Cases, welche Priorität 1 oder 2 haben, werden in Release 1 implementiert. Use Cases mit Priorität 3 werden in Release 2 implementiert; Use Cases mit Priorität 4 werden nicht oder in einem späteren Release implementiert.

5 Voraussetzungen und Grenzen des Ansatzes

Damit der hier vorgeschlagene Ansatz praktiziert werden kann, muss eine wichtige Voraussetzungen gegeben sein: Die einzelnen Anforderungen müssen so genau beschrieben sein, dass die Kosten, die mit der Implementierung und dem Betrieb (operation) verbunden sind, sowie der Nutzen, der aus der Realisierung einer einzelnen Anforderung generiert wird, zuverlässig geschätzt werden kann. Werden die Anforderungen in der Form von Use-Case Specifications beschrieben, so dürfte diese Voraussetzung gegeben sein. Am Rande sei vermerkt, dass es nicht notwendig ist, dass sämtliche Use Cases die gleiche Granularität aufweisen, denn "grosse" Use Cases verursachen tendenziell höhere Kosten, sollten aber auch einen tendenziell höheren Nutzen generieren als "kleine" Use Cases.

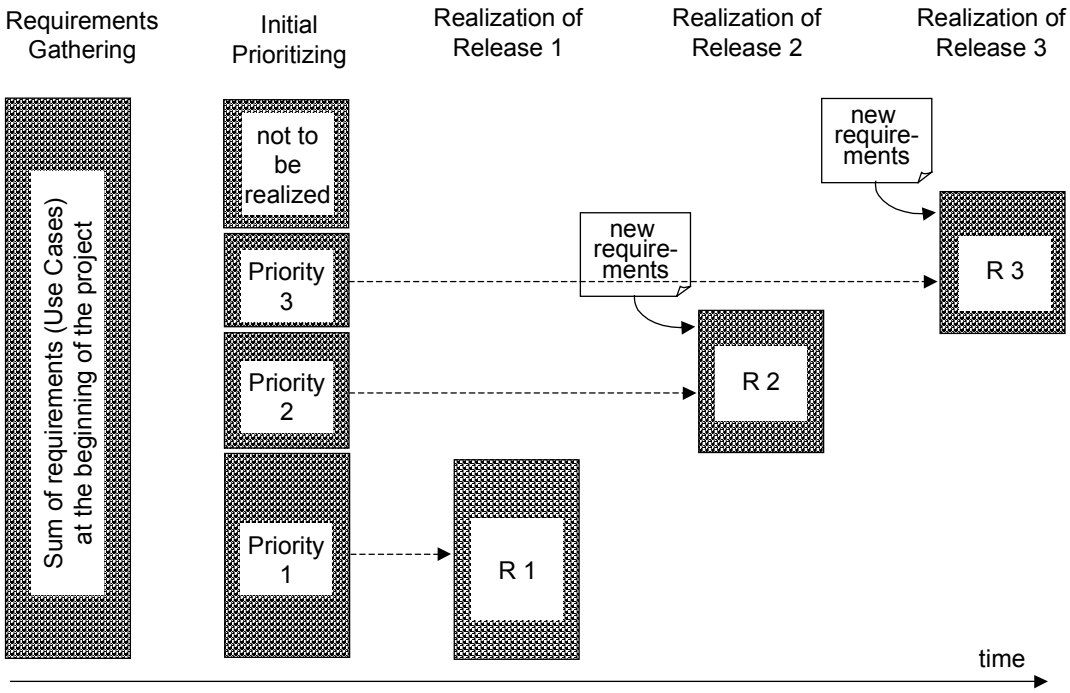
Wo liegen die Grenzen des vorgestellten Ansatzes? Wir sehen die drei folgenden Punkte:

- Nicht-funktionale Anforderungen werden nicht priorisiert und daher nicht berücksichtigt.
- Zwischen den einzelnen, zu priorisierenden Use Cases können Abhängigkeiten bestehen.[5] So kann es beispielsweise vorkommen, dass ein bestimmter Use Case sogenannte Inclusions beinhaltet. Dies bedeutet, dass ein Basis-Use Case nur dann ausgeführt werden kann, wenn der oder die zugehörigen Sub-Use Cases ebenfalls ausgeführt werden können.[6] Mit anderen Worten: Ein Basis-Use Case kann seine Funktionalität nur dann ausführen, wenn der Inclusion Use Case (falls er einen solchen nutzt) ebenfalls implementiert wird. Wir behandeln dieses Problem so, indem die Use Cases in einer ersten Bewertungsrunde als unabhängig betrachtet werden. Tritt nun der Fall ein, dass ein Basis-Use Case eine höhere Prioritätsstufe erhält als der zugehörige Inclusion-Use Case, so muss die Priorität des Inclusion-Use Cases künstlich erhöht werden.
- Der Umfang eines bestimmten Releases (z.B. Release 3 der Software xy) kann nicht vollumfänglich zu Projektbeginn mittels priorisierter Use Cases geplant werden; vgl. Bild 4. Neue Releases müssen neben den ursprünglich ermittelten Anforderungen auch neue Anforderungen abdecken: Anwenderwünsche (Use Cases), die bei der Planung von Release 1 noch nicht vorlagen, müssen berücksichtigt werden oder Fehlfunktionen müssen eliminiert werden. Mit anderen Worten, der Umfang der einzelnen Releases kann mit Hilfe priorisierter Use Cases nicht *vollumfänglich* geplant werden. Es ist jedoch möglich, den Umfang der in Zukunft zu erstellenden Releases teilweise zu planen; und dies kann als Fortschritt gewertet werden.

[5] Die Abhängigkeiten zwischen den einzelnen Use Cases werden in einem Use-Case-Übersichtsdiagramm visualisiert.

[6] Wir unterscheiden zwischen drei verschiedenen Typen von Use Cases: (1) Basis-Use Cases, (2) Inclusion-Use Cases, und (3) Extension-Use Cases.

Bild 4: Priorisierte Use Cases bestimmen den Umfang zukünftiger Releases wesentlich



Zusammenfassend kann festgehalten werden, dass Use Cases eine gute Basis bilden, um die funktionalen Anforderungen, welche an ein IT-System gestellt werden, zu priorisieren. Um eine sinnvolle Releaseplanung vornehmen zu können, ist die Priorisierung nicht nur aus Kundensicht, sondern auch aus der Sicht des Erstellers und Betreibers des IT-Systems vorzunehmen. Schliesslich ist zu beachten, dass das Priorisieren von Anforderungen auf Schätzungen und nicht auf exakten Messungen beruht. Mit einem guten Verfahren kann bewirkt werden, dass die Schätzgenauigkeit erhöht wird und die Releaseplanung zuverlässiger wird. Ein Teil Subjektivität wird jedoch immer vorhanden sein – die Frage ist nur, wie gross dieser Teil in der modernen Softwareindustrie sein darf.

Literaturverzeichnis

[Be99] Beck, Kent: Embracing Change with Extreme Programming. IEEE Computer, Vol. 32, No. 10 (October 1999), pp. 70-77.

[JCJ92] Jacobson, Ivar; Christensen, Magnus; Jonsson, Patrick: Object-Oriented Software Engineering – A Use Case Driven Approach. Addison-Wesley, Reading MA, 1992; ISBN: 0201544350.

[KWR98] Karlsson, Joachim; Wohlin, Claes; Regnell, Björn: An evaluation of methods for prioritizing software requirements. Information and Software Technology, Vol. 39, No. 14/15 (February 1998), pp. 939-947.

[KTY99] Kimura, A.; Toyota, T.; Yamada, S.: Economic analysis of software release problems with warranty cost and reliability requirement. Reliability Engineering and System Safety, Vol. 66, No. 1 (1999), pp. 49-55.

[KG00] Kulak, Daryl; Guiney, Eamonn: Use Cases – Requirements in context. Addison-Wesley, Reading MA, 2000; ISBN: 0201657678.

[Kr00] Kruchten, Philippe: The Rational Unified Process – An introduction. Addison-Wesley, Reading MA, 2000; ISBN: 0201707101.

[Pa01] Paulk, Mark: Extreme Programming from a CMM Perspective. IEEE Software, Vol. 18, No. 6 (November/December 2001), pp. 19-26.

[PB88] Parker, Marilyn; Benson, Robert: Information Economics – Linking Business Performance to Information Technology. Prentice-Hall, Englewood Cliffs NJ, 1988; ISBN: 0134645952.

[Sa80] Saaty, Thomas: The Analytic Hierarchy Process – Planning Setting Priorities, Resource Allocation. Mc-Graw-Hill, New York, 1980; ISBN: 0070543712.

[SP95] SPICE (Ed.): Software Process Assessment – Part 2 : A model for process management, version 1.00; verfügbar via: http://www.sqi.gu.edu.au/spice/suite/intro.html; Zugriff am 1. Juli 2002.

[XH98] Xie, M.; Hong, G.: A study of the sensitivity of software release time. The Journal of Systems and Software, Vol. 44, No. 2 (December 1998), pp. 163-168.