Grundlagen der Softwaretechnik

(SWT)

Prinzipien des Software Engineering
und ihre Umsetzung
in traditioneller und agiler Softwareentwicklung

Dieses Dokument enthält den

Roten Faden
und Vertiefung zu drei wichtigen Themen der Vorlesung,
als Ergänzung zu den Folien

© Prof. Dr. Kurt Schneider

FG Software Engineering
Leibniz Universität Hannover

Wintersemester 2023/2024

Inhalt

Der Zweck dieses Dokuments

In dieser Einführung wird erklärt, welche Themen in der Vorlesung behandelt werden. Dabei dient der erste Teil als Ergänzung zu den Folien. Nach einer Motivation und Einführung folgt jeweils ein Teil, der als „Roter Faden“ kurz den Zusammenhang der Themen und Folien bespricht. In der Vorlesung wird das mündlich gesagt, hier in kurzer Form zusammengefasst. Einzelheiten werden nicht wiederholt, Folien werden hier nicht noch einmal gezeigt.

Im zweiten Teil werden drei besonders wichtige Themen vertieft dargestellt. Bei diesen Themen hat es in der Vergangenheit oft gerade bei solchen Studierenden Schwierigkeiten gegeben, die die Vorlesung nicht besucht hatten, denn hier kommt es auf Begründungen und Erfahrungen an. Dieser zweite Teil mit den Vertiefungen soll dies abfedern. Idealerweise haben Sie in der Vorlesung die Erklärungen gehört und finden dann hier noch einmal eine Darstellung im Zusammenhang. Notfalls kommen Sie damit auch ohne Vorlesung zurecht. Vollständig lässt sich aber die Vorlesung nicht ersetzen.

Die Vorlesung ist nach Prinzipien des Software Engineerings gegliedert.

Software Engineering als Disziplin und als Begriff sind bei einer Konferenz im Jahr 1968 eingeführt worden. Die Entwicklung von Software sollte – wie andere Ingenieursdisziplinen – effizienter und planbarer werden.

Bis etwa zur Jahrtausendwende wollte man das durch disziplinierte und wohldokumentierte Ablaufbeschreibungen erreichen: Software-Entwicklungsprozesse und Vorgehensmodelle sollten Entwicklern helfen, das Wissen und die Erfahrung erfolgreicher Projekte wiederzuverwenden.

Um das Jahr 2000 änderten sich die Ziele: Nun wurde mehr Gewicht auf Schnelligkeit und Flexibilität in der Entwicklung gelegt. Dazu waren die traditionellen Prozesse zu schwerfällig. Seitdem haben sich SCRUM und andere agile Verfahren in der Praxis weit verbreitet. Aber auch die traditionellen Verfahren haben noch ihre Einsatzgebiete.

Da weder der eine noch der andere Ansatz in der Praxis alleine ausreicht, orientiert sich die Vorlesung an den allgemeinen Prinzipien, die im Software Engineering verfolgt werden – sowohl in den traditionellen als auch in den agilen Verfahren. Die Prinzipien werden unterschiedlich umgesetzt, und heute manchmal auch kombiniert. Daher ist es in der Praxis wichtig, beide Ansätze und die Grundideen dahinter (das sind die Prinzipien) zu kennen.

Zu den Materialien für die Vorlesung, die Übungen und die Klausur gehören zunächst die Folien, die in StudIP zur Verfügung stehen. Mit dem vorliegenden Dokument erhalten Sie zusätzlich ergänzende Erläuterungen über Hintergründe und Zusammenhänge, sowie vertiefte Erklärungen zu wichtigen Themen. Beachten Sie, dass alle Inhalte der Vorlesung danach ausgewählt wurden, wie relevant sie sind. Alle Themen sind daher Prüfungsgegenstand – nicht nur die vertieft erklärten.

Weitere Materialien sind ein Template (Vorlage) für eine Spezifikation, Teamaufgaben für Zusammenarbeit und die Übungsaufgaben, die in den Übungsgruppen besprochen werden.

Einführung in das Thema

Der traditionelle Entwicklungsansatz und Agile Softwareentwicklung

Softwareentwicklung findet seit etwa 50 Jahren auf professioneller Ebene in großen Projekten mit vielen teilnehmenden Entwicklern statt. Seitdem hat man versucht, diese Projekte zu unterstützen und Vorgehensweisen zu entwickeln, mit denen die Beteiligten erfolgreicher sind. Über viele Jahre hinweg entwickelte sich das, was man heute den „traditionellen Ansatz“ nennt. Hier werden Rollen, Ergebnistypen und Dokumente immer genauer geregelt und vorgeschrieben, damit auch neue Entwicklerinnen und Entwickler nach kurzer Einarbeitungszeit genau wissen, was sie tun sollen. Damit geht allerdings einher, dass Projekte alle Vorgaben kennen und berücksichtigen müssen. Ein gewisser Overhead (zusätzlicher, unproduktiver Aufwand) scheint unvermeidlich.

Um das Jahr 2000 waren viele kurze, kreative und innovative Projekte entstanden, die unmöglich alle diese Auflagen berücksichtigen konnten, wenn sie schnell genug fertig sein sollten. Es bildeten sich Gruppierungen, die ein anderes Vorgehen forderten, um den neuen Erfordernissen schneller, flexibler und innovative Projekte zu entsprechen. Schließlich gab es ein Treffen von ca. 15 Personen, bei dem das sogenannte „Agile Manifest“ www.AgileManifesto.org beschlossen wurde. Darin bricht man mit vielen bewährten Ansätzen der traditionellen Entwicklung und setzt ihnen ein revolutionär anderes Vorgehen entgegen.

Durch den scharfen Schnitt und auch die provokativen Formulierungen kam es in den folgenden Jahren zu einem regelrechten Methodenstreit, der mit ideologischen Argumenten geführt wurde. Nach etwa 10 Jahren beruhigte sich die Situation; nun hatten so viele Projekte die agilen Ansätze erprobt und waren nicht alle gescheitert. Andererseits zeigte sich, dass Agilität alleine auch keine Lösung für jedes Problem darstellt. Inzwischen kann man agile und traditionelle Methoden als verschiedene Werkzeuge im Baukasten professioneller Software Ingenieure betrachten: Man muss beide kennen, um heute erfolgreich und professionell in großen Softwareprojekten mitarbeiten zu können.

In Lehrbüchern und Vorlesungen des Software Engineering kann man diese Entwicklung nachverfolgen: Zunächst gab es nur reife und disziplinierte Prozesse (nach dem traditionellen Ansatz), dann spaltete sich die Community in Verfechter des etablierten Ansatzes und junge Wilde, die nur agil arbeiten wollten. Schon sehr früh hat Barry Boehm, einer der wichtigsten Experten des Software Engineering, darauf hingewiesen, dass beide Ansätze Gutes haben, das man nach Bedarf kombinieren könnte. Genau das geschieht heute in den meisten Unternehmen. In dieser Vorlesung möchte ich die Studierenden auf diese Realität vorbereiten. Hier versuche ich, die beiden Ansätze mit ihren Vor- und Nachteilen gleichberechtigt vorzustellen. Dabei geht es nicht um einen Besser/Schlechter-Vergleich, sondern um einen größeren „Werkzeugkasten“, aus dem man wählen kann.

Prinzipien, die in agilen, traditionellen und gemischten Projekten gelten

Interessanterweise kann man sowohl in traditionellen als auch in agilen Methoden einige gemeinsame Prinzipien erkennen, die überall befolgt werden; auch, wenn das auf unterschiedliche Weise geschieht. In der Vorlesung werden acht solche Prinzipien vorgestellt, die in allen Projekten zu bedenken sind. Sie bilden das Rückgrat der gegenwärtigen Entwicklungsmethoden und sind jede für sich verständlich und sinnvoll. Man sieht, dass bei Entwurf mit Information Hiding und beim Versuch, verständlich zu programmieren kaum ein Unterschied zwischen agil und traditionell besteht; bei Anforderungen und im Management geht man dagegen sehr unterschiedlich vor. In der Vorlesung sollen alle gegenwärtig üblichen Vorgehensweisen (traditionell, agil, gemischt) abgedeckt werden, damit die Studierenden einen guten Überblick erhalten und in der Praxis nützliches Wissen erwerben.

Der Rote Faden durch die Folien

Referenz: SWT-23.24-Teil-1-v30. Folien 3-6 sollen deutlich machen, wie wichtig und wie kompliziert Software heute ist. Wenn sie schwerwiegende Fehler hat oder ausfällt, führt das sofort zu großen materiellen Schäden und gefährdet oft auch Menschenleben.

Informatiker haben daher eine große Verantwortung für das, was sie abliefern. Im Software Engineering versucht man, Einsichten und Techniken zu sammeln, mit denen man möglichst „gute Software“ erstellen kann. Wie man an den agilen und den traditionellen Verfahren sehen wird, gibt es unterschiedliche Kriterien, was man unter guter Software versteht: Billig, effizient, fehlerfrei und so gestaltet, dass Kunden zufrieden sind. Die Frage ist, wie man das erreichen kann.

Früher gab es nur ungeplantes Handeln und disziplinierte Vorgehensweisen. Gut ausgebildete Softwareentwickler versuchten, nicht einfach drauf los zu programmieren, sondern vorher zu überlegen, zu planen und zu entwerfen. Mit den agilen Methoden wurden andere Prioritäten gesetzt. Hier ist es wichtiger, die Kunden zufriedenzustellen, und dazu schon früh lauffähige Software auszuliefern. Welcher Ansatz besser geeignet ist, hängt von den Zielen und Randbedingungen eines Projekts ab.

Egal, welches System man verwendet, beide Ansätze gehen davon aus, dass man bewährten Verfahren folgt. Je ein traditionelles und ein agiles Beispiel für solche Verfahren werden in der Einführung vorgestellt, damit man einen groben Eindruck hat, wie sie insgesamt funktionieren. Einzelheiten werden dann bei den Prinzipien behandelt. Das sind das ***V-Modell und SCRUM***.

Die Lernziele (F7) stecken ab, was mit der Vorlesung erreicht werden soll. Man kann nicht alles perfekt und im Detail lernen, oft reicht es auch zu wissen, wo man Details nachsehen kann. In anderen Fällen, wie zum Beispiel bei Design Patterns, sollten Sie mindestens einmal wirklich damit gearbeitet haben, nicht nur davon gehört haben. Dann ist die Chance größer, dass Sie auch in eigenen Projekten Design Patterns einsetzen können.

SWT setzt Programmierkenntnisse und Algorithmen/Datenstrukturen voraus. Viele andere Lehrveranstaltungen sind nützlich. Im SWT geht es darum zu verstehen, wie Sie alle Ihre Informatik-Kenntnisse in Projekten zur Wirkung bringen können. Auf SWT baut dann sehr viel auf: Wer über die Grundlagen hinaus Interesse an dem Thema findet, kann in Master-Vorlesungen und in praktischen Laborübungen viele Vertiefungen wählen. Auch für die Angebote anderer Lehrstühlen können Sie die Grundkenntnisse gut gebrauchen. Das Software-Projekt (SWP) ist dabei ganz besonders hervorgehoben: Im 5. Semester steht die Durchführung eines kompletten Projekts im Studienplan. Hier können Sie alles Gelernte einsetzen, von den Datenstrukturen und der Komplexitätstheorie bis hin zur Softwaretechnik. Letztlich sind alle Lehrveranstaltungen aus diesem Bereich auch für praktische Zwecke im Beruf ausgerichtet, jeweils ein anderer Aspekt.

Da die Geschichte des Software Engineering (SE) noch kurz ist, sollte man die Anfänge kennen: Auf der berühmten NATO Science Conference 1968 hat Prof. Bauer das Wort „Software Engineering“ geprägt (F12). Was mit Ingenieursprinzipien gemeint ist (F13), sollten Sie an Beispielen aus der Software erklären können. Beispielsweise: Qualitätsbewusstsein führt dazu, dass man ein Programm sorgfältig testet und nicht schon zufrieden ist, wenn es nicht abstürzt. Andererseits fordert das Kostendenken, mit dem Testen auch wieder aufzuhören, sobald der Aufwand größer wird als der Nutzen.

Sie werden feststellen, dass immer wieder in der Informatik von Modellen gesprochen wird. Bei jedem Modell ist wichtig zu wissen: *Wovon* es eigentlich ein Modell ist? Wieso man das Modell entwickelt hat und wer es benutzen soll? Hat man diese Fragen beantwortet, weiß man auch, was an dem Modell wichtig (relevant) ist. Dazu F15-16.

Folie F17 sollte man auswendig lernen, denn diese Grundbegriffe sind etwas anders definiert, als man vielleicht erwartet hätte. Diese Definitionen kommen aus einer Norm (Std), sind also weltweit bekannt und damit auch für uns in Hannover für die Verständigung verbindlich.

Wie eingangs angekündigt, ist die Vorlesung um acht Prinzipien aufgebaut. Sie bilden das Zentrum (F22), von dem aus erklärt wird, wie jedes Prinzip in traditionellen und wie in agilen Verfahren umgesetzt wird.

Die Zeitschiene auf F20 zeigt einige der bekannteren Meilensteine im Software Engineering. Die meisten davon werden in der Vorlesung angesprochen. Einige der Entdeckungen oder Erfindungen kann man dem agilen bzw. dem traditionellen Ansatz zuordnen. Die anderen können von beiden eingesetzt werden. F22 zeigt die acht Prinzipien. Sie werden als Hauptgliederung der Vorlesung immer wieder auftauchen. Um Ihnen die Zuordnung zu erleichtern, sind Aspekte, die nur für traditionelle Verfahren gelten, links mit einem blauen Streifen gekennzeichnet. Agile Aspekte haben rechts einen gelben Streifen. Von beiden verwendbare Aspekte haben keinen Streifen. Gerade im Entwurfs- und Programmierbereich gibt es viele Techniken, die in allen Projekten eingesetzt werden können, ohne gegen das jeweilige Prinzip zu verstoßen.

Prinzip 1: Systematisch arbeiten

Einführung und Motivation

Systematisch zu arbeiten bedeutet, sich zuvor zu überlegen, wie man vorgehen will. Nicht, was einem spontan in den Sinn kommt, wird gemacht, sondern was eine Methode oder Vorgehensweise vorgibt. Das Gegenteil von systematisch ist in diesem Sinne „ad-hoc“. F29 zeigt, wieso man das wollte: systematisch arbeiten.

Ab F30 (links, blau) sieht man ein bekanntes Beispiel für einen traditionellen Ansatz: Das Wasserfallmodell, Inbegriff disziplinierter, traditioneller Vorgehensweise. Auf F31 ist der eigentliche Wasserfall (rechts) eingebettet in begleitende Angaben: Oben ein Prototyp, bei dem man alles schon einmal schnell macht. Darunter dann Dokumente (gelb) und Prüfungen (Kreise, orange).

Ein System setzt sich aus Einheiten und Beziehungen zusammen. Wer systematisch arbeitet, bedenkt diese Teile und ihre Abhängigkeiten. Die Vorgehensweise berücksichtigt, wie sie zusammenhängen. Damit mehrere Personen systematisch arbeiten können, muss die Vorgehensweise vorgegeben sein und von Neuen erlernt werden können. Wie das im Einzelnen geschieht, kann sich aber stark unterscheiden.

Bei diesem Prinzip wird als ***traditionelle Vorgehensweise das V-Modell*** vorgestellt. Darin zeigt sich das System, möglichst alle Abläufe und Dokumente vorzudenken und vorzuschreiben, damit man sie einfach nur befolgen muss – ohne viel zu denken. Nach dieser Denkweise ist es gut, viel zu dokumentieren. In den 1990er-Jahren hat man versucht, Prozesse und ihre Erzeugnisse genau zu beschreiben. In sogenannten Reifegradmodellen konnte man feststellen, wie gut die wichtigsten Aspekte der Softwareentwicklung in einem Unternehmen beschrieben waren, und ob sie auch befolgt wurden. Sehr viele Unternehmen haben Jahre und große Summen investiert, um eine bessere Reifegradstufe zu erreichen.

Agile Ansätze verwenden ein ganz anderes System: Hier wird Dokumentation skeptisch gesehen, vielmehr setzt man auf direkte Kommunikation. Trotzdem sind die agilen Methoden in Büchern gut beschrieben, so dass neue Mitarbeiter sie erlernen können und sich nicht alles selbst überlegen müssen. Bei agilen Methoden spricht man weniger von Prozessen als vielmehr von Praktiken, denen zu folgen ist.

Manche agilen Praktiken widersprechen den traditionellen Vorgaben: Wie viel dokumentiert werden soll und in welcher Form, sehen die beiden Systeme ganz unterschiedlich.

Der Rote Faden durch die Folien

Referenz: SWT-23.24-Teil-1-v30. In den Anfängen der Softwareentwicklung haben Programmierer manchmal einfach losprogrammiert und dann erst nachgesehen, wie das Resultat dem Kunden gefiel. Das war nicht sehr zielführend und gilt heute als Inbegriff unsystematischer Entwicklung. Man sollte schon zuerst die Anforderungen erheben, zumindest zu einem kleinen Teil. Auf den Folien 45ff sieht man dann, wie sich ein einfacher Prozess aus fünf verschiedenen Aktivitäten zusammensetzt. Einige Dokumente kommen typischerweise in solchen Prozessen vor: F35 ist ein gutes Beispiel für einen normalen Entwicklungsprozess. Darin kommen die Aktivitäten und Dokumente vor, die man auf jeden Fall kennen sollte.

Die Prozesse haben sich weiterentwickelt und schließlich zum V-Modell F36f geführt hat. Das vollständige V-Modell ist etwa fünf Mal so groß wie auf dieser Folie und schreibt eine sehr große Zahl von Tätigkeiten, Dokumenten und Prüfungen vor. So viele Details ergeben viel Sicherheit, ein so umfangreicher Prozess ist aber auch sehr aufwändig und langsam. Sie werden das V-Modell oft in der symbolischen Darstellung von F38 sehen. Hier sind die meisten Details ausgeblendet, man sieht nur die wichtigsten Aktivitäten und Prüfungen. Der gelbe Teil bezieht sich auf die Software, zu der oft noch Hardware (weißer Teil) hinzukommt. Beispielsweise ist ein Auto das System aus Software, Elektronik und Hardware, für das dann Software entwickelt wird (mit Bremsassistent und vielen weiteren Funktionen).

Das Capability Maturity Model (CMM) ab F40 zeigt am klarsten die Grundidee, durch Reifegrade festzustellen, wie gut eine Organisation dem System der dokumentierten Prozesse folgt. Dabei ist es egal, *welcher* Prozess dokumentiert wird, solange nur gewisse Vorgaben erfüllt sind. F41 zeigt, welche Themen eine Organisation mit ihren dokumentierten Prozessen abdecken muss, um von Ebene 1 auf 2 zu kommen. Dabei müssen *alle* Themen abgedeckt sein. Wenn das gelingt, erreicht man die Vorteile, die auf F46 neben der jeweiligen Stufe stehen. Die anderen Themen sind auf F41 genannt. Man sieht hier auch, wie durch reifere Prozesse der Schätzwert für Dauer und Kosten eines Projekts zunächst steigt. Reifere Organisationen können also realistischer schätzen. Bei weiter steigender Reife verbessern sich die Prozesse und auch die Schätzungen kommen dem Ist-Wert immer näher. Das wollte man erreichen: Gut schätzen und planen können, weil Software so teuer und wichtig geworden ist.

Ab F45 finden Sie die ***agile Umsetzung des Prinzips „systematisch arbeiten“***. Das Ziel ist ähnlich wie beim V-Modell, aber die Herangehensweise ist grundverschieden. Um zu verstehen, wie agile Methoden und ihr System funktionieren, müssen Sie die agilen Werte, Prinzipien (agilen Prinzipien) und Praktiken kennen, die das System ausmachen. Wer agil arbeiten will, wird das lernen, um nicht ad hoc alles selbst erfinden zu müssen. Das bedeutet: Systematisch agil arbeiten.

Von F47 an sehen sie den derzeit bekanntesten Vertreter agiler Methoden: SCRUM (rechts, gelb). Auch diese Technik wird kurz im Zusammenhang dargestellt, die Details kommen nachher bei den Prinzipien noch viel ausführlicher zur Sprache. SCRUM wird gerne mit zwei verschränkten Zyklen dargestellt: Alle 30 Tage beginnt ein neuer Sprint. Innerhalb eines Sprints soll es keine Störungen oder Änderungen von außen geben. An jedem der 30 Arbeitstage gibt es jeden Tag ein SCRUM-Meeting oder Daily SCRUM, oder manchmal auch einfach Daily genannt. F49 zeigt das noch einmal auf andere Weise und betont in dieser Darstellung, wie sich die verschiedenen Rollen einbringen: Der Product Onwer (PO) vertritt den Kunden und muss sich ggf. mit anderen Kunden im Unternehmen abstimmen. Seine Hauptaufgabe ist, das Product Backlog stets so zu sortieren, dass die nützlichsten Aufgaben oben liegen. Wenn dann das SCRUM-Team, im Stil eines Sportler-Teams, alle 30 Tage einen neuen Sprint beginnt, holen sie sich den oberen Teil des Backlog-Stapels: das müssten die nützlichsten Aufgaben sein. In den 30 Tagen arbeiten sie an der Umsetzung und erzeugen ein Inkrement. SCRUM sagt nicht, wie diese Arbeit

organisiert wird; dafür verwenden aber viele Unternehmen XP. Das wird unter dem Prinzip „systematisch arbeiten“ vorgestellt. Die anderen Personen auf dieser Folie dürfen beim Daily SCRUM zuhören, aber nicht sprechen. Darauf achtet der SCRUM-Master, dessen Rolle einem beschützenden Ritter gleicht, der sein Team vor äußeren Störungen beschützt.

Beachten Sie dabei: Beim inkrementellen Entwicklungsmodell F53 geht man immer in gleichen Zeitschritten vor (time boxing), also z.B. sechs oder zwei Wochen. Dann kommt ein Funktionsblock heraus, der nicht nur ein Prototyp ist, sondern produktiv nutzbar F54. Im nächsten Schritt entsteht der nächste Block und kann mit dem ersten zusammen betrieben werden. Ältere Blöcke kann man ändern. Das Wichtigste ist aber, dass man immer die nützlichsten Funktionen zuerst umsetzt: Man hat sie ja auch zuoberst vom Product Backlog geholt. Dann hat man sogar bei einem Projektabbruch schon etwas Nützliches erstellt.

Heute ist SCRUM die bekannteste agile Technik, früher war es XP. Es ist eine gute Idee, die beiden Techniken miteinander zu kombinieren. Da SCRUM nicht vorschreibt, wie während des Sprints entwickelt wird, verwendet man dafür XP. Auch XP liefert wieder Praktiken, an die man sich halten muss. Das ist die Systematik der agilen Vorgehensweise. Sie wird jetzt detailliert vorgestellt.

Sehr wichtig sind die User Stories F60: Man kann sich vorstellen, dass der Product Backlog ein Stapel solcher Aufgabenzettel ist. Man kann völlig frei formulieren, was die Aufgabe ausmacht, was also zu implementieren ist. Das wird auf einen DIN A4- oder sogar A5-Zettel notiert. Zur Formulierung hat sich bewährt, den gewünschten Ablauf in Schritte zu zerlegen und aufzuschreiben, was der Reihe nach geschehen soll. Viele Unternehmen verwenden stattdessen Formulare für die kurzen User Stories, damit man die wichtigsten Aspekte (Rolle des Kunden, zu implementierende Funktion und Nutzen) nicht vergisst. Auf F68ff werden die Praktiken von SCRUM noch einmal explizit genannt.

Es gibt noch viel mehr Praktiken, einige davon sind ab F71 dargestellt. Hier ist wichtig, sich klarzumachen, dass ein Stapel von User Stories eine leichtgewichtige Art von Dokumentation darstellt als die Spezifikation und die Dokumente im V-Modell.

F66: Alle agilen Methoden sind nach diesem Muster aufgebaut: Sie haben die gemeinsamen Grundwerte des Agile Manifesto. Sie folgen leicht unterschiedlichen (agilen) Prinzipien, z.B. Kundenzufriedenheit durch lauffähige Software. Und sie setzten sich aus Praktiken zusammen, die man stets beachten muss. Sie ähneln den Zahnrädern eines Uhrwerks, die stets alle in Bewegung sind (F71). F77 zeigt eine Originalabbildung von Kent Beck, mit der er zeigt, wie alle Praktiken „ein System bilden“, voneinander abhängig sind und sich gegenseitig stützen.

Prinzip 2: Anforderungen wirksam berücksichtigen

Einführung und Motivation

Was soll ein Programm eigentlich leisten? Während die ersten Projekte noch nach Trial-and-Error oder Code-and-Fix entstanden waren, wurde sehr bald deutlich, dass der Erfolg oder Misserfolg eines Projekts zu einem großen Teil davon abhängt, ob man die Aufgabe richtig verstanden hat.

Man geht davon aus, dass gewissen „Kunden“, die die Software bestellen, bezahlen und nutzen, die Referenz dafür sind, ob die Software gut oder schlecht ist. Was sie wollen und brauchen, wird in „Anforderungen“ formuliert. Wie das genau passiert, und ob alle Kunden wirklich wissen, was sie wollen, ist eine andere Frage. Die Umsetzung dieses Prinzips in traditioneller und in agiler Weise unterscheidet sich sehr stark, und doch sind beide Sichtweisen nützlich. Wer professionelle Softwareentwicklung betreibt, muss beide Varianten kennen.

Auf konventioneller Seite ist besonders die Teildisziplin des Requirements Engineering zu nennen. Das sind Aktivitäten, die zur systematischen Sammlung, Interpretation und Prüfung von Anforderungen dienen – und zu ihrer Dokumentation. Als Ergebnis entsteht eine Spezifikation. Im Verlauf des Projekts kann sich noch etwas an den Anforderungen ändern. Das wird dann im Requirements Engineering verwaltet und kostet die Kunden normalerweise Geld: Alles, was sich nach Abschluss der Spezifikation noch ändert, wird wie ein eigenes kleines Projekt behandelt und eben auch eigens bezahlt.

Use Cases beschreiben Abläufe, die mit Hilfe eines Programms durchgeführt werden sollen. Das ist ein wesentlicher Aspekt einer Spezifikation. Diese Technik ist in der Praxis der Softwareentwicklung heute fast überall verbreitet.

In der ersten Teamaufgabe soll eine kurze, aber vollständige Spezifikation geschrieben werden. Darin tauchen auch Use Cases auf.

Im agilen System geht man mit Anforderungen ganz anders um. Sie müssen nicht vor allem aufgeschrieben und dokumentiert werden; vielmehr geht es darum, die Kunden eng einzubinden, damit jeder Wunsch und jede Anforderungsänderung sofort bemerkt und berücksichtigt werden kann. Es gibt einige zusammengehörige Praktiken, die zusammen dafür sorgen, dass man Anforderungen wirksam erheben und auch noch nachträglich ändern kann.

Der Rote Faden durch die Folien

Referenz: SWT-23.24-Teil-1-v30. F87 zeigt einen Überblick über die Aktivitäten, die im Requirements Engineering anfallen. Nach der Systmanalyse ist die Spezifikation vor und wird in der Regel vom Kunden unterschrieben. Bis dahin sind Änderungen noch Teil der Anforderungserhebung. Danach sind sie Änderungsaufträge und müssen bezahlt werden. In den darauffolgenden Folien werden alle diese Aktivitäten durchgegangen. Zuerst wird grob erklärt, was im Rahmen der Aktivität getan werden muss. Dann zeigt das Beispiel eines Bankautomaten, wie diese Aktivität angewendet wird.

Dokumentieren ist besonders wichtig im traditionellen Ansatz, und das ist auch bei den Anforderungen so. Es ist daher wichtig, was in einer guten Spezifikation stehen soll. F94 zeigt eine Klassifikation von Anforderungen. Das ist wichtig und nicht ganz einfach und wird daher an einem längeren Beispiel gezeigt. F97 ist ein typisches Inhaltsverzeichnis für eine Spezifikation. Das Schema heißt „Volere-Template“ und wird gerne verwendet, um Spezifikationen zu strukturieren. In diesem Template gibt es für alle Arten von Anforderungen nach F97 einen Platz.

Validierung und Verifikation dienen beide der Prüfung, aber es werden unterschiedliche Dinge geprüft. Wenn im Verlauf der Entwicklung ein Dokument auf dem anderen aufbaut (z.B. der Entwurf soll die Spezifikation umsetzen), dann stellt man mit der Verifikation fest, ob alle Aspekte des ersten Dokuments im zweiten berücksichtigt sind. Bei der Validierung geht es nicht um Dokumente, sondern um die Frage, ob zu dem entsprechenden Zeitpunkt noch der Wunsch des Kunden getroffen wird. Wenn der Kunde seine Meinung ändert, würde die Validierung eine Abweichung bemerken.

Die Idee hinter Use Cases F103ff ist einfach: Man schreibt sich Abläufe auf, in denen das System und irgendwelche Akteure miteinander interagieren sollen. Die Anforderung, die dahintersteckt, heißt: „Das System soll so gebaut sein, dass man diesen Ablauf damit durchführen kann und es so funktioniert wie im Use Case beschrieben“. Die Definition von F104 für einen Use Case sollte man auswendig kennen, sonst vergisst man gerne den zweiten Teil: Ein Use Case ist nicht nur die Beschreibung einer Interaktion, sondern *mit dem System soll auch ein Ziel erreicht werden*. Man darf beim Use-Case-Schreiben das Ziel nicht aus dem Auge verlieren.

Use Cases ziehen sich durch das gesamte (traditionelle) Projekt: Sie dienen als Anforderungen, zum Testen und für den Entwurf. Auf F107 ist schon eine Vorschau auf die Entwurfssprache UML zu sehen: Use Case-Diagramme sind graphische Modelle, auf denen alle Use Cases eines Systems als Ovale zu sehen sind. Es gibt auch einige Beziehungen zwischen den oval gezeichneten Use Cases. Rund um das Rechteck, das die Systemgrenze darstellt, gibt es die Akteure: Sie brauchen nicht entwickelt zu werden, sondern gehören zum Umfeld des Systems, die mit ihm interagieren. Personen und Gruppen zeichnet man als Strichfiguren, Geräte eher als Kästen.

Auch für Use Cases gibt es wieder Vorlagen, wie im traditionellen Ansatz üblich: Man soll ja daran erinnert werden, woran man denken soll. Wenn alle Einträge in die Tabelle ernsthaft vorgenommen werden, erreicht man ein gutes Verständnis des entsprechenden Anwendungsfalls (Use Case). Auf F113ff sieht man die Vorlage in Aktion: Es werden mehrere Use Cases gezeigt, die zum Bankautomaten gehören. Daran sieht man, wie mit dem Use-Case-Template umzugehen ist. Das wirkt auf den ersten Blick sehr einfach. Man stellt jedoch rasch fest, dass mit jedem Eintrag in die Tabelle auch eine Festlegung (Anforderung) verbunden ist. Indem man Use Cases schreibt, formuliert man Anforderungen in einer kompakten Form.

Die Teamaufgabe soll Sie dazu ermutigen, einmal selbst eine kurze Spezifikation zu schreiben und darin auch einige Use Cases zu verfassen – mit allen Aspekten, die dazu gehören. Die Erfahrung, die man dabei gewinnt, ist das Wichtigste an dieser Teamaufgabe. Um die Spezifikation ausfüllen zu können,

muss man die Anforderungen zuerst erheben. Das geschieht hier auf besondere Weise, weil nicht jede und jeder ein längeres Interview führen kann. Eine typische erste Kundenbefragung wurde daher auf Video aufgenommen und ist der Ausgangspunkte für die restlichen Aktivitäten des Requirements Engineering. Am Ende sollte die Spezifikation einen guten Überblick über das System bieten, das gebaut werden soll. F124 ist das Inhaltsverzeichnis des Template, das übrigens aus dem Softwareprojekt (SWP, Lehrveranstaltung im 5. Semester) stammt und dort auch verwendet wird. Indem Sie also die Teamaufgabe bearbeiten, bereiten Sie sich also schon ganz praktisch auf das Softwareprojekt vor. Und gleichzeitig lernen Sie, mit Spezifikationen umzugehen. Das brauchen Sie sicher später in der Praxis.

Man muss allerdings feststellen, dass die Spezifikation und alles, was damit zusammenhängt, charakteristisch für den traditionellen Ansatz ist. Es wird darin klar vorgeschrieben, was die Software tun soll. Viel wird dokumentiert und vorstrukturiert durch Templates.

Die agilen Ansätze gehen hier radikal anders vor, obwohl auch hier das Ziel ist, Anforderungen wirksam zu berücksichtigen. F128 zeigt ganz knapp, wie die Anforderungen in XP erhoben werden: Von allen Kunden in einem Unternehmen wird einer als On-Site Customer ausgewählt. Diese Person ist mehr oder weniger ständig bei den Entwicklern, sie können ihn stets fragen und er sagt ihnen, was sie programmieren sollen. Das sind nicht-dokumentierte Anforderungen. Außerdem nehmen diese Kunden auch am Planning Game teil. Es findet in SCRUM vor jedem Sprint statt, oder in XP nach einer Iteration. Darin laufen drei wichtige Schritte ab: 1. Schreiben die Kunden mit Hilfe der Entwickler User Stories auf. 2. Dann bewerten die Entwickler, wie viel Aufwand die User Stories im Vergleich untereinander machen. 3. Schließlich sortiert der Kunde die bewerteten User Stories nach ihrem Nutzen, ähnlich wie beim Product Backlog. Jetzt können sich die Entwickler die obersten herunternehmen und werden damit nützliche Aufgaben umsetzen. Nach der Iteration haben sie ein neues Inkrement erzeugt. Es geht an die Kunden, die es schon einmal einsetzen können. Inzwischen geht das Ganze in die nächste Runde.

Es gibt viele Werkzeuge für die Softwareentwicklung JIRA ist ein Vertreter für eine Gruppe von Werkzeugen, mit denen man Aufgaben priorisieren und verwalten kann. Die Aufgaben stehen auf User Stories, die im System angelegt und verschoben werden können. Man kann die Übersichten wie auf F132 unterschiedlich aufbauen und zusammen mit dem On-Site Customer immer wieder neu entscheiden, was wichtiger ist und nach oben kommt und welche Themen man überhaupt angeht.

Die letzten Folien zu diesem Prinzip gehen auf die direkte Kommunikation ein, die in agiler Anforderungserhebung so sehr betont wird: Reden geht schneller als Schreiben und Lesen, und die agilen Verfahren halten die Dokumentation für überschätzt. Mit den FLOW-Modellen F139f kann man den Fluss von Anforderungen und anderen Informationen in einem Softwareprojekt erfassen und aufzeichnen (modellieren). Dann kann man sich überlegen, ob und wo man darin etwas verbessern kann. F141 vergleicht die beiden Varianten. Etwas verblüffend an dieser Zusammenfassung ist vielleicht, dass der agile Ansatz wesentlich mehr Disziplin von den Entwicklern fordert als der traditionelle. Aber das stimmt schon: Beim traditionellen Ansatz ist ja alles im Detail aufgeschrieben und festgelegt. Beim agilen Ansatz verlangt und erwartet man, dass alle Entwickler sich an das System der Praktiken halten, damit sie wie ein Uhrwerk funktionieren können.

Prinzip 3: Struktur entwerfen mit Information Hiding

Einführung und Motivation

Wer erste kleine Programme schreibt, tippt sie oft direkt in den Rechner. Alles, was zwischen der Aufgabe und dem Programm geschieht, läuft im Kopf ab. In professioneller Softwareentwicklung sind schon die Komponenten viel zu groß, um sich alles im Kopf merken zu können; auch braucht man die Kompetenz mehrerer Entwickler, um die beste von vielen Möglichkeiten zu finden. Dazu muss man über den Aufbau des Programms diskutieren können – man muss ihn visualisieren. Und darf sich nicht von Details ablenken lassen.

Es gibt viele Teile an einem Programm, die sich ändern können, ohne die Struktur zu berühren. Das sind Bezeichner, Einrückungen und viele weitere Aspekte, die zwar für die Lesbarkeit eines Programms sehr wichtig sind. Aber die Struktur meint die Aufteilung in Module und Komponenten, die Beschreibung von Schnittstellen. Die Struktur ist oft sehr beständig, auch wenn sich andere Teile ändern und sogar einzelne Module ausgetauscht werden. Daher ist es wichtig, sich über die Struktur Gedanken zu machen, sie darzustellen und zu diskutieren. Erst danach fängt man an, die Aufgaben zu verteilen und schließlich auch zu programmieren.

Es gibt eine Reihe allgemeiner Regeln, wie ein guter Entwurf aussehen soll. Das sind aber eher Daumenregeln. Zwei Konzepte, Kohäsion und Kopplung, sind davon noch recht konkret. Sie besagen, dass alles, was man in eine Einheit (Modul, Klasse, Methode usw.) zusammensteckt, viel miteinander zu tun haben sollte (z.B. sich gegenseitig aufrufen); dagegen sollte es wenig mit den Inhalten *anderer* Einheiten zu tun haben. So reduziert man den Abstimmungsbedarf zwischen den Entwicklern.

Die Unified Modeling Language UML für die grafische Darstellung von Software-Entwürfen muss jede und jeder Informatiker/in kennen. Dann kann man in verschiedenen Teams, über unterschiedliche Unternehmen und Länder hinweg nahtlos zusammenarbeiten. Man versteht sofort, was ein UML-Diagramm bedeuten soll, das andere gezeichnet haben.

Information Hiding ist ein Konzept, das sich überall in der Softwareentwicklung wiederfindet. Es besagt: Wenn ich mir überlegt habe, wie ein Modul oder Programmteile funktioniert, muss sich das nicht jede/r wieder überlegen, der es verwenden will. Es ist besser, wenn die Verwendenden nicht wissen müssen, wie mein Modul genau arbeitet. Dann können sie sich um ihre Aufgaben kümmern. Daher will man in der Regel nicht wissen, welche Entwurfsideen und -entscheidungen jemand im Inneren einer Komponente umgesetzt hat, solange man weiß, wie man die Komponente *verwendet*. Man hat sich daher Mechanismen überlegt, um andere nicht mit unwichtigen Details zu belasten. Man versteckt die Details praktisch, betreibt „Information Hiding“. Nicht, weil man etwas geheim halten will. Sondern aus Entgegenkommen für die Verwendenden: Sie müssen sich nicht mit meinem Detailwissen belasten. Durch Information Hiding und die Definition klarer Schnittstellen können beide Seiten, Entwickler und Verwender, ihre Teile unabhängig entwickeln und ändern.

Eine der neuesten Durchbrüche der Informatik und des Software Engineering sind die Design Patterns. Ein Design Pattern wird oft in UML dargestellt und beschreibt, wie man ein oft auftretendes Entwurfsproblem am besten löst. Meist sind es scheinbar einfache Probleme, zu denen man schon auch selbst eine Lösung fände; aber aus Erfahrung haben gute Entwickler gelernt, dass man sich damit später Schwierigkeiten einhandeln kann. Daher haben sie diese Erfahrungen zusammengefasst. Design Patterns helfen ganz direkt und konkret in praktischer Softwareentwicklung. Sie sind meist in UML notiert, berücksichtigen Information Hiding und führen zu flexiblen Entwürfen von hoher Qualität.

Der Rote Faden durch die Folien

Referenz: SWT-23.24-Teil-2-v37. Allgemeine Entwurfsregeln wie auf F21ff sind zwar nicht sehr spezifisch, man weiß also nicht unbedingt gleich, wie man die Regel befolgen kann. Aber dafür gelten sie für einen weiten Bereich von Strukturierungen: Wie gliedert man ein Package in Java, wie verteilt man Methoden auf Klassen und vieles mehr.

Man könnte denken, der Entwurf einer guten Softwarestruktur sei vor allem ein technisches Problem. Das ist es auch zum Teil, denn die Struktur und der Entwurf sind die Voraussetzung für reibungslosen und schnellen Ablauf. Gerade deswegen ist es aber entscheidend, dass man verschiedene Entwurfsideen vergleicht. Mit Programmcode geht das nicht: Der ist viel zu detailliert, als Text hat er keine sichtbare Grobstruktur. Wenn man Entwurfsmodelle als ein Mittel zur Kommunikation zwischen Architekten und Entwicklern betrachtet, dann wird deutlich, dass ein guter Entwurf für die Kommunikation zwischen Menschen verständlich sein muss.

Entwurf bedeutet, eine Software zu gliedern, die es noch gar nicht gibt F14f. Die Vorstellungen im Kopf müssen aufs Papier, um mit anderen besprochen zu werden. Dabei ist besonders strukturbildend die Frage, wie verschiedene Teile zusammenhängen. Mit Schnittstellen beschreibt man, was verwendende Einheiten von verwendeten verlangen können und wie das geht. Dabei zeigt F17 die eher technische Seite. In einer Aufrufschnittstelle stehen alle Angaben, die eine aufrufende Einheit übergeben muss, um mit der aufgerufenen etwas zu tun. Dagegen zeigt F18 eine API, mit der auch menschliche Entwerfer etwas anfangen können. Später wird noch gezeigt, wie man solche APIs in schöner Formatierung aus JavaDoc-Kommentaren generieren kann. Damit sind dann Programme und Schnittstellenbeschreibungen eng beieinander.

Conway hat herausgefunden, dass ein Softwaresystem die Tendenz hat, genau so aufgebaut zu werden wie das Entwicklungsteam. Das hat verschiedene Konsequenzen: Man kann sich denken, wie viele große Komponenten ein Programm haben wird (so viele, wie Teilteams), und bei verteilter Entwicklung hat man dadurch die Schnittstellen praktisch gleich mitdefiniert. Das heißt umgekehrt: Wenn man von seiner Software gewisse Entwurfseigenschaften erwartet, sollte man auch das Entwicklerteam so strukturieren.

Kohäsion und Kopplung sind Konzepte, die man schon lange kennt. Sie besagen, wie die Software beschaffen sein soll. Um daraus konkrete Handlungsanleitungen abzuleiten, muss man aber noch nachdenken. Dagegen ist die Schichten-Architektur F27 sehr konkret und wird in den meisten Softwareprojekten befolgt. Ob es dann zwei, drei oder vier Schichten sind, kommt auf die Anwendung an. Im Zweifel ist die 3-Schichten-Architektur die häufigste. Die allgemeinen Entwurfsprinzipien sind nicht typisch nur für die traditionelle oder nur für die agile Entwicklung: Sie gelten vielmehr allgemein und in jedem Umfeld.

In agilen Projekten kommt noch eine Anforderung an den Entwurf hinzu: Er soll möglichst einfach sein F30. Es ist aber paradoxerweise gar nicht einfach, einen Entwurf einfach zu machen – also so, dass man ihn leicht verstehen und weiterentwickeln kann. Verständlichkeit ist für die agile Entwicklung kaum zu überschätzen: Wenn ein Programm zu kompliziert ist, kann man es nicht mehr gut ändern. Das behindert agile Entwicklung massiv, die ja darauf aufbaut, alles jederzeit ändern zu können. Wie bei den allgemeinen Entwurfsprinzipien kann man sich überlegen, was zur Einfachheit beiträgt. Es bleibt schwierig, Software einfach zu machen. Die agile Praktik des Einfachen Entwurfs ist eng verbunden mit der Kontinuierlichen Integration F33 und Refactoring F35. Damit man ständig testen kann, ob der einfache Entwurf dennoch ausreicht, um das wünschenswerte Ergebnis zu erzielen, muss die Software ständig integriert werden. Unter „kontinuierlich“ verstand man früher: täglich. Dann erreichten manche Projekte mehrere Integrationen pro Tag. Heute gibt es Internet-Software, die jede Sekunde

mehrfach integriert wird. Auf diese Häufigkeit kommt es in dem Prinzip gar nicht an. Wichtig ist nur, dass die Software sehr häufig aktualisiert und testbereit gemacht wird, damit nicht Fehler und Veraltetes lange unbemerkt darin bleiben. Refactoring ist der Umbau von Software, wobei sich deren Funktion nicht ändert. Der einzige Zweck des Refactorings ist die Vereinfachung von Code, der durch Sonderfälle und Patches sich vom Ideal des Einfachen Entwurfs entfernt hat. Da Kunden von sich aus nicht fordern werden, etwas zu tun, bei dem sich die Funktion nicht weiterentwickelt, müssen Refactorings regelmäßig als Teil der normalen Story Cards ausgeführt werden. Manche Autoren fordern, nach jeder fertigen Story Card zu überlegen, ob ein Refactoring nötig ist.

Unified Modeling Language (UML)

Von den 13 UML-Diagrammarten interessieren uns in SWT nur einige wenige F44. Das ist zunächst das Klassendiagramm, weil es als einziges die Statische Sicht auf alle beteiligten Klassen und Assoziationen bietet. In keinem Diagramm darf eine Klasse oder ihr Objekt vorkommen, die nicht im Klassendiagramm eingeführt worden sind. Um das dynamische Verhalten von Objekten zu beschreiben, dienen Sequenz- und Kommunikationsdiagramme. In beiden wird auf unterschiedliche Weise (aber mit derselben Ausdrucksmächtigkeit) modelliert, wie die Objekte sich gegenseitig Nachrichten schicken und darauf reagieren. Aus optischen Gründen sind Sequenzdiagramme eher für komplizierte Abfolgen zwischen wenigen Objekten geeignet, während Kommunikationsdiagramme bei vielen, einfachen Interaktionen und Objekten zu wählen sind.

Weiterhin sind Aktivitätsdiagramme nützlich und daher hier relevant. Darauf sieht man, wie mehrere Objekte an einer Aufgabe arbeiten. Häufig sind die Objekte sogenannten Swimlanes zugeordnet; das sind Spalten, in denen alle Aktivitäten des zugehörigen Objekts angeordnet sind. Übernimmt ein anderes Objekt, so sind dessen Aktivitäten in seiner Spalte angeordnet. Dagegen bezieht sich ein Zustandsdiagramm auf den inneren Zustand je eines Objekts. Man sieht zwar, wie von außen Nachrichten eintreffen und dann eine Zustandsänderung bewirken können. Aber ein Zustandsdiagramm zeigt nur die Perspektive eines Objekts.

Use Case-Diagramme sind keine Use Cases. Sie sind vielmehr grafische Übersichten darüber, welche Use Cases zu einem System gehören. Use Case-Diagramme sind eine UML-Diagrammnotation, dagegen sind Use Cases textliche Darstellungen in Tabellen; das ist kein UML. Die Verbindung zwischen Use Cases und dem Use Case-Diagramm ist lediglich, dass für jeden Use Case dessen Name im Use Case-Diagramm auftauchen muss.

Auf F47ff sieht man unterschiedlich detaillierte Ausschnitte aus einem Klassendiagramm. Alle sind korrekt, aber sie transportieren unterschiedliche viele Informationen. Auf F49 sieht man, wie aus einer Klasse in zwei verschiedene Programmiersprachen ein Coderahmen generiert werden kann. Im Code stehen natürlich nur Angaben, die auch im Modell zu sehen waren, nun aber als Code formatiert.

F50ff zeigt die oben erwähnten Diagrammarten im Beispiel, ebenfalls in der Ausführlichkeit, wie hier die Diagramme zu verwenden sind. Beachten Sie, wie auf F58 noch einmal die zuvor gezeigten Diagramme ausgewertet und ins Klassendiagram übernommen wurden: Jetzt gibt es auch hier eine Prüfbehörde (war in einem Sequenzdiagramm aufgetaucht). Wie hier gezeigt, wird man immer wieder prüfen und sicherstellen, dass alles im Klassendiagramm aufgeführt wird, was man zwischendurch in einem anderen Diagramm verwendet hatte. F60 erinnert noch einmal daran, dass dann am Schluss das eine Klassendiagramm mit allen zugehörigen anderen Diagrammen zusammen das UML-Modell bildet. Es ist Vorlage für die Programmierung oder auch die Generierung von Code-Teilen.

Prinzip 4: Verständlich Programmieren

Einführung und Motivation

Software Engineering soll die Entwicklung, Wartung und Weiterentwicklung von Programmen systematisch unterstützten. Dazu gehört, Anforderungen richtig zu erfassen und zu berücksichtigen (Prinzip 1) und generell eine systematische Arbeitsweise, die man als neuer Mitarbeiter verstehen und erlernen kann (Prinzip 2); man kann nicht einfach drauflosarbeiten, sondern muss bewährten Vorgehensweisen folgen. Neben dem Umgang mit Anforderungen ist die Tätigkeit des Entwerfens und Strukturierens von großer Bedeutung: Hier werden die Voraussetzungen für ein korrekt und gut funktionierendes Programm gelegt. Bis dahin wird noch nicht programmiert.

Das vierte Prinzip beschäftigt sich damit, was man konkret beim Programmieren berücksichtigen sollte, um ein möglichst verständliches Programm zu erhalten. Die Struktur in Pakete, Klassen und Methoden ist also schon vorgeplant (entworfen), und jetzt wird wirklich Programmcode geschrieben. Nun kann man sich natürlich fragen, wieso es gerade die *Verständlichkeit* ist, die beim Programmieren so wichtig sein soll. Ein Programm soll ja auch effizient und performant sein, es soll Angriffen widerstehen (security) und intuitiv bedienbar sein. In der Vorlesung „Software-Qualität“ werden diese und weitere Qualitätsaspekte besprochen und vertieft. Sie sind alle wichtig, je nach Anwendung unterschiedlich wichtig. Aber Verständlichkeit ist die Grundlage praktisch aller Qualitätsaspekte, und Verständlichkeit ist Voraussetzung dafür, Software ändern, warten und weiterentwickeln zu können. Nur Programme, die man versteht, kann man weiterentwickeln, ohne unabsichtlich Fehler einzubauen. Das trifft für jedes Projekt zu, und daher gibt es hier auch keine größeren Unterschiede zwischen agilen und traditionellen Ansätzen. In jedem Projekt muss verständlich programmiert werden.

Unter dieses Prinzip fällt die Frage, wie Bezeichner, Einrückungen und Kommentare gestaltet sein sollen, um das Programm für Entwickler verständlich zu machen. Auch nach mehreren Jahren helfen solche Details, um zu verstehen, was ein Programmteil tut – und wieso das auf eine bestimmte Weise gelöst worden ist. Dann weiß man, wo man anpacken muss und was man nicht verändern darf, um nicht andere Funktionen zu gefährden. In großen Softwareprojekten ändert sich ständig die Belegschaft; schon nach kurzer Zeit müssen neue Kollegen Software ändern, die sie nicht mitgeschrieben haben. Neben den Richtlinien für die Gestaltung von Programmen gibt es weitere Mechanismen, die die Zusammenarbeit zwischen vielen Entwicklern maßgeblich unterstützen und dazu beitragen, dass sie sich nicht gegenseitig stören und dabei unverständliche Programme hinterlassen. Dazu gehören vor allem auch Versions- und Konfigurationsverwaltungssysteme wie SVN oder Git.

Der Rote Faden durch die Folien

Referenz: SWT-23.24-Teil-2-v37. Bei Folie 117 beginnt das Thema. Die ersten Folien motivieren, wieso es so wichtig ist, dass Menschen Programme verstehen können – und nicht nur Compiler. Ein Hauptgrund besteht darin, dass man sich selbst nicht alles merken kann, was man sich vor einigen Monaten überlegt hat. Umso weniger kann man sich nur auf sein Gedächtnis verlassen, wenn viele Kollegen zusammenarbeiten. Dann muss eine/r den oder die andere verstehen, sonst besteht die Gefahr, bei einer Veränderung mehr kaputt zu machen also zu verbessern.

Das Beispiel auf F122 zeigt besonders schwer leserlichen Programmcode. Natürlich sind die meisten echten Programme nicht *so* unleserlich. Das Beispiel ist auch tatsächlich umgekehrt entwickelt worden: Ausgehend von einem gut lesbaren Programm wurden Schwächen eingebaut, durch die aber die Funktion des Programms nicht geändert wurde. So wurden Bezeichner durch überlange und kryptische Bezeichner ersetzt, die dennoch Java-korrekt sind. Ebenso wurden die Einrückungen gegenüber den üblichen Gepflogenheiten verändert; Kommentare sind entweder entfernt worden oder selbst eher missverständlich. Das Ergebnis ist ein „funktionierendes Programm“, das aber durch nur wenige Maßnahmen für Menschen völlig unverständlich – und damit in der Praxis unbrauchbar – geworden ist. Der Unterschied besteht also in Einrückungen, Bezeichnern und Kommentaren. Die können viel ausmachen. F127 fasst das zusammen und zeigt ein Programmstück in lesbarerer Form.

Vorschläge oder Vorschriften, wie in einer Organisation zu programmieren ist, sind in Code Conventions niedergelegt. Dort sollten grundlegende Festlegungen getroffen werden, die möglichst nachprüfbar sind und nicht so sehr Geschmackssache. Ein spezieller Aspekt der Kommentierung ist die Verwendung von JavaDoc. Im Code unterscheiden sich diese Kommentare (/**) kaum von normalen Entwicklerkommentaren (/*). Aber konzeptionell unterscheiden sich JavaDoc-Kommentare erheblich von normalen Entwicklerkommentaren: Aus Ihnen lassen sich Schnittstellenbeschreibungen für Verwendende (APIs) automatisch generieren. Diese stehen dann als vernetzte HTML-Seiten zur Verfügung und helfen anderen Entwicklern einzuschätzen, ob man diese Klasse oder Methode verwenden möchte. Entsprechende Informationen sollten in JavaDoc-Kommentaren stehen, die stets und nur am Anfang von Klassen oder Methoden stehen dürfen. Wie ein Algorithmus funktioniert, muss man dafür nicht wissen und sollte stattdessen in normalen Entwicklerkommentaren stehen.

Es gibt Aspekte, die man nicht in die Kommentare eines Programms schreiben würde: Weil sie dafür zu umfangreich sind oder auch zu sehr in die Tiefe gehen. Auf F129f ist der Algorithmus skizziert, mit dem man einen Huffmann Code aus einer sortierten Liste von Zeichen erstellen kann. Diesen Algorithmus, die Datenstrukturen und den Beweis, dass dieser Algorithmus tatsächlich zu einem optimalen Huffman-Code führt, würde man in der externen Dokumentation für Entwickler beschreiben.

Der zweite Punkt, der sich für die Kooperation in Teams ergibt, ist die Abstimmung über Änderungen an gemeinsam bearbeiteten Dateien. Oft werden ad hoc-Vereinbarungen getroffen, wenn man gemeinsam eine Datei ändern und weiterentwickeln will. Dann kann es zu Namen wie *XY*, *XY-neu*, *XY-ganzneu* usw. kommen. Solche Namensschemata verhindern kaum, dass verschiedene Personen Doppelarbeit leisten und sich andererseits gegenseitig Teile überschreiben. Um gemeinsame Arbeit an Dokumenten zu organisieren und automatisch zu unterstützen, gibt es Versions- und Konfigurationswerkzeuge wie SVN oder Git. In der Vorlesung werden die Grundlagen beider Systeme kurz gezeigt F152. Man sollte beide Ansätze kennen und vergleichen können. F155ff geht auf Git ein. Es ist etwas komplizierter als SVN, das eher die klassischen Versionsmanagement-Befehle anbietet. Git basiert dagegen auf einem verteilteren Arbeitsmodell, bei dem Änderungen letztlich angeboten werden (pull request), aber im Rahmen es pulls von den Verwaltern eines Systems oder einer Datei geholt werden müssen.

Prinzip 5: Prüfen und Qualität hochhalten

Einführung und Motivation

Oft hört man, dass ein Programm „läuft“. Was bedeutet das? Manchmal nicht mehr, als dass das Programm ohne Compilerfehler übersetzt wird und über mehrere Minuten hinweg kein unerwarteter Programmabbruch erfolgt ist. Das heißt aber noch lange nicht, dass das Programm fehlerfrei arbeitet und außerdem alle geforderten Qualitätsaspekte erfüllt. Es kann sein, dass man Fehler einfach noch nicht gefunden hat und beim Ausprobieren zufällig noch auf keinen Fehler gestoßen ist. Außerdem ist es oft gar nicht einfach, einen Fehler als solchen zu erkennen: Wenn bei einer Berechnung ein Wert ausgegeben wird, ist es dann auch schon der richtige Wert? Dazu müsste man nachrechnen, und man müsste sicherstellen, alle Anforderungen bei der Berechnung korrekt zu berücksichtigen. Beim professionellen Testen würde man nicht nachrechnen, sondern *vor* dem Test ermitteln, was genau herauskommen soll (Soll-Wert). Nur so kann man feststellen, ob das Ergebnis stimmt.

Softwareentwicklung ist schwierig und kompliziert. Dass es dabei immer wieder einmal Missverständnisse und Fehler gibt, ist nicht verwunderlich. Weil man das weiß, muss man Vorkehrungen treffen, um systematisch Fehler zu vermeiden und zu entdecken, bevor man die Software an die Kunden ausliefert. Dabei geht es darum, dass ein Programm richtig rechnet und sich so verhält wie gefordert. Qualitätsanforderungen wie Bedienbarkeit, Effizienz, Portierbarkeit und viele andere gehen darüber aber weit hinaus. Prinzip 5 besagt, dass jede Softwareentwicklung Maßnahmen treffen muss, damit hohe Qualität erreicht wird. Oft erfordert das Prüfungen. Besser ist es aber, wenn man schon bei der Entwicklung selbst, also konstruktiv, darauf achtet.

Es gibt drei große Gruppen von Qualitätssicherungsmaßnahmen, die in der Praxis und in Lehrbüchern unterschieden werden: (1) ***Konstruktive Qualitätssicherung*.** Die Maßnahmen, mit denen man versucht, schon bei der Entwicklung Fehler zu vermeiden und hohe Qualität zu erzeugen. Dazu gehört auch alles, was in den bisherigen Prinzipien 1-4 vorgestellt wurde. Zum Beispiel kann der Einsatz eines Composite-Patterns zum Qualitätsaspekt der Flexibilität beitragen. (2) ***Analytische Qualitätssicherung*** ist am bekanntesten und in Lehrbüchern am besten abgedeckt: Nach erfolgter Entwicklung wird das Resultat getestet und geprüft, wozu es besondere Verfahren gibt. Diese Verfahren sollen mit wenig Aufwand möglichst viele Fehler finden, was eine durchaus herausfordernde Aufgabe ist. Danach müssen die Fehler aber noch beseitigt werden, es reicht nicht, sie nur zu finden. (3) Oft vergessen wird die ***organisatorische Qualitätssicherung***, obwohl sie in der Praxis oft entscheidend ist. Hier geht es darum, dass für alle Prüfungen, Korrekturen und konstruktiven Maßnahmen ein organisatorischer Rahmen geschaffen, Zuständigkeiten und Zeit eingeplant werden muss. Es geht hier darum, dass Entwickler und alle Projektbeteiligten eine Einstellung entwickeln, die sie dazu ermutigt, Qualität hochzuhalten – und dafür auch die erforderlichen Mittel in die Hand zu bekommen.

Agile und traditionelle Ansätze für Software-Qualität unterscheiden sich deutlich. Bei den traditionellen Ansätzen ist der analytische Aspekt dominanter, während in der agilen Entwicklung Qualitätssicherungs- und Entwicklungsaufgaben ganz eng aneinander gebunden sind und sich iterativ wiederholen. Das muss man wissen, um erfolgreich an den jeweiligen Projekten teilnehmen zu können. Um in agilen Projekten gut Qualitätssicherung zu betreiben, muss man auch die traditionellen Ansätze beherrschen. Umgekehrt kann man sich auch in traditionellen Projekten von den Qualitätsmaßnahmen agiler Projekte inspirieren lassen.

Dieses Kapitel ist relativ kurz, weil Software-Qualität in der gleichnamigen Vorlesung im Sommersemester intensiv behandelt wird.

Der Rote Faden durch die Folien

Referenz: SWT-23.24-Teil-2-v37. Auf F160 werden die drei Ansätze der Qualitätssicherung eingeführt: Konstruktiv, analytisch und organisatorisch. Sie werden in traditionellen und agilen Ansätzen alle auftauchen, aber unterschiedlich umgesetzt werden.

Grundlagen des Testens werden ab F162 dargestellt. Die Definition auf F162 muss man auswendig kennen und erklären können. Kurze Beispiele auf F164 führen auf das Problem des Testens hin: Man kann nicht jeden einzelnen möglichen Fall testen F171). Aber welche sollte man auswählen, um dennoch möglichst viele Fehler zu finden? Das etwas längere Beispiel ab F165 verdeutlicht, dass auch scheinbar selbstverständliche Dinge zu hinterfragen sind. Bei finnischen Kassenzetteln ist die Summenbildung viel komplizierter als in Deutschland: Finnland benutzt keine 1c und 2c-Münzen mehr, Barzahlungen müssen also auf- oder abgerundet werden. Daraus ergeben sich viele Fälle, in denen man leicht Fehler machen könnte. Es zeigt sich deutlich, dass man genau auf die Anforderungen achten muss, um (1) beim Programmieren keinem Missverständnis zu unterliegen und (2) auch beim Testen an alle Fälle zu denken. Auch Testfälle können gut oder schlecht geschrieben sein. F169 zeigt Kriterien, wann Testfälle „gut“ sind, also nützlich. Auch sie betonen wieder, dass idealerweise Anforderungen und Testfälle ganz eng zusammengehören.

Aus Sicht der organisatorischen Qualitätssicherung sind Tests wichtig – sie reichen aber nicht aus. Man muss immer auch Zeit und Ressourcen einplanen, um möglicherweise gefundene Fehler auch zu beseitigen. Prüfen alleine verbessert die Qualität noch nicht. In der Forschung konzentrieren sich viele Autoren darauf, Testen zu vereinfachen und Teile davon zu automatisieren. Das hat prinzipiell Grenzen, denn es müssen ja nicht nur die Eingaben, sondern auch die korrekten Ausgaben generiert werden, was nur in Ausnahmefällen möglich ist. Dagegen kann man sich einige Daumenregeln merken, um das Testen wirksam und doch effizient zu machen: Man sollte nicht viele ähnliche Fälle testen, sondern alle verschiedenen Fälle mindestens ein Mal. F175 zeigt: Das Testen sollte von kleinen Einheiten (Modulen, Units) beginnen und erst dann zu größeren übergehen, wenn in den kleinen Einheiten keine Fehler mehr gefunden wurden. Denn würde man beim Gesamtsystem beginnen und Fehlerverhalten beobachten, wüsste man nicht, wo man suchen soll: in Einzelteilen? In der Kombination?

Abnahmetests sind ein wichtiger Sonderfall. Hier möchten die Beteiligten eigentlich keine Fehler mehr finden, eher sollen die Tests zeigen, dass man die zuvor vereinbarten Fälle abgedeckt hat und dort keine Fehler mehr auftreten. Das widerspricht unserer Definition von Testen; da der Sprachgebrauch „Abnahmetest“ aber üblich und verbreitet ist, muss man sich diesen Sonderfall eigens merken. Der Abnahmetest ist für ein Projekt von zentraler Bedeutung, weil sich dabei entscheidet, ob die Leistung erbracht ist und bezahlt werden kann und muss. Daher darf man den Abnahmetest F176 keinesfalls auf die leichte Schulter nehmen.

Agile Qualitätssicherung (ab F178) ist grundsätzlich anders in die Entwicklungstätigkeiten eingebettet. Es gibt keine Trennung, jedes Praktik hat konstruktive und qualitätssichernde Aufgaben zugleich, wie auf F179 dargestellt. Dabei spielt Testen eine sehr wichtige Rolle, denn ohne ständigen und umfangreichen Test könnte man weder Refactoring durchführen noch die vielen Änderungen verantworten. F182 zeigt auf nur einer Folie das Prinzip von Test First: Eigentlich soll keine Programmzeile geschrieben werden, wenn nicht ein Test das erfordert. Immer schreibt man erst einen Test, der für eine Anforderung steht. Dann erst das Programm, das diesen Test erfüllen soll. Das Werkzeug JUnit zeigt anfangs einen roten Balken, dann einen grünen, wenn die Tests alle erfüllt sind. Da Testfälle und Programm

Schritt für Schritt gemeinsam entstehen, sollte es eigentlich keine ungetesteten Programmteile geben können – und keine Tests, zu denen es keine Programme gibt. Mit dem Test-Driven Development wurde eine ganze Entwicklungsmethode um das frühe Testen erstellt.

Qualität zu prüfen und zu gewährleisten ist ein Grundprinzip jedes professionellen Softwareprojekts. Die hier gezeigten Beispiele sollen nur die Grundidee zeigen und Appetit auf mehr machen: Die Vorlesung Software-Qualität im Sommer ist genau darauf ausgerichtet. Sie bietet deutliche Vertiefung beim Testen, zusätzlich Reviews und mehrere andere Ansätze. Usability / Bedienbarkeit wird als eines der Beispiele behandelt.

Prinzip 6: Fortschritt schätzen

Einführung und Motivation

Wie kann man feststellen, ob ein Softwareprojekt gut unterwegs ist oder in Schwierigkeiten? Während eine Brücke während der Bauphase sichtbare Fortschritte macht, ist das in der Softwareentwicklung gar nicht so einfach abzuschätzen: Reicht es aus, die Länge des Programms in Codezeilen zu zählen? Oder die Entwickler zu fragen? Erfahrungsgemäß fallen subjektive Schätzungen („wir sind fast fertig“) oft zu optimistisch aus.

Um vergleichen zu können, wie gut sich ein Projekt entwickelt, müsste man zuvor schätzen, wie lange es vermutlich dauern wird, wie viel Aufwand und Kosten es verursachen dürfte. Dann kann man die aktuell gemessenen Fortschrittsdaten mit den zuvor geschätzten Werten vergleichen.

Unter „predictability“ versteht man die Fähigkeit, Dauer und Kosten eines Projekts abschätzen zu können. Diese Fähigkeit ist von größter Bedeutung für den wirtschaftlichen Erfolg des Projekts. Schätzt ein Softwarehaus die Kosten zu hoch ein, so wird es ein unnötig teures Angebot abgeben und den Auftrag vermutlich nicht bekommen. Häufiger sind allerdings zu optimistische Schätzungen. Das Projekt dauert nachher länger, bindet die Entwickler über längere Zeit und kostet mehr. Da häufig Festpreisangebot abgegeben werden müssen, gehen solche Fehleinschätzungen auf Kosten der Anbieter, also des Softwarehauses. Das kann man sich nicht leisten.

Im Idealfall lässt sich schon recht früh in einem Projekt, sobald die wichtigsten Anforderungen bekannt sind, eine erste Schätzung abgeben, die einigermaßen zuverlässig ist. Auf dieser Basis kann ein realistisches und gleichzeitig attraktives Angebot erstellt werden. Außerdem kann man mit den Schätzdaten fortlaufend vergleichen, ob man gesetzte Zwischenziele wie geplant erreicht.

Software Engineering umfasst nicht nur Entwurf und Programmierung, sondern auch Aspekte der Qualität und der Fortschrittsprüfung. Auf die eine oder andere Weise muss jedes Projekt Aufwand und Fortschritt schätzen; so gibt es in der traditionellen Entwicklung dafür Verfahren wie COCOMO und Function Points, während agile Projekte den noch verbleibenden Aufwand schätzen und regelmäßig nachjustieren. Mit der Technik der Meilensteine kann man in der Planung komplexe Erwartungen beschreiben und bündeln: Man erwartet beispielsweise, dass zum Zeitpunkt t alle Anforderungen erhoben, dokumentiert und vom Kunden unterschrieben sind. Formuliert man diese Erwartung als Meilenstein, so wird zum Zeitpunkt t geprüft, ob alle Kriterien erfüllt sind. Meilensteine sind sehr individuelle, praxisnahe Möglichkeiten, um zu verfolgen, ob der Fortschritt im Projekt sich gut entwickelt. In die Meilensteine können auch Schätzdaten für Teilprodukte integriert werden.

Neben der anfänglichen Schätzung dienen Schätzungen, Meilensteine und Kriterien als Frühwarnsystem: Wenn es merkliche Abweichungen zwischen Schätzung, Plan und tatsächlicher Entwicklung gibt, müssen Projektleitung und Unternehmensleitung reagieren, damit am Ende nicht das ganze Projekt Verspätung hat und Mehrkosten verursacht.

Der Rote Faden durch die Folien

Referenz: SWT-23.24-Teil-3-v16. Während alle Projektarten bei Entwurf und Programmierung sehr ähnlich vorgehen, gibt es beim Management merkliche Unterschiede. Die angemessene Planung ist in Prinzip 7 verankert. Die Schätzung von Aufwand, Umfang und Dauer eines Projekts gehört eigentlich zu den Management-Aufgaben, wird aber wegen ihrer herausgehobenen Stellung als ein eigenes Prinzip behandelt. Denn hier gibt es in traditionellen wie in agilen Projekten jeweils eigene Techniken.

Folie F5 zeigt, wie das Schätzen im Prinzip immer funktioniert: Weil man nicht gut ein großes Projekt als Ganzes schätzen kann, identifiziert man kleinere Komponenten davon, für die man leichter schätzen kann. Dabei wird häufig der Umfang (wie viele dieser Komponenten sind zu entwickeln?) und die Schwierigkeit der Entwicklung getrennt ermittelt. Die Schwierigkeit ergibt sich aus der Komplexität der gesuchten Lösung, aber auch aus den Vorkenntnissen der Entwicklungsorganisation, aus vorhandenen Werkzeugen und Erfahrungen. Schwierigkeit und Umfang zusammen erlauben eine Schätzung, die dann noch gegen „ähnliche Projekte“ aus dem Erfahrungsschatz zu vergleichen sind.

Das Prinzip wird direkt bei Function Points F6 oder Use Case Points (keine Folie) eingesetzt. Aber auch COCOMO, eines der bekanntesten Schätzverfahren, funktioniert nach diesem Schema: Hier wird zuerst der Umfang in KLOC geschätzt, und dann kommen wieder Komplexitätsfaktoren hinzu, die die Schwierigkeit betreffen. Von F7 bzw. F8 sollte man sich vor allem die Größenordnung der Parameter merken und die Kurvenverläufe f(KLOC), die sich daraus ergeben. Da der Parameter b beispielsweise nur knapp über 1 ist, steigt die Kurve zwar exponentiell – aber sehr langsam, fast noch linear. Dagegen sind die d-Werte deutlich unter 1. Wenn also der Aufwand sich verdoppelt, wird sich die geschätzte Dauer bei weitem nicht verdoppeln, sondern viel weniger schnell wachsen. Wie kann das sein? Vor allem, weil große Projekte mit viel Aufwand von mehreren Bearbeitern (mehr Personal) parallel bearbeitet werden. Also wird die Personal-Zahl wachsen, die Dauer dagegen nicht so stark. Nimmt man aber beides zusammen (Personal*Dauer), dann sieht man, dass größere Projekte überproportional teurer werden: Ein hundert Mal so umfangreiches Projekt (600 KLOC statt 6) dauert zwar nicht hundert Mal so lang, es braucht aber über hundert Mal so viel Aufwand, kostet also insgesamt mehr als hundert Mal so viel wie das kleine Projekt. Wieso eigentlich? Ein Hauptgrund liegt im Abstimmungsbedarf. Mehr Personal braucht mehr Besprechungen und mehr Abstimmung, das ist zusätzlicher Overhead.

Nur kurz werden weitere Ansätze genannt F9. Bei der Prozentsatzmethode wendet man einen Dreisatz an: Wenn ich zuletzt (z.B. in den letzten Projekten) für die Anforderungen 20% des Projektaufwands gebraucht habe und jetzt im neuen Projekt 5 PM dafür investieren musste, dann heißt das nach Dreisatz, dass ich mit 25 PM Gesamtaufwand rechnen muss. Expertenschätzungen rechnen nicht, sondern überlassen es völlig den Experten, wie sie zu ihren Schätzungen kommen; die können auch aus dem „erfahrenen Bauch“ kommen. Jeder Experte sieht im nächsten Schritt die Schätzungen der anderen Experten und muss jeweils begründen, wieso sie oder er ggf. anders geschätzt hat. Nach mehreren Runden sollen die Schätzungen auf Basis der gesammelten Erfahrung konvergieren.

Für die Praxis sind Meilensteine F10, die Meilensteintrendanalyse F11 und Quality Gates F12 besonders wichtig. Sie dienen alle dazu, individuell für ein Projekt zu definieren, bis wann es welche Resultate geschafft haben sollte. Das wird in Meilensteinen definiert und für bestimmte Zeitpunkte eingeplant. In der Meilensteintrendanalyse beobachtet man, ob die Meilensteine eingehalten werden und zur geplanten Zeit fertig werden. Beobachtet man dagegen, dass alle Meilensteine erst später fertig werden, so leitet man aus diesem Trend eine Warnung ab: Wenn das so weitergeht, wird auch das ganze Projekt verspätet enden. Möchte man das verhindern, muss man umplanen oder andere Maßnahmen ergreifen, aber nicht einfach weitermachen und hoffen, dass es schneller weitergehen wird. Generell hat man festgestellt, dass Projekte mit Verspätung diese fast nie aufholen können.

Prinzip 7: Angemessene Planung und Management

Einführung und Motivation

Softwaretechnik oder Software Engineering ist eine Disziplin, bei der die Entwicklung von Software im Mittelpunkt steht. Sie wird aus verschiedenen Perspektiven betrachtet, die alle zusammengehören. Die ersten vier Prinzipien haben sehr eng mit Softwareentwicklung im engeren Sinne zu tun: Wie stellt man sicher, dass man genau verstanden hat, was die Software leisten soll? Wie soll sie strukturiert und entworfen sein, um das zu erreichen? Wie kann man verständlich programmieren? Und querschnittlich dazu: nach welchem System kann man arbeiten?

Das sind unverzichtbare Prinzipien, ohne die Softwareentwicklung nicht möglich wäre. Aber sie reichen nicht aus. Es ist kennzeichnend für Software Engineering, dass zusätzlich auch die wirtschaftlichen Aspekte berücksichtigt werden müssen. Projektleiter müssen nicht nur in der softwarenahen Technik kompetent sein, sie brauchen auch Grundkenntnisse von Wirtschaft und Management. Jeder gute Entwickler muss auch diese Seite sehen können. Dabei reicht es aus, gewisse Grundkenntnisse zu haben, man muss durchaus nicht auch noch Betriebswirtschaft studieren.

In diesem Kapitel werden diejenigen Aspekte vorgestellt, mit denen Projektleiter und eigentlich alle professionellen Softwareentwickler umgehen müssen: Es sind viele Aspekte der Planung, auch die oben schon eigens genannte Schätzung gehört dazu. Projektleiter kontrollieren den Status (den Fortschritt des Projekts), vergleichen ihn mit den Zielen (den Schätzungen und den Plandaten) und reagieren, wenn es größere Abweichungen gibt. Nebenbei vertreten Projektleiter ihre Projekte in Lenkungs- und Leitungskreisen mit den Kunden und im eigenen Unternehmen. Softwareentwicklung ist vielseitiger als man möglicherweise anfangs denkt.

Viele Untersuchungen haben gezeigt, dass der Erfolg oder Misserfolg von Softwareprojekten zu einem großen Teil davon abhängt, wie gut mit Anforderungen umgegangen wird – und wie realistisch die Planungen sind. Vernachlässigt man einen dieser Aspekte, ist das Projekt in Schieflage. Wenn dieser Rahmen nicht passt, hilft es auch nicht mehr viel, vorbildlich zu programmieren.

Muss denn wirklich jede und jeder Informatiker/in Projekte leiten? Die Grenze zwischen erfolgreichen Entwicklern und Projektleitungsaufgaben ist fließend. Wer in die Projektleitung berufen wird, sollte zu diesem Zeitpunkt schon ein solides Verständnis auch der Planungsaufgaben haben, denn dann ist es zu spät, alles nachzulernen. Daher gehört ein gewisses Maß an Planung und Management zu den Prinzipien des Software Engineering, zu den Grundlagen des Fachs.

Agile und traditionelle Methoden unterscheiden sich auch in diesem Aspekt sehr deutlich. Zwar sind alle Projekte auf angemessene Planung und Management angewiesen, aber was das konkret heißt, unterscheidet sich sehr stark. Ein Aspekt von Projektmanagement spielt eine ganz besondere Rolle: Dem Umgang mit Risiken ist daher ein eigenes Prinzip gewidmet.

Der Rote Faden durch die Folien

Referenz: SWT-23.24-Teil-3-v16. Das Kapitel über die Planung ist umfangreich. Für Informatiker ist es zunächst wichtig, ein paar Missverständnisse auszuräumen F22: Planung und Schätzen sind nicht dasselbe; man kann beim Planen durchaus von einer Schätzung abweichen. Andererseits ist es nicht hilfreich, völlig unrealistisch zu planen. Nur weil man es sich wünschen würde, wird man dennoch nicht in der halben geschätzten Zeit fertig werden. Wenn das wichtig ist, muss man entsprechende Maßnahmen ergreifen: mehr Personal einsetzen, bessere Werkzeuge, vielleicht auch externe Unterstützung.

Das Reglerschema auf F23 vergleicht die Aufgabe des Projektmanagements mit der eines Reglers, zum Beispiel eines Temperaturreglers. Beide müssen versuchen, ein Ziel zu erreichen. Das gelingt, indem man sich etwas vornimmt (plant) und dann entsprechende Maßnahmen ergreift, um das Ziel zu erreichen (führt). Dabei muss man externe Störungen (z.B. Risiken) abwehren. Um zu wissen, ob man auf gutem Kurs ist, muss man die Ausgangsgröße (Projektergebnis bzw. Temperatur) dauernd messen und mit den Planungen vergleichen. Am Anfang ist auch Schätzung erforderlich, um überhaupt zu vernünftigen Plänen zu kommen. Meist erfreulich: Projektleiter dürfen ihr Projekt auch repräsentieren und dabei Lob (und Tadel) einheimsen.

Barry Boehm hat geschrieben: *Never make a plan you do not believe in*. Das ist guter Rat. Wenn schon der Planende nicht glaubt, dass man den Plan einhalten kann, dann sollte man den Plan ändern. Auf F25ff sind verschiedene Möglichkeiten aufgetragen, wie viel Energie man in die Planung stecken kann. Die Darstellung lehnt sich an ein Buch von Barry Boehm und D. Turner an. Hier sieht man, dass sie verschiedene Methoden als Ausprägungen eines Spektrums sehen. Agile und traditionelle, planbasierte Verfahren gehören dazu.

Auf F30 zunächst eine begriffliche Klärung, wann man von einem Projekt spricht. Nicht alles ist ein Projekt. Wartungsaufgaben sind nach dieser Definition eher kein Projekt, sondern eine Dauerbeschäftigung.

Man macht sich als technisch orientierter Informatiker vielleicht gar nicht klar, dass auch ein Projekt nicht für sich besteht, sondern von einem Unternehmen durchgeführt wird, um Geld zu verdienen. Die Kunden wiederum wollen über den Fortschritt informiert werden. Alles dies muss ein Projektleiter/in leisten, und dafür gibt es entsprechende Strukturen, je nach Projektgröße.

Ab F35 geht es um die konkreten Planungsaufgaben, unter denen sich jede/r Entwickler/in etwas vorstellen können sollte. Die Aspekte werden in der Folge kurz am Beispiel eines Mensa-Karten-Projekts gezeigt. F41 ist wichtig: Hier sehen sie den Algorithmus, mit dem man in Netzplänen rechnet. Das müssen Sie beherrschen.

In agilen Projekten hat man zwei Planungsebenen: Einen langfristigen Masterplan, und kurzfristig verwendet man Task Boards F48. Manche Aspekte sind vorgeplant: Jeder Sprint dauert gleich lang, hier 30 Arbeitstage. Das Team besteht aus 7+/-2 Personen oder Paaren. In diesem Umfeld werden Task-Boards wie F54ff eingesetzt, um die Aufgaben nach Inhaltsthema (Spalten) und nach Dringlichkeit (Iterationen vertikal) zu sortieren – und immer wieder neu zu sortieren. F49 ist praktisch dasselbe, aber aus einem echten Projektumfeld. So sehen Task Boards in der Praxis aus.

Es gibt verschiedene Varianten, wobei aber das Task-Board mit die häufigste sein dürfte. Mit sogenannten KANBAN-Boards F51-F53 geht man einen etwas anderen Weg. Man geht davon aus, dass die Aufgaben alle ähnlich groß bzw. umfangreich sind. Auf dem KANBAN-Board ist der Entwicklungsprozess abgebildet. Für jeden Schritt zwei Spalten: Das, was wartet (links) und das, was gerade bearbeitet wird (rechts).

Die Idee von KANBAN: Man drückt nicht immer mehr Aufgaben ins System, dadurch geht es auch nicht schneller. Im Gegenteil: Das Umschalten führt zu Verzögerungen und Fehlern. Also kehrt man es um und folgt einer pull-Strategie: Statt links etwas hineinzudrücken, wird von rechts gezogen, wenn eine Aufgabe fertig ist. Dann entsteht eine Lücke, es kann von der nächst linkeren Spalte eine Aufgabe nachgezogen werden. Das kann zu Folgeeffekten führen. Wann ist eine Spalte „voll“? Das entscheidet der WIP-Wert, der von den Prozessingenieuren eingestellt wird F52. Er steht für Work-in-Progress. Wenn Entwurf einen WIP von 7 hat, dürfen höchsten 7 Aufgaben in den beiden Entwurfsspalten stehen. JIRA F53 ist ein kommerzielles Werkzeug zur Verwaltung von Aufgaben, Issues (Anliegen, die bearbeitet werden müssen) und verwandten Informationen. Es bietet eine weitere Variante an, bei dem drei Zustände für jede Aufgabe unterschieden werden. Damit ähnelt es mehr einem KANBAN-Board als einem Task-Board, bei dem ja inhaltlich definierte Spalten verwendet werden. In JIRA kann man allerdings vieles anpassen, so auch das Board.

Prinzip 8: Bewusst mit Risiken umgehen

Einführung und Motivation

Jedes interessante Projekt ist einzigartig und es ist nicht vollständig vorhersehbar, was im Projektverlauf passieren wird, und ob man alle Ziele wie geplant erreicht. Gerade dadurch, dass die Ziele herausfordernd sind, kann auch einmal etwas schiefgehen. Projektrisiken sind die Kehrseite von interessanten Projekten. Weil man spannende, vielversprechende Projekte mit hohem Potenzial machen möchte, muss man auch deren Risiken akzeptieren. Ein Projekt ohne Risiken ist langweilig und wirft vermutlich nicht viel Gewinn ab.

Andererseits sollen die Probleme nicht das Projekt gefährden, oder gar das eigene Softwareunternehmen in Gefahr bringen. Daher muss man balancieren: Risiken dürfen und werden auftreten, man muss aber so mit ihnen umgehen, dass sie keinen zu großen Schaden anrichten können. In traditionellen Projekten gibt es dafür das Risiko-Management, einen Spezialbereich des Projektmanagements. Es geht im Wesentlichen darum, große Risiken früh zu erkennen und sie beherrschbar zu machen. Entweder sorgt man dafür, dass sie wahrscheinlich nicht eintreten. Oder man trifft Vorsorge, damit der Schaden nicht so groß wird, wenn ein Risiko dann doch eintritt. Das ist Risikomanagement.

In agilen Projekten ist der bewusste Umgang mit Risiken dagegen nicht an einer Stelle (wie dem Risikomanagement) konzentriert, sondern an vielen Stellen eng mit der Entwicklung verwoben. Schon die kurzen Entwicklungszyklen, die ständige Einbindung des Kunden und der kontinuierliche Test sind Maßnahmen, die häufig auftretende Risiken in Softwareentwicklungsprojekten begrenzen sollen. Durch die kurzen Zyklen kann man erkannten Risiken noch entgegenwirken, bevor sie zu richtigen Problemen geworden sind. Ständiger Kundenkontakt soll das häufige und gravierende Risiko reduzieren, eine Software zu bauen, die den Kundenanforderungen nicht genügt. Wenn der Kunde ständig mit dem neuen Stand konfrontiert wird, können Missverständnisse früh erkannt werden. Testen vermindert stets das Risiko, fehlerhafte Software auszuliefern. Neben diesen allgegenwärtigen Projektrisiken gibt es aber auch ganz spezifische, die sich aus den Schwierigkeiten eines Projekts ergeben.

In diesem letzten Prinzip sollte man in jedem Projekt die Maßnahmen von agilen und von traditionellen Methoden kennen und im Hinterkopf mitberücksichtigen. Das beginnt bei Barry Boehms Definition eines Risikos: *Ein Risiko ist ein mögliches Problem, das eintreten kann, aber nicht muss.* Die Dringlichkeit eines Risikos ermittelt sich aus dem Produkt der Auftretenswahrscheinlichkeit (des Problems) und dem dann zu befürchtenden Schaden. Agile Projekte haben gute Mechanismen fest eingebaut, die praktisch nebenher auch Risiken mindern. Diese Anregungen können auch traditionellen Projekten guttun.

Der Rote Faden durch die Folien

Referenz: SWT-23.24-Teil-3-v16. Auf F71 beginnt die Darstellung des traditionellen Risiko-Managements mit den Grundbegriffen, die man auch auf agiler Seite verwenden kann: Die Definition für ein Risiko ist sehr wichtig, es lohnt sich, sie zu verstehen und auswendig zu lernen. Barry Boehms Definition von Risk Exposure lehnt sich direkt daran an: Indem die Wahrscheinlichkeit (die eben weder 0 noch 1 ist, sondern dazwischen liegt) mit dem möglichen Schaden verknüpft, hat man diese beiden Aspekte berücksichtigt. Sie werden, genauer gesagt, multipliziert, so dass die Risk Exposure sehr klein ist, wenn nur eine der beiden Größen sehr gering ist. Was entweder keinen Schaden anrichtet oder keinesfalls eintritt, ist kein wichtiges Risiko. Die höchste Risk Exposure haben Risiken, die wahrscheinlich eintreten und außerdem großen Schaden anrichten. Man wird sich zuerst um die Risiken mit hoher Risk Exposure kümmern, denn dort ist der Erwartungswert (Wahrscheinlichkeit * Schadenshöhe) für mögliche Schäden am höchsten.

Es gibt sehr konkrete Konsequenzen aus der Erkenntnis, dass spannende Projekte praktisch immer auch riskant sind. Ein bekanntes Vorgehensmodell, das zwischen traditionellen und agilen Methoden steht, ist das Spiralmodell von Barry Boehm F73. Es ist einerseits iterativ, innerhalb der Iterationen gibt es aber traditionelle Abläufe. Das Kennzeichnendste ist die Grundidee, immer das größte Risiko zuerst anzugeben. Wenn es gelöst ist, wird die Lösung ins Produkt übernommen, und dann fährt man mit dem nun größten Risiko fort. Dabei kann ein Risiko software-technischer Art sein (Entwurf ist zu un-flexibel), oder auch organisatorischer (wir können gar nicht so viele Pakete verschicken), oder die Bedienbarkeit betreffen (Wenn die Kunden nicht verstehen, wie man bestellt….). Die Spirale ergibt sich, weil man ganz innen beginnt und in jedem Iterationsumlauf zunächst das höchste Risiko (nach Risk Exposure) bestimmt, dann mit Prototypen die Lösung sucht, sie einbaut und plant und dann wieder neu bewertet, was jetzt das größte Risiko ist. Das Produkt wächst dabei, angedeutet durch die nach außen wachsende Spirale. In der Praxis ist das Spiralmodell zwar sehr bekannt, es wird in reiner Form aber nur sehr selten angewendet. Man stößt aber immer wieder einmal auf die Idee, sich im Zweifel zuerst um die größten Risiken zu kümmern.

F74 zieht einen Vergleich von Risiken zu Eisbergen: Man sieht sie kaum, und doch können sie großen Schaden anrichten, wenn man nicht aufpasst. Was man tun kann, ist ähnlich. F75 führt dann das Format ein, das wir in der Vorlesung verwenden, um die wesentlichen Aspekte eines Risikos zu betonen und deutlich zu machen, wo man beim Umgang damit angreifen kann:

> WENN <Bedingung> DANN <Folge mit Schaden>.

Noch schöner ist auf F78 der Schaden explizit gezeigt. Man kann etwas tun, um die Bedingung unwahrscheinlicher zu machen oder den Schaden kleiner. F77 zeigt ein Risikoportfolio, eine sehr häufige Darstellung der zwei Achsen (Wahrscheinlichkeit, Schaden). Die Risk Exposure (das Produkt Wahrscheinlichkeit*Schaden) ist am kleinsten bei geringem Schaden und geringer Wahrscheinlichkeit. Die gravierendsten Risiken haben hohe Wahrscheinlichkeit und Schäden, sind also rechts oben angesiedelt. Man wird daher von rechts oben nach links unten diagonal durch das Portfolio gehen, wenn man die Risiken priorisieren will: umgekehrt zu dem gestrichelten Pfeil, der wachsende Risk Exposure zeigt. Im Endeffekt wird man sich dann nur um die wichtigsten kümmern können.

Agile Methoden wurden eigens entwickelt, um bestimmte, immer wiederkehrende Risiken besser in den Griff zu bekommen. Daher sind die entsprechenden Maßnahmen tief mit den Entwicklungs-schritten verbunden F81. F82 zeigt das explizit. Wie die Methoden funktionieren, wurde ja schon besprochen. Hier ist wichtig zu verdeutlichen, dass sich agile Methoden intensiv um Projektrisiken kümmern – aber nicht unter einer eigenen Überschrift „Risiko Management“, sondern als Teil der täglichen Arbeit.

Dennoch sind es eher die allgemeinen Risiken (Kundenanforderungen missverstanden, Fehler im Programm, Verzögerungen im Projekt). Möchte man ganz spezifische Projektrisiken miteinbeziehen, muss man das auch in agilen Projekten eigens tun, also das nutzen, was man im Risiko-Management schon lange tut: Risiken suchen, bekämpfen, verfolgen.

Hybride Entwicklung und Mischformen

Am Anfang der Vorlesung wurde beschrieben, dass es den traditionellen und den agilen Ansatz zur Softwareentwicklung gibt. Statt die beiden einfach nacheinander zu präsentieren, wurden hier acht Prinzipien identifiziert, die in beiden Ansätzen vorkommen – wenn sie auch teilweise verschieden umgesetzt sind. Besonders bei den ersten und den letzten Prinzipien dominieren die Unterschiede: Anforderungen werden in agilen Projekten ganz anders gehandhabt als in traditionellem Requirements Engineering. Auch Planung und Steuerung verlaufen ganz anders. Dagegen sind die programmiernahen Prinzipien ähnlich ausgeprägt: Es spielt keine große Rolle für das Information Hiding oder für den Einsatz von Design Patterns, in welcher Art von Projekt man sich befindet.

Mehrfach wurde darauf hingewiesen, dass seit einigen Jahren die strenge Trennung in agile und traditionelle Projekte nachlässt. Immer mehr Unternehmen ergänzen ihre Vorgehensweise mit Elementen aus dem anderen Bereich. Nicht jede agile Praktik wird vollständig umgesetzt; beispielsweise kann *Pair Programming* nur zu einem Teil der Zeit eingesetzt werden; oder man verständigt sich einfach regelmäßig über Reviews. Das sind graduelle, nur teilweise eingesetzte Praktiken. Daher ist es so wichtig, beide Seiten zu kennen und nicht nur in einem der Bereiche kompetent zu sein.

Aktuell nehmen die Mischformen zu, bei denen entweder einzelne Elemente in die jeweils andere Projektart übernommen werden: SCRUM-Meetings in traditionellen Projekten; Spezifikationen in agilen Projekten; Risiko-Management in einem SCRUM-Projekt. Dabei ist heute noch nicht erforscht, was die Auswirkungen solcher Mischungen sind. Im Rahmen der Vorlesung soll einerseits darauf hingewiesen werden, dass Sie in der Praxis mit Mischformen zu rechnen haben. Daher folgt hier ein kurzer Überblick über die wichtigsten Arten. Anschließend wird das Vorgehensmodell des Softwareprojekts (SWP) im fünften Semester behandelt. Das hat zwei Gründe:

- Erstens sehen Sie daran eine echte Mischform, in der Elemente aus beiden Bereichen in ähnlichem Umfang auftreten.
- Zweitens sind Sie konkret auf das SWP vorbereitet und müssen sich nicht erst an dessen Anfang die Grundlagen aneignen. Wir gehen im SWP aber auch davon aus, dass Sie die Einführung in die Vorgehensweise bereits kennen. Sie wird zwar noch einmal aufgefrischt, sollte für Sie aber schon bekannt sein.

Eine weitere Vertiefung zu hybrider Entwicklung können dann entsprechende Vorlesungen im Masterstudium bilden.

Praktiken nicht vollständig umsetzen. Die einfachste Anpassung besteht darin, einzelne agile (oder auch traditionelle) Vorgehensbestandteile absichtlich nicht vollständig umzusetzen. Beispielsweise schaffen es viele Unternehmen nicht, Test First in aller Konsequenz umzusetzen. Sie beschließen, immerhin einen großen Teil von Testfällen schon vor den Programmen zu schreiben – oder zumindest gleichzeitig. Das sind Einschränkungen, und sie werden nicht die ganze Wirkung von Test First entfalten, bei dem ja jedes Mal zuerst ein Test auf ein nicht bestehendes Programm geschrieben werden müsste. Das ist vielen zu albern. Vor allem aber fehlt es an der Disziplin, jedes Programmstück nur als Reaktion auf einen zuvor fehlgeschlagenen Test zu schreiben. Das kann man so einschränken, es ist pragmatisch, wird aber auch nicht so gut wirken wie der konsequente Test First.

Umgekehrt könnte ein Unternehmen eine verkürzte Form der Spezifikation einführen, in der schon User Stories enthalten sind. Das ist in Spezifikationen nicht üblich; die Use Cases wurden extra eingeführt, damit man auch an Akteure und Interessen, Vorbedingungen und Garantien denkt. Wenn man das alles weglässt, geht es natürlich erst einmal schneller.

Aber man vergisst auch leichter etwas. Früher hätte man solche Abschwächungen immer als einen Fehler betrachtet, heute sind absichtliche Anpassungen üblich und nicht immer von Nachteil.

Elemente übernehmen. Man übernimmt agile Praktiken in traditionelle Vorgehensweisen oder umgekehrt: SCRUM-Meetings in einem traditionell strukturierten Projekt tragen den iterativen Charakter hinein. Risiko-Management in einem agilen Projekt erhöht die Aufmerksamkeit gegenüber potenziellen Problemen. Meist hat das Vorteile, aber auch Nachteile. Weder das SCRUM-Meeting im traditionellen Projekt, noch das Risiko-Management im agilen sind mit dem Rest des Projekts ideal verbunden. Sie waren ja dort nicht vorgesehen, und daher gibt es Sprünge bei den Übergängen.

Phasenweiser Wechsel. Sehr häufig versuchen Unternehmen (z.B. aus dem Automobilbau), schneller und flexibler in ihrer Softwareentwicklung zu werden, ohne dabei das bewährte V-Modell aufzugeben. Denn das reife V-Modell kennen sie gut; es bietet klare Schnittstellen zwischen Hard- und Software, und es sieht viele qualitätssichernde Maßnahmen vor, die für Audits und Zertifizierungen unerlässlich sind. Das V-Modell soll also das hauptsächliche Vorgehensmodell bleiben. Aber im Rahmen eines V-Abschnitts (von den Anforderungen über Implementierung bis zu Test und Integration) könnte man ja agiler arbeiten? Also wird aus einem Projektplan aus vielen V-Durchläufen eines oder einige herausgenommen und intern durch agile Praktiken ersetzt. Das ist möglich und erspart viele Dokumente, die sonst viel Zeit in Anspruch genommen hätten. Damit das Resultat in den Gesamtplan passt, müssen aber die Schnittstellen zwischen V und agilem Teil definiert und eingehalten werden. Das ist nicht einfach, und ein Teil der Schnelligkeit, Flexibilität und andererseits Prüfbarkeit gehen dabei wieder verloren. Dennoch ist das Verfahren weit verbreitet; offenbar lohnt es sich.

Neues Verfahren durch konzeptionelle Durchdringung. Die komplizierteste und anspruchsvollste Mischform ergibt sich, wenn nicht nur einzelne Elemente übernommen oder das ganze Verfahren zwischendurch gewechselt wird – sondern größere Anteile zweier oder mehrerer Verfahren konzeptionell eng miteinander verbunden werden. Dann entstehen neue Praktiken und neue Ideen, wie man die Softwareentwicklung verbessern kann. Diese höchste Form der Verschmelzung ist relativ selten; sie erfordert viel Vorarbeiten und gute Prozesskenntnisse. Das können in der Regel nur Spezialisten vorbereiten; neben dem Tagesgeschäft lässt es sich nicht machen. Man muss auch deutlich sehen, dass die Entwicklung einer neuen Methode in diesem Sinn hohe Risiken birgt: Die gewünschten Effekte könnten ausbleiben; unerwartete Probleme könnten auftreten; und sogar, wenn alles klappt, wird es nicht möglich sein, neue Mitarbeiter einzustellen, die die Methode schon kennen. Denn es gibt sie nur in dem einen Unternehmen.

Konkretes Beispiel: Das hybride Vorgehensmodell im SWP

Nach der obigen Klassifikation ist das Vorgehensmodell durch einen „Wechsel“ nach dem ersten Monat gekennzeichnet F57ff. Die Grundidee war, einerseits unbedingt eine herkömmliche Spezifikation zu schreiben, weil sowohl die Kunden als auch die Entwickler unerfahren sind und einen Gesamtüberblick über das Projekt brauchen, wenn sie erstmals ins SWP kommen – und das gilt für praktisch alle Studierenden, die das SWP ja nur einmal belegen. Also beginnen wir mit einer üblichen, traditionellen Vorgehensweise: Requirements Engineering, das zu einer normalen Spezifikation führt. So eine Spezifikation können Kunden lesen und verstehen, und während der gesamten Projektlaufzeit kann man sie immer noch konsultieren. Das Wichtigste: Es gibt einen klaren Startpunkt, der eindeutig dokumentiert und freigegeben wird.

Dann allerdings verändern sich die Prioritäten, und das führt zu einem absichtlichen Wechsel im gesamten Verfahren. Von jetzt an werden nicht mehr alle Dokumente und Prüfungen eines typisch traditionellen Vorgehensmodells erstellt, sondern man geht auf leicht-gewichtige, flexible und schnelle Entwicklung in Iterationen über. Das ist kein streng agiles Vorgehen nach XP, aber die meisten

Elemente von SCRUM sind enthalten; allerdings in angepasster Form: Denn das SWP trifft sich nicht täglich, wie normale Industrieprojekte. Daher gibt es keinen daily SCRUM, sondern einen weekly SCRUM. Der allerdings ist konsequent einzuhalten und so durchzuführen, wie es auch in Industrieprojekten erfolgen würde. Man ist also vollständig von der traditionellen auf eine agilere Phase umgestiegen. Wichtig: Bei Änderungen an den User Stories wird nicht mehr die ganze Spezifikation nachgeführt und aktualisiert – nur noch die Abnahmetests. Das ist eine Mischung, die sich bewährt hat.

Jede Mischung hat Vor- und Nachteile, so auch im SWP. In sorgfältiger Abwägung haben wir entschieden, dass der relativ geringe Mehraufwand zum Schreiben einer Spezifikation vertretbar ist, wenn dafür die entsprechenden Vorteile entstehen:

Aus Projektsicht

- Ein klarer Bezugspunkt zum Start, die Spezifikation. Sie wird aber nicht weiter aktualisiert.
- Danach eine klar auf Flexibilität und Schnelligkeit optimierte Weiterarbeit.
- Die Spezifikation zur Orientierung und Verständigung zwischen Entwicklern und Kunden.
- Abnahmetests werden bei Änderungen mitgeändert.

Aus Sicht der Lehre bzw. der Studierenden:

- Nebenbei ist es aus Sicht der Lehre erforderlich, dass Studierende in einem echten Projektumfeld eine Spezifikation schreiben können und dies auch üben.
- Umgekehrt soll die kontinuierliche Anpassung und Neupriorisierung von User Stories praktiziert und geübt werden, daher ist die Nutzung von Story Cards auf einem Task Board erforderlich.

Das angepasste Entwicklungsmodell nach F57 vereint diese Vorteile.

Vertiefung zu besonders wichtigen Themen

Dieses Vertiefungskapitel fasst drei wichtige Themen noch einmal zusammen. Ohne auf einzelne Folien zu verweisen, werden diese Themen erklärt, damit man für das Nötigste auch ohne die Erklärungen in der Vorlesung auskommt. Vor allem sollen diese Erläuterungen Ihnen helfen, vor der Klausur und vor dem Software-Projekt im fünften Semester noch einmal im Zusammenhang zu wiederholen.

Alle Themen aus der Vorlesung gehören zum Grundwissen der Softwaretechnik und sind damit klausurrelevant. Wer sie besser beherrscht, wird praktische Projekte besser bearbeiten können. Die Themen der Vertiefungen sind danach ausgewählt, was erfahrungsgemäß den größten Einfluss auf Erfolg oder Misserfolg eines Projekts haben wird. Dazu bieten wir auch die Teamaufgaben an.

Vertiefung: Spezifikation und Use Cases

In Spezifikationen und Use Cases werden Anforderungen und zugehörige Informationen aufgeschrieben. Meist sind sie in natürlicher Sprache (normales Englisch oder Deutsch) gehalten, damit alle Beteiligten sie gut verstehen können. Sowohl Use Cases als auch die gesamte Spezifikation folgt häufig einer vorgegebenen Struktur, die sich aus Erfahrung bewährt hat.

Wozu dient eine Spezifikation?

Softwareprojekte sind heute so groß, dass man sie nur in Arbeitsteilung erledigen kann. Daher dient die Spezifikation als gemeinsamer Bezugspunkt von Kunden und Entwicklern. Im Idealfall steht in der Spezifikation klar und verständlich, was die Kunden haben möchten; sie können das überprüfen und bei Bedarf korrigieren. Gleichzeitig soll die Spezifikation aber für die Entwicklung unmissverständlich mitteilen, wie diese Kundenwünsche umzusetzen sind, was also das System am Ende leisten soll. Manche Kundenwünsche lassen sich auf unterschiedliche Weise erfüllen.

Welche Varianten gibt es?

Wenn man eine Spezifikation schreiben soll, muss man sich zunächst fragen, ob sie aus Benutzer- bzw. Kundensicht verfasst sein soll. Sie sagt aus, was der Kunde möchte. Auf Englisch sagt man dazu „user specification“, auf Deutsch heißt das *Lastenheft*. Dieses Dokument ist die Basis für ein Angebot, das ein Softwarehaus macht. Das Lastenheft wird gelesen und das Softwarehaus beschreibt in einer „system specification“ (*Pflichtenheft*), wie das System aussehen würde, wenn das Softwarehaus den Auftrag bekommt. Es gibt eben oft mehrere Möglichkeiten, die Kundenwünsche zu erfüllen. Im Angebot sucht das Softwarehaus eine davon aus und bietet sie als Pflichtenheft an.

Darin ist nun schon etwas genauer beschrieben, ob z.B. eine Datenbank oder eine Dateistruktur zur Ablage verwendet werden soll. Das war in der user specification vielleicht noch offengeblieben. Die system specification oder das *Pflichtenheft* ist dann die Arbeitsanweisung an die Entwickler. Sobald der Kunde die system specification unterschrieben hat, ist das der Startschuss: Das System soll dann so aufgebaut werden, dass es dem Pflichtenheft entspricht.

Bei kleineren Projekten unterscheidet man nicht zwischen Lasten- und Pflichtenheft, sondern fasst beides unter dem Begriff „*Spezifikation*“ zusammen. So ist es auch in der Lehrveranstaltung Software-Projekt (SWP). Entsprechend müssen solche kombinierten Spezifikationen beide Aspekte umfassen: Benutzer- und Systemsicht. Damit sie zur Prüfung durch die Kunden und als ausreichend klare Vorgabe für die Entwickler taugen, müssen solche Spezifikationen *für beide Seiten verständlich formuliert* sein. Das bedeutet: Nicht zu technisch; denn das würden die Kunden vielleicht nicht verstehen; aber auch spezielle Kundenausdrücke müssen erklärt werden, damit die Entwickler wissen, was gemeint ist.

Die Falle: Das sieht so einfach aus!

Anforderungen, Use Cases und die Spezifikation: Damit hat jedes Softwareprojekt zu tun; sieht man sich solche Dokumente an, wirken sie nicht besonders kompliziert. Wenn man noch nie selbst daran mitgeschrieben hat, kann man den Eindruck bekommen, das sei ganz einfach: Natürliche Sprache, bei den Use Cases in eine Tabelle geschrieben, das kann nicht so schwierig sein. Denkt man. Doch das ist falsch. Denn es ist viel schwieriger, Use Cases (UC) gut zu schreiben als sie zu lesen.

Macht man es dann zum ersten Mal in einem echten Projekt, merkt man nämlich, dass es sehr genau darauf ankommt, *welche* Sätze man schreibt und *wie* man die Use Case-Tabellen ausfüllt. Ist man dann in einem echten Projekt, dann muss man es können. Dann ist es zu spät, das noch zu lernen oder erst zu üben. Sehr vielen Studierenden ist es im Software-Projekt (SWP) in der Vergangenheit so ergangen: die Kunden hatten eine Spezifikation nicht unterschrieben, weil Use Cases falsch, belanglos oder unverständlich waren. Daher legen wir schon in SWT jetzt wesentlich mehr Wert auf das Thema: Schreiben Sie schon jetzt so viele Spezifikationen und Use Cases wie möglich und versuchen Sie, aus dieser Erfahrung und ggf. aus eigenen Fehlern und Erfolgen zu lernen. Wenn es dann im SWP darauf ankommt, sollten Sie schon wissen, wie es praktisch funktioniert.

Das Problem ist ja: Fehler oder Missverständnisse in Anforderungen, Spezifikationen und Use Cases sind viel schlimmer und folgenreicher als in Programmcode. Einen Syntaxfehler findet sofort der Compiler, dann ist er in Minuten behoben. Falsche Anforderungen dagegen führen zu falschen Entwürfen, diese zu Programmen, die zwar „laufen“, aber etwas Falsches tun. Man findet die Fehler im Programm auch erst einmal gar nicht, weil das (falsche) Programm ja genau den (falschen) Anforderungen entspricht. Da hilft selbst Testen nichts, denn auch die Testfälle gehen ja von den falschen Anforderungen aus. Es zeigt sich in der Praxis, dass korrekte Anforderungen einer der wichtigsten Voraussetzungen für ein erfolgreiches Projekt ist.

Daher lohnt es sich, Spezifikationen und Use Cases noch einmal ausführlicher zu behandeln, und es lohnt sich auf jeden Fall für Sie, mehrere (ca. 20) Use Cases zu schreiben, bevor Sie auch nur ins SWP gehen. Das können Sie in beschränktem Umfang in den Übungen tun, in schon deutlich realistischerer Weise aber auch in der Team-Aufgabe. Experten aus Wissenschaft und Unternehmen sind sich einig, dass alle kompetenten Projektmitarbeiter in der Lage sein müssen, eine gute Spezifikation zu schreiben, in der auch Use Cases enthalten sind. Daher lernen und üben Sie das intensiv. Das erfordert Ihre Eigeninitiative; sie lohnt sich.

Aufbau und Eigenschaften einer Spezifikation

In kleineren Projekten spricht nur von der Spezifikation (user and system specification), und darauf ist auch die Vorlage (das Template) ausgerichtet, das wir in der Vorlesung verwenden. Es ist in Unternehmen üblich, für Spezifikationen ein Inhaltsverzeichnis oder ein gesamtes Template zur Verfügung zu stellen, das dann für jedes neue Projekt ausgefüllt wird. Diese Vorlage soll es den Requirements Engineers erleichtern, an alles zu denken, was in eine Spezifikation gehört. Und diejenigen, die die Spezifikation lesen, sollen sich darin schnell und gut zurechtfinden: Beispielsweise können Designer, Entwickler oder Tester eine standardisierte, immer gleiche Struktur beim dritten oder vierten Projekt schon viel leichter verstehen, weil sie wissen, was an welcher Stelle steht. In unserem Template sind sogar Hinweise und Beispiele enthalten. Achtung: das sind „normale“ Beispiele, keine Musterlösung.

Müsste man ***ohne*** solch eine Vorlage eine Spezifikation schreiben, so würde man sich fragen:

1. Welche inhaltlichen Punkte müssen in welcher Form in der Spezifikation enthalten sein?
2. Wie schafft man es, diese Punkte herauszufinden bzw. vom Kunden mitgeteilt zu bekommen?
3. In welcher Reihenfolge werden die verschiedenen Aspekte präsentiert?
4. Wie formuliert man Anforderungen, Qualitätsanforderungen und Use Cases am besten?
5. Welche Eigenschaften sollte eine Spezifikation als Ganze haben und wie kann man das prüfen?
6. Gibt es Beispiele für gute Spezifikationen – und darf man davon abweichen?

Die Vorlage (das Template) versucht, bei diesen Fragen zu helfen. Eine Vorlage (ein Template) enthält ein Inhaltsverzeichnis, das man möglichst unverändert übernehmen sollte. Wenn ein Punkt im Inhaltsverzeichnis nicht anwendbar ist, löscht man nicht die Überschrift, sondern schreibt darunter: „nicht anwendbar". Fehlt dagegen ein Punkt im Inhaltsverzeichnis, der in einem speziellen Projekt benötigt wird, so hängt man ein Kapitel hinten an, oder ein Unterkapitel unter ein bestehendes. Auf jeden Fall versucht man, die vorgegebene Struktur nicht zu verändern, sondern sie nur bei Bedarf zu erweitern. Dann wissen alle Beteiligten aus Erfahrung, welche Punkte in welcher Reihenfolge auftreten. Welche Inhalte man jeweils unter die Überschriften schreibt, hängt natürlich vom konkreten Projekt ab.

Unsere spezielle Vorlage in SWT und SWP enthält außerdem farbig markierte Eintragungen und Hinweise, die zum Teil aus früheren Projekten stammen und als Beispiel verwendet werden. Natürlich muss man diese farbigen Texte löschen und durch eigene ersetzen. Die Beispiele zeigen, wie formuliert werden kann. Leider ist es kaum möglich, eine perfekte Musterlösung anzugeben, weil es in einem realen Projekt nie nur eine gute Lösung/Spezifikation gibt.

Wenn man also schon weiß, was und in welcher Reihenfolge beschrieben werden muss, stellt sich noch die Frage nach den ***Eigenschaften einer guten Spezifikation als Ganzes***. Dazu werden in Literatur und Lehrbüchern häufig genannt:

7. ***Vollständig***: Alle Angaben, die die Entwickler brauchen, stehen drin.
8. ***Korrekt***: Es sind keine falschen Angaben enthalten.
9. ***Prüfbar***: Durch die Formulierung kann man feststellen, ob Anforderungen erfüllt sind.
10. ***Konsistent***: Es gibt keine Widersprüche innerhalb der Spezifikation.
11. ***auf den Leser hin formuliert***: Für Kunden und Entwickler verständlich.
12. ***Sortierbar***: Man kann den Inhalt nach verschiedenen Kriterien filtern und sortieren.

Diese Eigenschaften sind wünschenswert und man muss sie anstreben. Die Eigenschaften widersprechen sich aber zum Teil. Bei manchen weiß man nicht, was sie praktisch im Einzelfall bedeuten (z.B. korrekt); man muss sich dann selbst bemühen, sie möglichst gut zu erfüllen. Zum Beispiel kann man Vollständigkeit (im Vergleich zu den Kundenwünschen) und Korrektheit eigentlich nur prüfen, indem man den Kunden fragt. Das ist aber wieder schwieriger als gedacht und erfordert z.B. Prototypen, damit sich die Kunden vorstellen können, wie das Produkt dann aussähe, wenn es nach der Spezifikation gebaut würde. Das heißt, dass man nun das Ziel kennt, aber nicht genau weiß, wie man es in einem konkreten Projekt erreicht. Dennoch ist es wichtig, zumindest das Ziel zu kennen.

Was kann man tun, um dem Ziel nahe zu kommen? Idealerweise sollten Anforderungen klar formuliert sein, so dass Tester leicht Testfälle schreiben können, ohne raten zu müssen, was genau gemeint war. Da eine Spezifikation oft umfangreich ist und mehrere Bestandteile hat (z.B. Use Cases, Qualitätsanforderungen, Text), sollten alle Teile zusammenpassen, dürfen sich also nicht widersprechen (nicht inkonsistent sein).

Voraussetzung für die Prüfung von Vollständigkeit und Korrektheit ist, dass Kunden die Spezifikation verstehen können. Daher sind Spezifikationen in der Regel auch in natürlicher Sprache gehalten und nicht in einem komplizierten Formalismus. Relativ konkret ist die Forderung nach „Sortierbarkeit“. Das ist eigentlich nur dann umsetzbar, wenn die Anforderungen in einer Datenbank stehen und die Datensätze darin nach verschiedenen Kriterien gefiltert und sortiert werden können: Welche Person hat die Anforderung eingebracht? Bezieht sie sich auf Oberfläche, Logik oder Datenhaltung? Sind es wichtige Anforderungen oder nur „nice-to-have“? Wie hängen Anforderungen zusammen und wie stehen sie zu den Use Case-Diagrammen? In der Praxis wird oft Doors verwendet. Das ist im Wesentlichen eine Datenbank aus Einträgen, sie man sehr gut sortieren und filtern kann. Die Einträge sind Anforderungen und damit zusammenhängende Informationen. Jede ausgedruckte Fassung dieser Spezifikation ist dann nur eine Sicht; die eigentliche Spezifikation selbst steht in der Datenbank.

Die Vorlage (das Template) einer Spezifikation

Im Folgenden wird jeder Teil des Template erläutert. Verwenden Sie das Template vor allem als eine Gedächtnishilfe oder Checkliste: Welche Punkte müssen Sie herausfinden, worüber mit den Kunden sprechen? ***Referenz***: Template-Spez-WS19.20-v05

Aspekt	Kapitel/ Seite	Kommentar
Verwen-dungs-hinweise (gelb markiert)	Deckblatt	Lesen und befolgen Sie die Angaben und Hinweise, sie sind zu Ihrer Unterstützung gedacht. Löschen Sie sie danach.
Unterschrif ten		Der Projektleiter muss bestätigen, dass sowohl User- als auch System- Specification abgedeckt sind. Man kann sich später nicht herausreden, das habe man nicht gewusst. Die Kundenunterschrift ist Voraussetzung dafür, dass weitergearbeitet werden kann. Im Quality Gate wird die Unterschrift gefordert: Es hat ja keinen Sinn, etwas umzusetzen, womit der Kunde nicht einverstanden ist.
Inhalt	Inhalts-verzeichnis	Das Template gibt Überschriften für alle Kapitel vor. Damit sie aktuell bleiben, sollte das Inhaltsverzeichnis immer automatisch generiert werden.
Mission	Kap. 1	Diese kurze, natürlich-sprachliche Beschreibung zeigt, ob die Bearbeiter das Problem ausreichend verstanden haben. Unterschätzen Sie diesen Absatz nicht! Hier stehen nur die wichtigsten Dinge über das Projekt, die müssen aber richtig getroffen sein. Kunden und Entwickler müssen verstehen und einig sein, worum es im Projekt vor allem geht.
	Weitere Teile	Siehe die Kommentare in blau im Template.
Umfeld	Kap. 2	Alle speziellen Aspekte des Einsatzumfeldes sollen beschrieben werden: In welcher örtlichen Umgebung, unter welchen Sicht- und Lärmverhältnissen, auf welcher Bildschirmgröße usw. Allgemeine oder vage Angaben vermeiden. Schnittstellen zu angrenzenden Systemen (Systemumfeld) möglichst konkret beschreiben. Darf auch gerne länger als ein paar Worte sein, wenn Sie mehr wissen.
Funkt. Anforde-rungen	Diagramm	Sie sollten ein (meist: nur ein) Use Case-Diagramm zeichnen. Die Namen aller nachfolgenden Use Case-(Tabellen) müssen darin vorkommen. Korrekte UML-Syntax im UC-Diagramm beachten!

	Use Cases (UC)	Das sind die UC-***Tabellen***. Use Cases werden unten noch ausführlich beschrieben. Auf die Use Cases sollten Sie größte Sorgfalt verwenden und nicht nur schnell irgendetwas hinschreiben! Es wird sinnvoll sein, bei kleinen Projekten ca. 3-10 Use Cases zu formulieren. Triviale UC mit nur 1-2 Schritten vermeiden, die bringen keine neuen Informationen. Gerne Erläuterungen, Präzisierungen und einschlägige nicht-funktionale Anforderungen, die sich auf den Use Case beziehen, direkt unter die betreffende UC-Tabelle schreiben.
Qualitäts-anforderun-gen	Kap. 4	Hier steht wenig im Template. Seien Sie ruhig kreativ und halten Sie fest, welche Qualitätsaspekte den Kunden besonders wichtig sind und wie Sie diese erreichen wollen (Pflichtenheft-Aspekt). Der Sinn ist, dass Entwickler sonst nicht wissen, worauf sie besonders achten sollen, wenn sie einen Kompromiss eingehen müssen oder wenn sie die Qualität der SW optimieren wollen.
Umsetzung	Kap. 5	Diese Angaben beziehen sich auf Constraints (Einschränkungen bei der Auswahl von Lösungen) und auf alles, was Kunden darüber vorgeben, wie die Lösung aussehen soll. Normalerweise werden Kunden eher Anforderungen formulieren, was das System können soll – aber nicht, *wie* es das tut (Lösung). In manchen Fällen gibt es aber wirklich Einschränkungen, und dafür ist dieses Kapitel gedacht. Beispielsweise kann eine Programmiersprache oder ein Framework oder sogar ein Entwicklungsprozess vorgegeben sein. Die Länge des Kapitels hängt davon ab, wie viel das ist. Typisches Beispiel: Wie soll die SW-Oberfläche aussehen? Das enthält Anforderungen und auch schon Lösungsvorschläge. In diesem Kapitel können Sie daher z.B. Oberflächen-Mockups unterbringen, also Skizzen für GUI.
Risiken	Kap. 6	Sie sollen einige wenige Risiken aufschreiben, die Sie für Ihr spezielles Projekt gefunden haben. Das zeigt den Kunden, dass Ihnen die Risiken bekannt sind, sie aber auch darüber nachdenken, wie Sie mit den Risiken umgehen. Verwenden Sie die WENN-DANN-Abhilfe Formulierung aus der Vorlesung, damit die Aspekte der Risiken (Wahrscheinlichkeit, Schaden, Gegenmaßnahme) klar erkennbar sind. Sie sollen nicht allgemeine, generische oder theoretische Risiken beschreiben, damit irgendetwas dasteht. Sondern solche Risiken, die Ihnen Sorgen bereiten oder das Projekt gefährden könnten. Höchstens fünf, mehr kann man sowieso nicht im Auge behalten.
Aufwands-reduktion	Kap. 7	Dieses Kapitel ist in unserem Template stärker ausgeprägt als in vielen Unternehmen. Es dient Ihrer Risikofürsorge: WENN Sie nicht genug Zeit haben, alle Kundenwünsche umzusetzen, DANN kann es zu Verärgerung kommen. ABHILFE: Sie legen gemeinsam mit dem Kunden schon zu Anfang fest, worauf Sie im Zweifel verzichten. Das schreiben Sie hier hin, damit sie Ihre Arbeit entsprechend planen können und der Kunde schon einmal weiß, was er vielleicht nicht bekommt. Auch darüber muss man Übereinkunft erzielen. ***Generell nehmen wir im SWP und im Template an, dass Sie in Inkrementen arbeiten***, also z.B. in drei Runden, in denen jeweils die

		wichtigsten Teile bearbeitet werden. In agilen Projekten ist das, was hier in Kap. 7 steht, durch Iterationen und andere Praktiken schon tief verankert. Hier in der Spezifikation stehen sicherheitshalber die Maßnahmen ausdrücklich dokumentiert.
Glossar	Kap. 8	Definieren Sie die wichtigsten, möglicherweise missverständlichen Begriffe und Konzepte aus der Spezifikation. Da Kunden und Entwickler die Spezifikation verstehen müssen, dient das Glossar der jeweils anderen Gruppe, spezielle Ausdrücke zu verstehen, z.B. aus dem Anwendungsbereich.
Abnahme-Testfälle	Kap. 9	Hier soll eine Tabelle von Testfällen stehen, die mit dem Kunden ausgehandelt wurde. Sie werden typischerweise von den Entwicklern aus den wichtigsten Use Cases abgeleitet. Wenn diese Testfälle ohne Fehler ablaufen, ist zumindest der Kern der Aufgabe erfüllt. Achten Sie auf vollständige Testfälle mit den Angaben: 13. Was muss vorbereitet werden (set-up)? 14. Wie sieht der Aufruf und die Parameter aus (input)? 15. Was genau sind erwartete Reaktion und Resultat (output)? Beschreiben Sie Parameter und Resultat konkret, nicht in allgemeinen Angaben (z.B. „1234“ statt „PIN“).
Weitere Aspekte	(Kap. 10 ff)	Wenn Sie noch ganz andere Angaben unterbringen wollen, die in ihrem Projekt wichtig sind, hängen Sie weitere Kapitel hinten an.

Die Spezifikation dient dazu, Kundenwünsche zu dokumentieren, so dass die Kunden sie auf Vollständigkeit und Korrektheit prüfen können. Architekten und Entwickler sollen anhand derselben Spezifikation abschätzen können, wie sie die Entwicklung vorantreiben. Dieses oberste Ziel sollten Sie immer im Auge behalten, wenn Sie an den Anforderungen schreiben. Es geht nicht darum, sinnlose Formalitäten abzuarbeiten oder rasch irgendwelche vagen Texte hinzuschreiben, sondern gute Voraussetzungen für ein erfolgreiches Projekt zu schaffen. Alle Tabellen, Checklisten und Vorlagen sollen dabei helfen.

Woher weiß man, was man in die Spezifikation schreiben soll?

Generell sind nicht die Entwickler oder Requirements Engineers aufgefordert, sich Anforderungen auszudenken oder alleine für sich Use Cases und Spezifikationen zu schreiben. Es sind vielmehr die Kunden und alle von einer geplanten Software betroffenen Personen („Stakeholder“), deren Meinung man in der Spezifikation niederschreiben sollte. Es gibt Fälle, in denen Kunden selbst einen Teil der Spezifikation schreiben, oft werden sie aber auch unterstützt durch andere.

In der Regel werden Requirements Engineers (oder Entwickler mit dieser Aufgabe) Anforderungen, Use Cases und Spezifikationen schreiben. Sie müssen also mit den Stakeholdern sprechen, um zu wissen, was sie hinschreiben sollen. Wie man die Erhebung von Anforderungen angeht, fasst man im Requirements Engineering zusammen. Das ist ein großer und wichtiger Teilbereich des Software Engineering. Er wird in SWT kurz angesprochen, kann aber aus Umfangsgründen nicht weiter vertieft werden. Dafür gibt es die Mastervorlesung „Requirements Engineering“ an der LUH.

Ein Punkt ist jedoch wichtig, wenn Sie mit Templates für Spezifikation und Use Cases arbeiten: ***Bitte denken Sie nicht, Sie müssten hier durch Nachdenken die eine richtige Formulierung finden.*** Erstens gibt es mehrere richtige Formulierungen und zweitens wird Nachdenken nicht reichen: Sie müssen in Erfahrung bringen, was Kunden und Stakeholdern wollen und brauchen.

Am besten, Sie betrachten die Templates als Interviewleitfäden und sprechen mit den geeigneten Personen alle Punkte durch. Überlegen Sie bei jedem Punkt, ob es auch Alternativen gegeben hätte und was der Unterschied wäre.

Beispiel: Mit dem Eintrag „Umfeld“ im Use Case-Template kämpfen viele Studierende. Was soll man da hinschreiben? Man könnte zum Beispiel notieren: „Wird zu Hause auf dem PC ausgeführt“. Dann wissen die Entwickler, dass sie von einem relativ großen, stationären Bildschirm ohne Blendung ausgehen können. Feine Details sind dann darstellbar. Aber stimmt das, was sagt Ihr Kunde? Was wäre eine Alternative? „Wird im Freien auf dem Smartphone ausgeführt“. In diesem Umfeld kennen Sie die Bildschirmmasse nicht; viele Formate sind möglich, aber sie werden kleiner sein als ein PC-Bildschirm. Größere Tasten und klarere Kontraste sind im Freien gefordert. Man sieht schon an diesem einen Beispiel: Kein Eintrag ist selbstverständlich. ***Es könnte immer so oder auch anders gewollt sein***. Bei der Spezifikation geht es darum, herauszufinden und aufzuschreiben, was in Ihrem Projekt gewollt ist.

Use Cases: Was das System können soll

Wer Informatik studiert, muss programmieren können. Das ist klar. Wer in einem echten Projekt mitarbeiten will, muss aber auch einige ernst gemeinte Use Cases geschrieben haben, sonst fehlt eine mindestens so wichtige Voraussetzung. Wie schwierig es ist, *gute* Use Cases zu schreiben, merkt man erst, wenn man Feedback auf die ersten Versuche bekommen hat. Dazu haben Sie in den Übungen Gelegenheit, in der Team-Aufgabe wieder. Bis zum Software-Projekt sollten Sie dann die gröbsten Fehler schon nicht mehr machen.

Weil Use Cases so wichtig und doch so unscheinbar sind, bespreche ich unten mehrere Beispiele. Das erste stammt aus der Vorlesung, die anderen aus dem „Software-Projekt (SWP)“, wo sie von Studierenden erstellt worden sind. Sie werden hier anonymisiert widergegeben, denn es geht nicht darum, andere zu kritisieren oder herauszustellen. Die Beispiele sind alle ernst gemeint, und doch kann man an jedem diskutieren, welche Teile gut gelungen sind und welche weniger.

Dabei ist kein Use Case „richtig“ und keiner ganz „falsch“. Es kommt darauf an, mit den Kunden zusammen herauszufinden, was im Use Case stehen sollte. Um das zu verdeutlichen, zeige ich zum Vorlesungsbeispiel auch mögliche Alternativen – was wäre, wenn der Kunde *etwas anderes* gewollt hätte?

Beispiel aus der Vorlesung: Überweisen am Bankomaten.

Aus Gründen der Übersichtlichkeit ist dieser Use Case in zwei Teile zerlegt; auch in der Vorlesung steht er auf zwei Folien.

UC 2	**Überweisung tätigen**	
Umfeld	Bankautomat mit Alpha-Tastatur im Foyer	
Systemgrenzen	HW und Bankensoftware werden unverändert übernommen	
Ebene	Hauptebene	
Hauptakteure	Kunde dieser Bank	
Stakeholder u. Interessen	Kunde möchte schnell und direkt vor Ort Überweisung durchführen	
	Bank möchte Aufwand für Beleglesen sparen	
Voraussetzungen	Kunde hat ein überweisungsfähiges Konto bei dieser Bank	
	Bankomat läuft und hat Verbindung zum Banksystem	
	Kunde hat sich über seine Karte und PIN identifiziert und authentifiziert	
Garantie	Am Ende der Sitzung wird unmissverständlich angezeigt, ob die Transaktion erfolgreich war	
Erfolgsfall	Die Überweisung ist abgeschickt und ins normale Banksystem übergeben	
Auslöser	Kunde wählt Aktion "Überweisung" aus	
Beschreibung	1.	Kunde wählt Aktion "Überweisung" aus
	2.	System weist darauf hin, dass mTAN erforderlich ist
	3.	Kunde bestätigt
	4.	System zeigt Formular wie auf Papier an
	5.	Kunde füllt das Formular aus
	6.	System verschickt mTAN per SMS, fragt Kunden danach
	7.	Kunde gibt mTAN ein
	8.	System übergibt Überweisung an das Bankensystem
	9.	System *bestätigt erfolgreiche Überweisung*
	10.	System beendet die Sitzung

Erweiterungen	3a.	WENN Kunde nicht bestätigt, DANN Abbruch (10)	
	5a.	WENN Pflichtfelder nicht ausgefüllt, DANN weist System auf Fehlendes, zurück zu 5	
	5b.	WENN Kunde abbricht, DANN beende Sitzung (10)	
	7a.	WENN keine gültige mTAN, DANN zurück zu 6. Sperrung nach drei Fehlversuchen	
	9a.	WENN Kunde Beleg wünscht, DANN	
		9a.1	System druckt Beleg, beendet Sitzung (10)
Technologie	In Filialen mit Fingerkuppenleser ist keine TAN nötig		

Wie immer hat der Use Case einen **Namen**, der sich aus dem Ziel des Hauptakteurs ergibt: Er oder sie möchte eine Überweisung tätigen. Außerdem hat der Use Case eine **Nummer**, hier UC 2. Die wichtigsten Use Cases sind die auf der **Hauptebene**, denn hier findet man Abläufe, um derentwillen man das System baut: Einen Bankomaten baut man, damit man Geld abheben und überweisen kann.

Der Hauptakteur ist hier der „Kunde dieser Bank“. Das ist bereits eine Einschränkung, denn es könnten ja auch Kunden *anderer* Banken eine Überweisung tätigen wollen. So, wie der Use Case hier geschrieben ist, wird das aber nicht unterstützt. Jedenfalls nicht durch diesen Use Case. Was möchte der Kunde?

Gut ist, dass für alle Stakeholder deren **Interessen** angegeben sind. Es ist klar zu sehen, dass es Stakeholder gibt, die keine Akteure sind. Das sind also Personen, die von dem System betroffen sind, das System aber nicht bedienen. Zum Beispiel hat „die Bank“ berechtigte Interessen. Entwickler, die das lesen, wissen schon einmal sicher: Es dürfen keine aufwändigen Abläufe implementiert werden.

Voraussetzungen dürfen die Entwickler dieser Software als erfüllt voraussetzen. Es ist nicht ihre Aufgabe, diese Voraussetzungen zu prüfen und ggf. Korrekturen vorzunehmen.

Vielmehr muss jedes Programm, das diesen Use Case startet, dafür sorgen, dass die Voraussetzungen erfüllt sind. Ist das nicht der Fall, muss der Use Case auch nicht funktionieren. Die Entwickler entnehmen aber den Voraussetzungen noch mehr: Sie wissen, dass sie eine Authentifizierung nicht mehr durchführen müssen, denn die ist nach Voraussetzung ja schon erfolgt. Das wollte der Kunde so; es hätte auch anders geregelt werden können, aber Use Cases sind ja ein Teil der Anforderungen.

Garantien sind heikel, weil man unter Umständen erheblichen Aufwand treiben muss, um sie in jedem Erfolgs- und Fehlerfall sicherzustellen. Zumindest bei allen Erweiterungen muss nachher die Garantie erfüllt sein. Man muss sich aber keine Gedanken über absolut unsinnige Risiken machen (z.B. wenn ein Komet die Erde trifft). Die Garantie in diesem Beispiel ist einigermaßen ausgewogen: Sie versichert, dass die Kunden nachher wissen, ob sie den Use Case noch einmal durchführen müssen. Andererseits sind dafür nur maßvoll aufwändige Vorkehrungen erforderlich. Meist kommt eine gute Garantie in Verhandlungen mit dem Kunden zustande. Wenn Sie mehr garantieren, wird es in der Entwicklung aufwändiger und teurer. Am Ende muss das der Kunde bezahlen.

Der **Auslöser** ist auch der erste Schritt in der Beschreibung, was üblich ist. Hätte auch etwas anderes der Auslöser sein können? Natürlich: Der Kunde hat gerade seinen Kontostand geprüft; oder: der Kunde hat gerade seine Karte eingesteckt und sich authentifiziert (aber keinen Knopf gedrückt). Es gibt immer viele Möglichkeiten, und der Sinn von Use Cases besteht darin herauszufinden, was der Kunde will und was gemacht werden soll. Die Entwickler sind darauf angewiesen, das zu erfahren.

Die Beschreibung hat hier 10 Schritte im Hauptszenario. Das ist ziemlich viel, am besten sind 3-7 Schritte. Wenn es weniger sind, erreicht man mit dem Use Case nicht viel. Es bringt ja nichts, einen Use Case zu schreiben, der nur einen Schritt hat: Dann weiß man immer noch nicht genauer, wie dieser eine Schritt gemacht werden soll; man wollte ja gerade Teilschritte identifizieren. Sehr lange Abläufe sollte man dagegen eher in mehrere kürzere Schrittfolgen unterteilen und daher mehrere Use Cases schreiben, die vom Ober-Use Case aufgerufen werden. Verweis auf einen anderen Use Case deutet man an, indem man dessen Namen unterstreicht (wie hier in Schritt 9). Im Use Case-Diagramm sollte der Ziel-Use Case dann per <<include>> verbunden sein. Natürlich ist ein Use Case mit 2 oder 12 Schritten nicht falsch; es zeigt sich nur aus Erfahrung, dass mittel-lange Use Case-Beschreibungen Ihren Zweck meist *besser* erfüllen.

Es gibt zu einigen Schritten Sonderfälle in den **Erweiterungen**, zu Schritt 5 sogar zwei verschiedene (5a, 5b). Hier soll jeweils beschrieben sein, unter welchen Umständen (WENN) in diesem Schritt die hier genannten Variante ausgeführt wird, und was dann zu tun ist (DANN). Hier sehr explizit: Was geschieht nachher, wie kommt man von diesem Sonderfall wieder zurück ins Hauptscenario (z.B. zurück zu Schritt 6).

Der **Technologie**-Aspekt bleibt oft leer, weil es keine technologisch bedingten Varianten gibt. Hier hat er einmal einen sinnvollen Eintrag, weil nur bei Vorhandensein eines Fingerkuppenlesers eine Authentifizierung ohne TAN möglich ist.

Beispiele aus dem SWP von Projekten Studierender

Die folgenden Use Case-Tabellen stammen aus studentischen Projekten im SWP und sind weder perfekt noch besonders fehlerhaft; betrachten Sie sie bitte einfach als authentische Beispiele. Bitte beachten Sie, dass hier **drei Felder in der Tabelle hinzugekommen sind**: Erläuterung (eine allgemeinverständliche Beschreibung, was der Use Case tut), eine Priorität und eine Verwendungshäufigkeit. Die letzten beiden Felder dienen dazu, den Use Case im Entwicklungsprozess angemessen weiterzubearbeiten. Wenn Sie in SWT-Übungen Use Cases schreiben sollen, können Sie diese drei Felder auch weglassen. Hier in SWT ist das egal, im SWP sind die Felder wichtig und daher obligatorisch.

Beispiel 1: Navigieren für Wanderer im Deister (anonymisiert aus SWP)

Der folgende Use Case ist recht gut gelungen. In allen Feldern sind sinnvolle Einträge gemacht worden. Mithilfe dieses Use Case konnte der Kunde den Entwicklern verständlich machen, wie der Ablauf sein soll. Nach der Tabelle folgen kurze Erläuterungen zu Auffälligkeiten in diesem Use Case.

Use Case 1	**Ziel auswählen**
Erläuterung	Der User drückt auf ein Ziel auf der Karte und steuert es an.
Umfeld	Auf dem Deister
Systemgrenzen	Innerhalb des Berg Gebiets
Ebene	Hauptfunktion
Vorbedingung	App ist eingeschaltet, GPS aktiviert und Geoinformationen offline verfügbar.
Mindestgarantie	Karte zeigt den eigenen Standort und Umgebung an.
Erfolgsfall	Jedes vorgegebene Ziel in unmittelbarer Umgebung wird angezeigt und die Navigation zeigt das Ziel an auf der Karte.
Stakeholder	App User: will eine möglichst einfache Bedienfläche, um möglichst einfach ein Ziel auszuwählen. Deister Betreiber: wollen ebenfalls eine einfache Auswahlmöglichkeit mit allen Zielen, um den Zweck der App zu erfüllen.
Hauptakteur	App User
Auslöser	App User klickt auf den Reiter „Karte“
Hauptszenario	1. App User klickt auf den Reiter „Karte“ 2. Die App zeigt eine Karte an mit auswählbaren Zielen in der Umgebung an 3. App User klickt auf ein als solches gekennzeichnetes Ziel. 4. Die App zeigt ein Popup, wo die Entfernung zum Ziel und weitere Buttons ersichtlich sind. 5. Der App User wählt „Navigieren“ 6. Die App wechselt in den Navigationsmodus und beginnt mit der Navigation
Erweiterungen	keine
Priorität	unverzichtbar
Verwendungshäufigkeit	regelmäßig

Die kurze **Erläuterung** dient nur dazu zu verstehen, worum es hier überhaupt geht. Dafür ist eine Zeile ausreichend, denn der Use Case steht ja im Rahmen einer ganzen Spezifikation, die das Umfeld erklärt.

Das **Umfeld** wird sehr unterschiedlich ausgefüllt: Es könnte der funktionale Umfang sein oder die Bildschirmgröße. Hier fanden die Autoren am wichtigsten, dass die Nutzung im geographischen Umfeld des Deisters („Berg Gebiet“) stattfinden soll.

Das ist interessant, weil die Internetabdeckung dort sehr schlecht ist; man muss sich also Auswege einfallen lassen. Das ist aber leider hier nicht so explizit hingeschrieben worden. Es steht aber schon in der „Mission“ des Projekts.

Die **Mindestgarantie** ist leicht einzuhalten (GPS empfängt man); um die Umgebung anzuzeigen, muss man aber schon einen Kartenausschnitt laden – weil man nicht auf Internet setzen kann. Das ist wieder eine Konsequenz mit Folgen für die Entwicklung, an die man sonst vielleicht nicht denken würde.

Bitte beachten Sie den **Erfolgsfall**; hier gab es in der Vergangenheit oft Missverständnisse, mitunter war dieser Aspekt auch als "Erfolgsgarantie" beschriftet. Das ist irreführend, denn eine Garantie für den Erfolg kann man nicht geben. Vermeiden Sie daher diese Bezeichnung. Unter Erfolgsfall soll man beschreiben, was am Schluss des Use Case erreicht ist, wenn (erfreulicherweise) der Erfolgsfall eintritt, es also keine Schwierigkeiten oder Fehleingaben gibt. Das ist gut zu wissen – aber garantieren können wir es nicht, weil eben auch ein Sonder- oder Problemfall eintreten kann.

Sehr schön: Zu jedem **Stakeholder** ist angegeben, was die besonderen Prioritäten und Wünsche sind. Wiederum dient die Information den Entwicklern, auf die relevanten Dinge zu achten. Es könnte ja statt einfacher Bedienung auch die Genauigkeit oder die Schnelligkeit der Anzeige sein. Dann müsste man andere Vorkehrungen treffen.

Auslöser und Schritt 1 sind hier identisch. Das ist üblich, aber nicht zwingend. Falls nicht, wird der Auslöser sozusagen zu Schritt 0, der vor dem ersten Schritt zu erfolgen hat.

Ungewöhnlich ist, dass es **keine Erweiterungen** (Sonderfälle, Varianten) gibt.

Für das SWP wurden die Felder der Priorität (hier: unverzichtbar) und der Verwendungshäufigkeit eingeführt, weil damit dann die Implementierungsreihenfolge gesteuert wird. Man wird die Tabelle nicht unnötig verändern, aber wenn es so triftige Gründe gibt, kann man die Tabelle erweitern. Felder zu löschen ist dagegen problematischer, weil Entwickler sich an sie gewöhnt haben.

Im **zweiten Beispiel** (ebenfalls aus einem SWP anonymisiert) gibt es dagegen mehr kritische Punkte anzumerken. Auch dieses Beispiel soll hier besprochen werden, damit man sieht, dass man auch in einem Use Case Fehler machen kann. Hier ist zum eigentlichen Use Case eine dritte Spalte rechts hinzugekommen, die die Einträge kommentiert.

Use Case 4	Gesamtsystem starten (BEISPIEL)	Hinweise und häufige Fehler
Erläuterung	Indem die Applikation gestartet wird, wird auch die Konfiguration aus verfügbaren Komponenten geladen und gestartet. Sie sind über ein Menü startbar.	Was tut der UC anschaulich und im Überblick? Schreiben Sie keine Trivialitäten! Was man im UC eindeutig sieht, muss man hier nicht erläutern. Allgemeinverständlich schreiben.
Systemgrenzen (Scope)	Gesamtsystem	Was gehört in Bezug auf diesen UC zum System, was nicht? Das können Funktionen, Plattformen, Sonderfälle, Einsatzorte sein. „Gesamtsystem“ ist dagegen keine sinnvolle Angabe: zu kurz, und hilft nicht bei der Abgrenzung. Besser: Applikationen liegen übersetzt vor.
Ebene	Hauptfunktion	Es gibt nur drei mögliche Einträge: Hauptfunktionen sind die Ebene, für die man die

		Software entwickelt. Teilfunktionen sind unselbständige Teile davon (z.B. Login) und übergeordnete Funktionen fassen mehrere Hauptfunktionen zusammen.
Vorbedingung	Zumindest eine Komponente ist in der Applikation integriert	Was muss erfüllt sein, damit der UC sinnvoll ablaufen kann?
Mindest-garantie	Menü mit mindestens einer Komponente wird angezeigt	Je mehr Sie garantieren, desto mehr Aufwand müssen Sie treiben, um es unter allen Umständen zu erreichen (auch bei Sonderfällen, siehe unten). Diese Mindestgarantie ist gut: Sie ist klar und umsetzbar. Es wird z.B. nicht garantiert, dass der Erfolgsfall eintritt – das wäre zu aufwändig.
Erfolgsfall	Ein Menü wird angezeigt, das der Konfiguration entspricht und von dem aus die verfügbaren Komponenten startbar sind.	Hier knapp charakterisieren, unter welchen Umständen der UC erfolgreich abgelaufen ist.
Stakeholder und Interessen	Systembediener	Das ist nur ein Stakeholder, vermutlich wurden andere vergessen. Ebenso fehlen die Interessen. Der Systembediener möchte z.B. das System sehr einfach starten können. Oder, Alternative, das System soll schnell starten. Diese wichtige Information fehlt hier.
Hauptakteur	Systembediener	Einer der Stakeholder ist auch Hauptakteur. Der UC soll ein Ziel des Hauptakteurs erreichen.
Auslöser	Systembediener doppelklickt auf das Systemlogo	Hier gibt es immer verschiedene Möglichkeiten, daher diesen Punkt genau klären! Es könnte ja auch sein, dass ein Rahmenprogramm eine Funktion auslöst. Oder dass sich der Hauptakteuer ins Programm einloggt.
Hauptszenario	1. Systembediener doppelklickt auf das Systemlogo 2. Applikation liest Konfigurationsdatei ein 3. Applikation zeigt Menü an, das der Konfiguration entspricht 4. Applikation <u>wartet</u>	Häufig ist der Auslöser auch Schritt 1. Danach wechseln sich meist Aktivitäten eines Akteurs und des Systems ab. Schreiben Sie sie in der Form <wer> <tut was> auf. Unter-UCs (wie hier „wartet“) werden unterstrichen. Sie funktionieren wie ein Unterprogramm-Aufruf. Im UC-Diagramm muss es einen <<include>>-Pfeil von UC1 auf UC „wartet“ geben.
Erweiterungen	2a: WENN Konfigurationsdatei nicht korrekt, DANN zeige alle Komponenten als schlichte Liste an	Es gibt keine Verzweigungen oder Schleifen im Hauptszenario. Alle Abweichungen von einer einfachen Sequenz werden als „Erweiterungen“

		ausgedrückt. Man bezieht sich auf den Schritt, in dem die Abweichung erfolgt, hier 2a. Dann wird das wie ein Spezialszenario beschrieben. Am Ende sollte man wieder im Hauptszenario landen. Das steht hier nicht, daher ist wohl gemeint, dass es nach Schritt 2 weitergeht.
Priorität	unverzichtbar	Das Feld ist im SWP obligatorisch.
Verwendungshäufigkeit	regelmäßig	Es war für dieses Projekt wichtig zu wissen, ob eine Funktion nur sehr selten oder ständig ausgeführt wird – das hatte Auswirkungen auf die Programmierung.

Im **dritten Beispiel** wird ein Turnierplan erstellt (ebenfalls anonymisiert aus SWP) und hier eine Mannschaftsliste für das Turnier erstellt. Wieder gehe ich unter der Tabelle auf Besonderheiten ein.

Use Case 3	**Mannschaftsliste erstellen**
Erläuterung	Der Benutzer erstellt eine Mannschaftsliste durch das Hinzufügen von einzelnen Mannschaften und speichert diese eventuell in eine Datei
Status	geplant
Systemgrenzen (Scope)	Software für den Turnierplaner
Ebene	Hauptaufgabe
Vorbedingung	Das Programm wurde gestartet und vorangegangene Turnier- und Mannschaftsinformationen wurden bereits eingegeben
Mindestgarantie	Fehlermeldung mit Verweis auf inkorrekte Eingabe oder erfolgreicher Abschluss
Erfolgsfall	Eine Mannschaftsliste wurde erfolgreich erstellt und gegebenenfalls gespeichert
Stakeholder	Benutzer, Volleyballverein, Spieler
Hauptakteur	Benutzer
Auslöser	Benutzer startet Mannschaftsauswahl
Hauptszenario	1. Programm öffnet Mannschaftsauswahlfenster 2. Benutzer gibt den Namen einer Mannschaft ein 3. Benutzer drückt „Mannschaft hinzufügen“ 4. Programm fügt die Mannschaft zur Liste hinzu 5. Schritt 2 bis 4 werden solange wiederholt, bis alle Mannschaften in der Liste eingetragen wurden 6. Benutzer beendet die Mannschaftseingabe mit „Weiter“
Erweiterungen	3a: WENN der Mannschaftsname nicht den Vorgaben entspricht, DANN öffnet sich ein Fenster, das den Beutzer auf den Fehler hinweist 5a: WENN Benutzer eine Mannschaft aus der Liste markiert, DANN kann er mit „Mannschaft entfernen“ die Mannschaft aus der Liste löschen. 5b: WENN Benutzer „Mannschaftsliste laden“ drückt, DANN öffnet sich ein Fenster in dem er eine Datei mit einer gespeicherten Mannschaftsliste auswählen kann um die Liste zu laden 5c: WENN Benutzer auf „Mannschaftsliste speichern“ drückt DANN öffnet sich ein Fenster in dem der Benutzer die Datei zum speichern auswählt oder eine neue Datei erstellt 5d: WENN Benutzer auf abbrechen drückt DANN zurück zum vorherigen Fenster
Priorität	unverzichtbar
Verwendungshäufigkeit	regelmäßig

Einzelne Projekte im SWP verwendeten eine **Statusangabe**. Das ist zwar möglich, widerspricht aber der dortigen Vorgehensweise. Eigentlich soll ein Use Case beschreiben, welcher Ablauf möglich sein soll. Durch Priorität und Verwendungshäufigkeit kann man dann planen, welche Use Cases in welcher Reihenfolge implementiert werden. Aber wenn man während der Implementierung dauernd den Status aktualisieren müsste, der ja in der Spezifikation steht, dann wäre das unangemessen. Diesen Eintrag sollte man also nicht in den UC aufnehmen, ***diese Erweiterung ist nicht sinnvoll***.

Die Angabe unter **Systemgrenzen** bringt hier nicht viel zusätzliche Informationen. Es ist nicht klar, was innerhalb der Grenzen und was außerhalb ist, darum geht es aber bei den Grenzen. Auch die Angaben

zu den Stakeholdern sind zwar offensichtlich korrekt, aber es fehlen die wichtigen Informationen, was deren Wünsche und Prioritäten sind.

Interessant sind hier das Hauptszenario und die zahlreichen Erweiterungen. Alle Erweiterungen außer 3a beziehen sich auf Schritt 5 des Hauptszenarios; es sind alternative Fälle, sie treten also nicht alle nacheinander auf, sondern entweder 5a oder 5b oder 5c oder 5d. Alle Erweiterungen sind gut formuliert: Sie geben klar an, unter welchen Umständen (WENN) welche Konsequenz (DANN) möglich ist, die vom Hauptszenario abweicht. Oft beschreibt man damit Reaktionen auf Fehler; es ist aber auch möglich, Varianten und Fallunterscheidungen zu beschreiben, sofern diese eher Ausnahmefälle sind. Wenn es keine Ausnahmen sind, sollte man stattdessen eigene Use Cases formulieren.

Hier fällt auf, dass nicht explizit gesagt wird, wie es nach den Erweiterungen weitergeht. Das wäre noch gut, denn eine Erweiterung soll sagen, wie man aus einem Sonderfall wieder zurück ins Hauptszenario kommt: Beispielsweise gibt man eine PIN noch einmal ein und landet dann im Folgeschritt. Hier weiß man nicht genau, in welchem Schritt man landen soll. Andererseits gibt es nur noch Schritt 6, also ist wohl der gemeint? Oder doch ein Rücksprung? Das sollte man explizit sagen.

Abnahmetestfälle: eine Hilfe für Entwickler

Abnahmetests sind zunächst einmal normale Testfälle. Sie müssen also set-up, input und erwarteten output für jeden Testfall enthalten (siehe oben). Alle diese Angaben müssen konkret und nachprüfbar sein. Normalerweise wird man versuchen, Testfälle automatisch zu prüfen, sie also in einem Testrahmen auszuführen. Dafür braucht man natürlich ganz konkrete Soll-Werte für die Ergebnisse, keine allgemeinen Beschreibungen. Es darf nicht „Name“ im Testfall stehen, sondern „Pia Maier“ oder „Paul Meier“. Welche konkreten Werte oder Namen Sie in die Abnahmetests aufnehmen, zeigt, wie gut Sie testen können. Für die Abnahmetests sollten Sie sich merken: Typische Werte, die wahrscheinlich oft in diesem Zusammenhang auftreten, sind besser geeignet als ausgedachte und ungewöhnliche (wie „Paul Karl von der Hüttenburg-Strassnitz“).

Der wichtigste Unterschied von normalen Tests und Abnahmetests liegt in ihrer Anwendung: Bei normalen Tests wollen Sie Fehler finden, um sie zu bereinigen. Im Abnahmetest hoffen Sie, alle Fehler längst beseitigt zu haben und mit den Kunden festzustellen: Die allerwichtigsten Funktionen werden von der Software korrekt bewältigt.

Dabei kann man bei kleinen Projekten die wichtigsten Use Cases verwenden und dafür jeweils ein oder zwei konkrete Ausprägungen als Testfälle formulieren. Dazu vielleicht noch einige wenige Sonderfälle, die der Kunde wichtig findet (z.B. weil er eben doch auch längere Namen speichern will). Natürlich kann man leichter solche konkreten Testfälle erfüllen als eine ganze Spezifikation. Daher ist es umso wichtiger für die Entwicklerseite, diese Abnahmetests immer wieder einmal durchzuführen. Wenn die Software ihre Abnahmetests nicht schon mehrfach und auch mit leichten Varianten erfolgreich ausgeführt hat, sollte man keinesfalls den Abnahmetermin wagen: Dann kann es nur schief gehen und wird sehr viel Ärger verursachen. Denn Kunden denken sich zu Recht: Wenn nicht einmal diese bekannten sieben Testfälle funktionieren, was ist dann noch alles faul am System? Es wäre unverzeihlich, sich als Projektteam so zu präsentieren.

Umgang mit der Spezifikation: Baseline oder Änderungen zulassen?

Diese Vorlesung ist nicht auf die traditionelle Entwicklung oder auf die agilen Methoden beschränkt. Heute müssen Sie beides können, und auch Mischformen kommen immer häufiger vor.

In ***traditionellen Ansätzen*** ist die Spezifikation nicht nur der Bezugspunkt für Kunden und Entwickler. Häufig ist sie auch Vertragsbestandteil und kann folglich kaum geändert werden. In dieser Situation sind beide Seiten daran gebunden, die einmal niedergeschriebenen Anforderungen Wort für Wort zu

erfüllen (Entwickler) und das Ergebnis dann auch abzunehmen (Kunden) – selbst, wenn sich inzwischen etwas geändert hat oder ein Missverständnis erkannt wurde. Da das sehr unbefriedigend ist, verwendet man viel Zeit und Aufwand auf die Anforderungsklärung und das Schreiben der Spezifikation. Alles soll sauber und korrekt sein. Änderungen sind unmöglich oder sehr aufwändig.

Eine Grundbotschaft der ***agilen Ansätze*** war: Es wird dennoch nicht gelingen, so eine Spezifikation ein für alle Mal korrekt aufzuschreiben. Es gibt immer irgendwelche Änderungen. In der Regel ist eine umfangreiche Spezifikation schon veraltet, wenn sie endlich fertig ist. Daher vermeidet man in agilen Methoden Spezifikationen und arbeitet lieber mit einem groben Masterplan und mit einer größeren Zahl von User Stories. In agilen Projekten wird üblicherweise eine Hierarchie von Begriffen und kurzen Beschreibungen verwendet:

Epic ist eine größere inhaltliche Einheit. Dort geht es zum Beispiel um einen Teilbereich eines Flugbuchungssystems, das sich mit der Meilengutschrift beschäftigt. Alles was dazu gehört, hängt auf einer Übersichtstafel (Agile Task Board) unter dem Namen der Epic. Das kann dann die Erfassung der Personalie, Anlegen eines Meilenkontos usw. sein. Diese Bestandteile heißen User Stories, die Karten, auf denen sie stehen, heißen Story Cards. Jede Story Card zeigt eine kleine, in sich abgeschlossene Funktion, die dem Kunden Nutzen bringt. Falls erforderlich, können User Stories noch einmal in Tasks zerlegt werden. Die sind so feingranular, dass der Kunde sich nicht mehr dafür interessiert – während Epics und User Stories ständig mit den Kunden besprochen und neu priorisiert werden. Auch für User Stories gibt es in Werkzeugen (z.B. JIRA) wieder Templates für eine gute Formulierung. Ein Beispiel dafür ist: "As a < type of user >, I want < some goal > so that < some reason >"."As a < type of user >, I want < some goal > so that < some reason >". Man muss also alle in <Klammern> gesetzten, nicht-terminalen Teile durch konkrete Ausdrücke ersetzen. Dadurch sollen Formulierungen entstehen, die andere Entwickler gut verstehen können.

In Unternehmen ist oft keines der beiden Extreme geeignet: Man braucht ein kurzes Schriftstück auf das man sich zwischen Kunden und Entwicklung einigt; der Überblick über System und Anwendung muss einmal erfragt und formuliert werden, damit neue Entwickler eine Chance haben, den Zusammenhang ihrer Arbeit zu erkennen. Kunden und Stakeholder erhalten durch eine kurze Spezifikation das Gefühl, die Entwicklerfirma habe verstanden, worum es geht. Eine große Zahl von User Stories alleine sind dafür viel zu kleinteilig. An einem gewissen Punkt überwiegt aber das Bedürfnis nach Flexibilität bei Änderungen, die Möglichkeit, über Prioritäten zu sprechen, noch eine Funktion hinzuzunehmen usw. Sobald es so weit ist, ist es besser, mit User Stories iterativ zu arbeiten. Das Schwierige ist der Übergang von der Spezifikation zu den flexibleren agilen Formaten. Wir tun genau das auch im SWP, weil es in den Projekten einfach nötig ist und in Unternehmen ebenfalls vorkommt.

Zusammengefasst: Je nachdem, wie ein Projekt mit der Spezifikation umgeht, muss man mehr oder weniger Zeit und Aufwand investieren, um sie zu schreiben. Verschiedene Formen sind möglich und können sinnvoll sein. ***Im SWP*** benutzen wir ein hybrides Verfahren, das für diese Art von Projekten optimiert wurde: Wir starten mit einer kurzen Spezifikation, die durch ein Template unterstützt wird; es ist fast identisch mit dem Template dieser Vorlesung. Die Spezifikation dient Kunden und Entwicklern zum Gesamtüberblick, zur Abstimmung und zum Aufbau eines Vertrauensverhältnisses zwischen Kunden- und Entwicklerseite.

Nach nur einem Monat wird die Spezifikation vom Kunden unterschrieben und damit abgenommen. ***Jetzt ändert sich ihre Rolle***: Von diesem Punkt an wird sie nicht mehr aktualisiert, nur noch die Abnahmetestfälle werden bei Veränderungen mitgeführt. Die Spezifikation wird zerlegt in User

Stories: Aus der Spezifikation, speziell den Use Cases, werden User Stories abgeleitet (siehe eigenes Thema). In den restlichen drei Monaten wird jede Woche interaktiv mit den Kunden besprochen, ob und was sich ggf. verändert hat; nebenbei wird implementiert. Das wirkt auf Unerfahrene manchmal wie unnötiger Aufwand. Möchte man aber einen klaren Bezugspunkt und flexible Weiterarbeit, kommt man um so einen Übergang nicht herum. Die Spezifikation ist wichtig, und zwar für den Start des Projekts. Dann geht sie über in User Stories. Die meisten Teile der Spezifikation bieten aber bis zum Projektende noch einen guten Überblick. Es ist wichtig, dass auch die Kunden verstehen, wie die Spezifikation verwendet wird. Die Spezifikation ist zur Anforderungsklärung wichtig; danach sind Änderungen über User Stories erlaubt und üblich. Die Spezifikation ist ein Hilfsmittel, aber kein Vertrag. Sie ist maßgeschneidert für das SWP; viele Unternehmen gehen ähnlich vor.

Vertiefung: UML und Design Patterns

Wenn man ein Programm entwickelt, möchte man seine Struktur festlegen, ohne dabei auf Codebasis diskutieren zu müssen. Das ist besonders wichtig, wenn viele Personen oder sogar mehrere, verteilt arbeitende Teams an der Entwicklung beteiligt sind: Sie müssen sich vor der Implementierung und für die gemeinsame Entwicklung abstimmen. Dazu hat man schon immer grafische Sprachen mit Kästen und Linien verwendet, allerdings war lange Zeit keine einheitliche Sprache verfügbar. Je nach verwendeter Entwicklungsmethode wurden ähnliche, aber verschieden aussehende Sprachen eingesetzt. Erst durch die Entwicklung der Unified Modeling Language (UML) gibt es jetzt eine weltweit bekannte und einheitlich definierte Sprache zur Darstellung von Programmstrukturen und -entwürfen.

Man merkt dieser Sprache UML an, dass sie durch Kombination mehrerer Vorgängersprachen entstanden ist, die nicht hundertprozentig zusammenpassen und sich teilweise auch überlappen. Im Endeffekt wurden die Ansätze nach Booch, Rumbaugh und Jacobson kombiniert und zur UML erklärt. Da diese drei Entwicklungsmethoden weit verbreitet waren, hat sich auch UML schnell verbreitet und ist heute der Quasi-Standard für die Beschreibung von Softwareentwürfen und -architektur. Durch diese Geschichte und die drei Ursprungssprachen bietet UML nicht immer nur eine Möglichkeit, um einen Sachverhalt auszudrücken. Oft gibt es mehrere Symbole zur Wahl, und selbst manche der Diagramme in der UML sind gegeneinander austauschbar. Teilweise ist es Geschmackssache, ob man ein Sequenz- oder ein Kollaborationsdiagramm verwendet. Es gibt entsprechend auch Kritik an der UML, aber die Vorteile überwiegen doch die Nachteile. Im Rahmen der Vorlesung konzentrieren wir uns auf die Ausdrucksfähigkeiten und die guten Einsatzmöglichkeiten der Sprache.

Im Rahmen dieser Vorlesung sollen Sie lernen, die wichtigsten UML-Diagrammarten zu erkennen und zu verstehen. Sie sollen sehen und zu schätzen lernen, dass es unterschiedlich ausführliche UML-Diagramme gibt und wozu sie dienen. Außerdem sollen Sie in der Lage sein, Ihre Entwurfsideen korrekt in UML zu formulieren, damit andere sie gut verstehen können. Da es verschiedene Versionen der UML gibt, legen wir uns für die Vorlesung auf genau diejenigen Sprachmittel fest, die Sie im „UML-Poster“ zu dieser Vorlesung finden. Das ist die Referenz für Übungen, Teamaufgaben und auch die Klausur. Mehr müssen Sie nicht beherrschen, diesen Teil sollten Sie aber korrekt einsetzen.

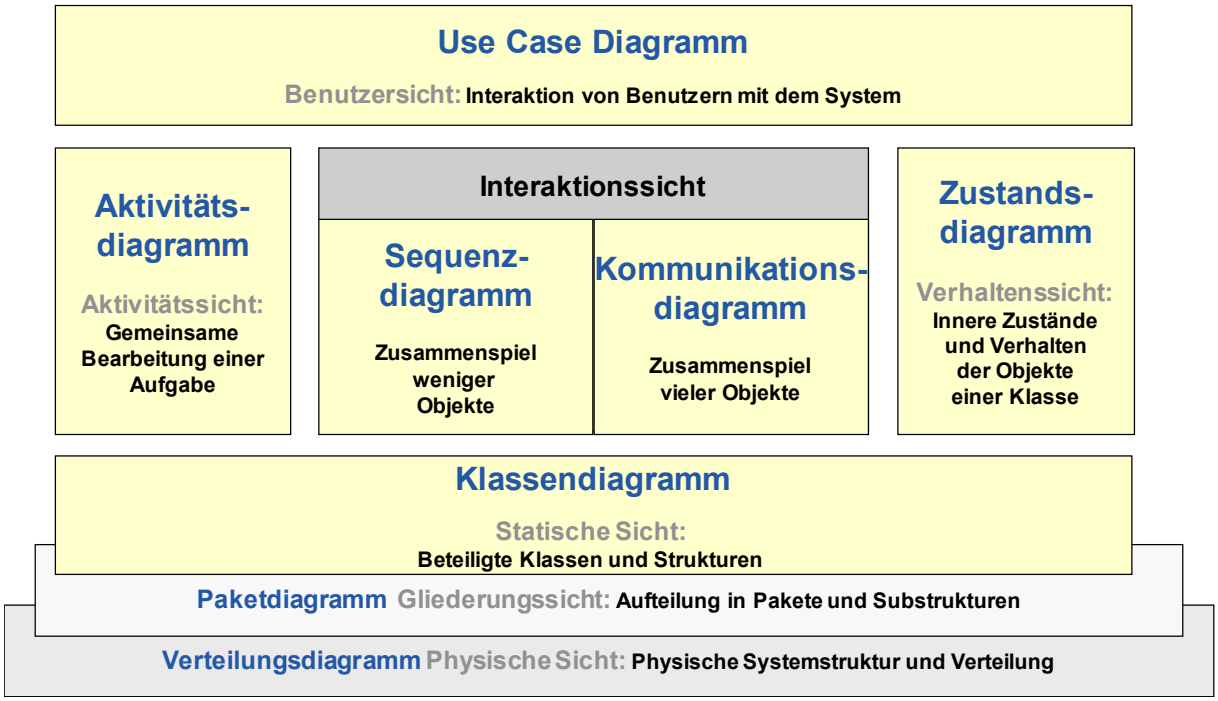
Weiter unten werden dann Design Patterns eingeführt. In fast jedem Design Pattern kommen kleine Ausschnitte aus UML-Diagrammen vor, die dort Lösungsmuster sind. Insofern sind Design Pattern Anwendungsfälle für UML-Diagrammausschnitte. Darüber hinaus sind Design Patterns aber auch gute Mittel, um bekannte Aufgaben so gut wie möglich zu lösen. Das wird unten ausführlich erläutert.

UML: Grundlagen

Ausdrücklich sei auf die Folien verwiesen. Eine grafische Sprache kann man sich am besten anhand grafischer Beispiele vorstellen, wie sie in der Vorlesung vorkommen. Sie können und sollen hier nicht noch einmal wiederholt werden. Hier wird stattdessen erläutert, worauf es besonders ankommt.

Diagrammarten: Ein UML-Modell besteht nach Definition aus mindestens einem Klassendiagramm und möglicherweise einigen weiteren Diagrammen, wie Sequenz- oder Aktivitätsdiagrammen. Dem Klassendiagramm kommt dabei eine besondere Rolle zu. Was dort nicht zu sehen ist, das kann auch in keinem anderen Diagramm auftreten; Objekte auf irgendeinem Diagramm müssen zu einer Klasse gehören, die auf dem Klassendiagramm zu sehen ist. Wenn Sie also beschreiben wollen, dass ein Account geöffnet wird, dann muss es im Klassendiagramm einen entsprechende Klasse *Account* geben. Alle Diagramme, die zu einem UML-Modell gehören, können sich ein gemeinsames Klassendiagramm teilen.

Für die Vorlesung brauchen Sie nicht alle ca. 13 Diagrammarten zu kennen. Es reicht aus, wenn sie etwa die Hälfte verstehen und erstellen können, nämlich genau diejenigen, die in der Vorlesung vorgekommen sind.



Wieso sind Klassendiagramme so unterschiedlich ausführlich? Die Entwicklung komplizierter Software geht schrittweise und oft iterativ vonstatten. Man hat zunächst nur eine grundlegende Idee, denkt an einige Klassen, hat sich aber über deren Unterklassen, Attribute oder Methoden noch keine Gedanken gemacht. Später weiß man zwar, welche Methoden eine Klasse braucht, um die nötigen Interaktionen und Use Cases umzusetzen, möchte sich aber noch nicht mit deren Parametern, deren Typen oder Sichtbarkeit belasten. Dementsprechend kann man UML-Klassendiagramme auf verschiedenen Entwicklungs- und Detailstufen zeichnen. Alle sind korrekt; ob sie hilfreich sind, hängt davon ab, in welchem Entwicklungsstadium man sich befindet. In frühen Stadien werden grobe Strukturen definiert und diskutiert. Da würden zu viele Details nur schaden. Später dienen die detaillierteren UML-Modelle den Entwicklern als Vorlage für die Programmierung. Im Laufe der Entwicklung wird man immer mehr Details hinzufügen. Dabei ist es üblich, innerhalb eines Diagramms zu jedem Zeitpunkt einen Detailgrad konsequent durchzuziehen, also entweder überall Parameter anzuzeigen oder bei keiner Methode. Die schrittweise Vervollständigung gilt dabei nicht nur für Klassen, sondern auch für die Assoziationen zwischen Klassen: Zuerst ist es schon ausreichend, überhaupt Assoziationen einzuzeichnen. Später wird man Multiplizitäten, Variablenbezeichner usw. hinzufügen. Die UML wächst mit dem Wissen über das neue Softwaresystem. Das ist eine große Stärke und unterstützt einen iterativen Entwicklungsprozess, wie er heute weit verbreitet ist.

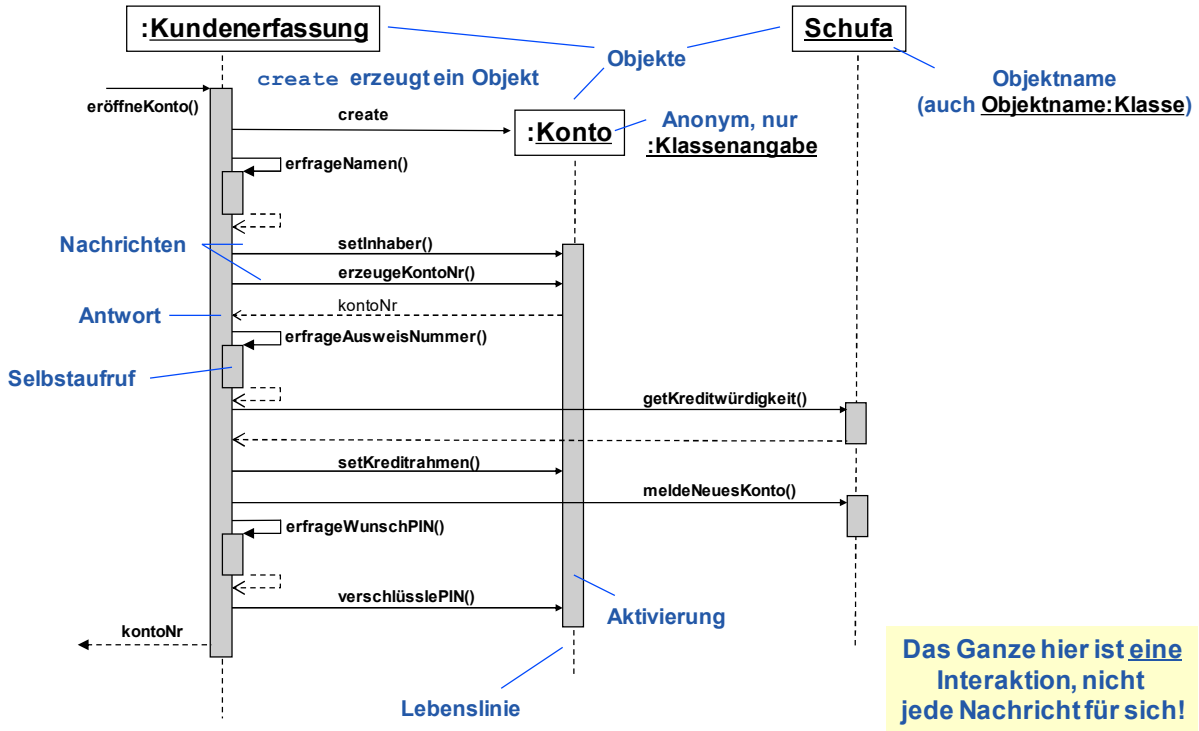
Einige Werkzeuge (auch die in der Vorlesung genannten, kostenlosen) können aus UML-Diagrammen Code generieren. Das erfordert natürlich, alle Angaben (Bezeichner, Typen, Sichtbarkeiten), die im Generat stehen sollen, auch in den Diagrammen zu zeigen: Wenn Sie im Diagramm keine Parametertypen angeben, können sie auch nicht im generierten Code auftauchen. Codegenerierung geht über den Rahmen der Vorlesung hinaus; Sie sollten aber immer abwägen, ob es sich lohnt, so viele Details in die Diagramme zu stecken. Das macht sie unübersichtlicher. Es könnte besser sein, nur die Grundstrukturen für die Generierung zu verwenden und dann auf Codeseite weiter auszufüllen.

Eigenschaften wichtiger Diagrammarten

Klassendiagramme wurden oben schon besprochen. Sie beschreiben die Statik einer Software, also die Klassen, Beziehungen und ableitbaren Objekte. Eine kleine Spitzfindigkeit: Methoden sind in UML durch ihre Namen und (zumindest) ein Klammerpaar gekennzeichnet. Das erinnert sehr an die Klammern () hinter Java-Methoden. Doch UML ist nicht nur auf Java anwendbar, sondern auf alle objektorientierten Sprachen, also auch auf C++, Smalltalk und viele mehr. Daher sind es, streng-genommen, UML-Klammern, nicht Java-Klammern. Aus einem UML-Modell können unterschiedliche Sprachen erzeugt werden.

Wie die Klassen bzw. ihre Objekte miteinander interagieren, ist in Sequenz-, Objekt- und Aktivitätsdiagrammen geregelt. Wichtig ist die Unterscheidung zwischen Klassen und Objekten. Beide können als Rechtecke dargestellt werden, was etwas missverständlich ist. Damit man sie eindeutig unterscheiden kann, werden Objektnamen grundsätzlich unterstrichen; Klassennamen nicht. Diese Konvention gilt in allen UML-Diagrammtypen.

Sequenzdiagramme zeigen (mit nur sehr wenigen Ausnahmen) stets die Interaktion zwischen *Objekten*. Deren Namen sind daher unterstrichen. Aber von welchen Klassen sind das die Ausprägungen? Hier ist die Konvention, vor dem Doppelpunkt : jeweils den Objektnamen, dahinter den Klassennamen zu notieren. Auf den Folien tritt das Objekt *Schufa* auf, dessen Name keinen Doppelpunkt hat. Per Konvention ist es damit ein Objektname, von dem man vielleicht zuerst gar nicht wusste, wie die zugehörige Klasse heißt. Möchte man dagegen ein anonymes Objekt der Klasse A notieren, so hat das umgekehrt keinen Objektnamen, aber den Klassennamen A. Man schreibt :A. Hier im Beispiel: ein anonymes Konto hat keinen eigenen Namen, ist also mit :Konto notiert. Ein Unterstrich, also ein Objekt. Kein Objektname vor dem Doppelpunkt, aber der Klassenname dahinter (A). Auch diese Namenskonvention gilt in allen UML-Diagrammen.



Zu einem Sequenzdiagramm gehören die genannten Objekte und darunter jeweils eine Lebenslinie (gestrichelt). Sie zeigt an, wie lange das Objekt existiert. Objekte können innerhalb eines Sequenzdiagramms angelegt und zerstört werden, dann beginnt oder endet die Lebenslinie. Die dicken, grauen Balken sind dagegen Aktivierungsblöcke. Sie zeigen an, wie lange das Objekt im Sinne

der gewählten Sequenz aktiv ist. Ein Objekt kann zwischendurch inaktiv werden und dann wieder aktiviert; das ist bei der Schufa so, die nicht auf eine Anwendung wartet, sondern inzwischen etwas Anderes tut. Bezeichner, Lebens- und Aktivierungslinien sind Vorgaben für die Entwickler, die dann das Programm entsprechend schreiben sollen.

Sequenzen ergeben sich zum Beispiel ziemlich direkt aus Use Cases, wo das Hauptszenario genau zu einer Sequenz weiterentwickelt werden kann. Im Sequenzdiagramm wird aber nicht nur das Hin und Her zwischen dem Benutzer und dem System als Ganzem dargestellt, sondern es wird gezeigt, welche Teile (Objekte) die Nachrichten des Benutzers erhalten und wie sie darauf reagieren.

Bitte machen Sie sich klar, dass die meisten Use Cases auf unterschiedliche Weise in Sequenzdiagramme und auch in Software umgesetzt werden können! Es ist der kreative Beitrag von Designern, festzulegen, welches Objekt sich worum kümmern soll. Auch UML-Diagramme sind wieder ein Schritt voran zu laufender Software, der mehr Informationen enthält als der vorherige Schritt (Use Cases). Wenn Sie solche Diagramme zeichnen, nehmen Sie wichtige Entscheidungen vor, nach denen sich die Programmierer richten müssen. Sie entwickeln Software.

Kommunikationsdiagramme sind ähnlich ausdrucksmächtig wie Sequenzdiagramme. Es ist in gewissem Sinne Geschmackssache, welches Diagramm Sie bevorzugen. Dabei gibt es aber einige Unterschiede in der Darstellung zu bedenken: Wenn sehr viele Objekte relativ wenige Nachrichten austauschen, bringen Sie das in einem Kommunikationsdiagramm leichter unter. Komplizierte Abfolgen zwischen wenigen Objekten sehen dagegen in Sequenzdiagrammen übersichtlicher aus. Lesen und schreiben müssen Sie beide Sorten von Diagrammen können.

Unterschätzen Sie nicht ***Kommentare in UML***: Sie sind in allen Diagrammarten erlaubt und werden durch ein Dokument mit umgeknicktem Ohr dargestellt. Von diesem Symbol geht eine gestrichelte Linie zu genau dem Punkt, der kommentiert wird. Das kann ein Objektsymbol oder eine Assoziation oder auch eine Methode sein. Manchmal stehen wichtige Einschränkungen oder Zeitbeziehungen in Kommentaren, oder sogar Algorithmen. Diese Kommentare sind direkte Botschaften an die Entwickler.

Wie ein UML-Modell entsteht

Wie erstellt man ein UML-Modell? UML ist nur eine Notation, keine Entwicklungsmethode. Man kann UML also sehr verschieden verwenden. Es hat sich jedoch bewährt, nach Abschluss der Anforderungserhebung mehrere Architektur- oder Entwurfsvarianten zu erarbeiten und diese mit (eher groben) UML-Modellen auszudrücken. Sie zeichnen also einige zentrale Sequenzdiagramme zu den Use Cases und übertragen dann alle benötigten Klassen bzw. Objekte in ein Klassendiagramm. Dann kann man die Entwürfe diskutieren und vergleichen, auch erweitern und weiterentwickeln. So werden sie detaillierter. Meist arbeitet man intensiv an einer Stelle und vergleicht nachher, ob jetzt in anderen Teilen des Modells eine Inkonsistenz besteht. So wird das Modell immer detaillierter, vollständiger, und besser diskutiert.

Information Hiding ist nicht ausdrücklich Teil dieser Vertiefung. Sie ist aber ein so fundamentales Konzept für den Entwurf, dass sie auch in Design Patterns gebraucht und häufig in UML-Diagrammen ausgedrückt wird. Was man sich generell über Information Hiding merken kann ist, dass die Schnittstellen einer Klasse, eines Moduls oder Programms besonders wichtig sind. Denn wer ein Programm benutzen will, muss die Schnittstellen kennen. Die Interna dagegen nicht. Das ist eine Erleichterung, man muss nicht so viel lernen und wissen.

Design Patterns

Design Patterns sind Muster für das Softwaredesign. Bekannte, immer wieder auftauchende Fragestellungen („Problems“) sollen auf bekannte, bewährte Lösungen („Solutions“) abgebildet werden. Es gäbe jeweils auch andere Lösungsmöglichkeiten, aber die jahrelange Erfahrung vieler Entwickler führt zu einer Empfehlung für die Lösung. Das Paar aus Problem und Solution macht im Wesentlichen das Muster/Pattern aus. Zu der Beschreibung eines Musters gehören noch zahlreiche Erklärungen und Klassifikationen. Den Kern bilden aber eine Problembeschreibung und eine Lösungsbeschreibung. Da es sich um Entwurfsmuster handelt, sind die Lösungsbeschreibungen oft in UML gehalten. Je nach Art des Musters werden die dazu passenden Diagrammarten gewählt.

Es gibt einige Konzepte in der Informatik, von denen man noch sagen kann, wer sie entwickelt oder populär gemacht hat. Bei Design Patterns ist das ein Buch: „Design Patterns“ von Erich Gamma und seinen drei Kollegen; sie werden teilweise liebevoll die „Gang of Four“ genannt. Obwohl es vorher die Konzepte sicher auch schon gab, hatten sie keinen klaren Namen und waren vielen professionellen Softwareentwicklern nicht bewusst oder bekannt. Heute zählt das Design-Pattern-Buch zu den meistgekauften der Informatik. Heute darf man erwarten, dass professionelle Entwickler die Idee der Design Patterns kennen; und einige konkrete Patterns aus dem Gamma-Buch ebenfalls. Das ist auch das Ziel dieser Vorlesung.

Viele Design Patterns sind einfach

Wer das liest und noch nie ein Design Pattern gesehen hat, erwartet nun vielleicht große Architekturprobleme, die Novizen gar nicht lösen könnten. Das ist aber nicht der Fall. Die rund 20 Design Patterns im Gamma-Buch behandeln kleine Alltagssituationen, die dafür aber häufig vorkommen. Studierende sollten nach den ersten Programmierkursen in der Lage sein, die Alltagsaufgaben „irgendwie“ zu lösen, so dass sie „funktionieren“. Aber dann kommt die Erfahrung von Gamma et al.: Es gibt innerhalb der irgendwie-funkionierenden Lösungen eine, die sich mehr bewährt hat als die anderen, und die daher als Design Pattern empfohlen wird. Am besten sieht man das an den drei einfachen Beispielen, die in der Vorlesung vorkommen:

Adapter: Eine Klasse K bietet eine bestimmte Methode nicht an, die aber von einem großen Softwaresystem aufgerufen werden soll. Was tun? Das große System könnte eine Fallunterscheidung enthalten und für diesen bestimmten Fall aus anderen Methoden der Klasse K die gewünschte Leistung zusammensetzen. Oder man könnte statt K eine neue Klasse schreiben. Das Design Pattern Adapter sagt aber stattdessen: Schreibe zu der Klasse K eine Adapter-Klasse, die die gewünschte Methode M enthält. Innerhalb dieses Adapters und innerhalb der Methode M übersetzt nun der Adapter, was K tun muss, um die Leistung der Methode zu erbringen. Es wird also in K (nicht im Anwendungssystem!) die Methode zusätzlich implementiert. Das Anwendungssystem muss dann statt K den Adapter aufrufen, denn nur der hat die Methode M. Was innerhalb dieser Methode geschieht, kann dem Anwendungssystem egal sein (Information Hiding).

Composite: In einem Anwendungssystem werden verschiedene Objekte mit den immergleichen Methoden bearbeitet: Sie werden gelöscht, verschoben, modifiziert. Beispielsweise tut das ein grafischer Editor mit den gezeichneten Symbolen: auf der Oberfläche platzieren, Farbe geben, vergrößern, schieben, löschen. Nun kommt jemand auf die Idee, mehrere solche Objekte zu einer zusammengesetzten Einheit zu verbinden: Ein Haus besteht aus mehreren Strichen. Man möchte nun auch das Haus als Ganzes verkleinern, verschieben usw. Was tut man also? Der Editor könnte immer, wenn er ein Symbol ausgewählt hat, unterscheiden, ob es atomar oder zusammengesetzt ist. Atomare verschiebt er direkt, bei zusammengesetzten macht er eine Schleife über alle seine

Teile. Wenn ein Teil wiederum zusammengesetzt ist, prüft der Editor das auch und gibt die Verschiebung an die Unter-Teile weiter. So kann man das lösen. Das Design-Pattern Composite sagt aber: Die Prüfung, ob etwas zusammengesetzt ist oder nicht, gehört nicht in jedes Anwendungssystem, sondern allein in die zusammengesetzten Teile (also das Haus, in diesem Beispiel). Wir sollen verlangen, dass die Methoden, z.B. das Verschieben, von allen atomaren und zusammengesetzten Objekten verstanden wird, also in deren Klassen definiert sind. Um das zu erreichen, wird eine Oberklasse eingeführt, die die Methode verschieden definiert, aber nicht implementiert. Alle Unterklassen (atomar oder zusammengesetzt) müssen die Methode daher so implementieren, wie es ihnen entspricht. Atomare verschieben sich ganz einfach, nicht-atomare geben den Verschiebungsbefehl an alle ihre Teile weiter; und wenn die wieder zusammengesetzt sind, geht es automatisch rekursiv in die Tiefe. Vorteil: Nur die Auslöser für die Fallunterscheidung, nämlich die zusammengesetzten Teile bzw. deren Klassen, müssen auf diesen Sonderfall reagieren und die Schleife enthalten. Alle atomaren und vor allem die Anwendungsklassen brauchen davon nichts zu wissen (Information Hiding). Man kann Anwendungen programmieren ohne zu erfahren, dass per Composite-Pattern verschiedene Teile zusammengesetzt sind. Wie ein Adapter eine fehlende Methode hinzugefügt hat, so übersetzt ein Composite die Methode bei Bedarf in eine Schleife. Die UML-Struktur zeigt, wie alles dazu aufgebaut sein soll.

Observer: Eine Klasse K wird von vielen anderen beobachtet, weil sich in K Wichtiges tut: Dort können die Börsenkurse oder die Fahrtgeschwindigkeit eines Autos konzentriert sein. Die Frage lautet hier: Wie erfahren alle, die es interessiert, welche Werte ein Objekt von K gerade hat? Muss K beim Beschleunigen dem Tacho die neue Geschwindigkeit melden? Und dem Navigationsgerät auch? Das könnte man machen. Aber das Design Pattern Observer sagt: Dann müsste ja die zentrale Klasse K genau wissen, welche Werte und in welcher Form alle möglichen anderen Klassen benötigen. Vielleicht ist das Navigationsgerät aber von einem anderen Hersteller, dann wird das schon schwierig. Besser wäre es, das zu flexibilisieren. Die Klasse K soll unabhängig davon bleiben, wer sie beobachtet. Sollen doch die Beobachter genau nachsehen, wie sie ihre Informationen bekommen. Also: Soll jeder selber in K nachsehen. So ist das Pattern auch aufgebaut. Jeder Interessent kann sich bei Objekten von K registrieren lassen. Dort weiß man dann nur, welche Objekte beobachten. Ändert sich etwas am K-Objekt, so wird es allen seinen Beobachtern nur kurz mitteilen: „Ich habe mich geändert!“. Aber was sich genau geändert hat und welchen Wert das jetzt hat, sagt das K-Objekt nicht. Das müssen jetzt im nächsten Schritt alle Observer-Objekte genau vom K-Objekt abfragen. Der Tacho fragt nach der Geschwindigkeit und zeigt sie an. Die zentrale Klasse K verwaltet nur die Observer und sagt ihnen kurz und pauschal Bescheid, wenn sie sich geändert hat. Alles andere machen die Observer. Sie können kreativ sein. Da es hier vor allem um einen Ablauf zwischen Anwendungssystem, Objekt der Klasse K und Observern geht, hat man ein Sequenzdiagramm gewählt. Dieses Diagramm zeigt ein Beispiel im Detail, als UML. Will man das Design Pattern einsetzen, muss man das Beispiel natürlich auf die eigenen Klassen und Bezeichner anwenden.

Wie die Lösungsmuster genau aussehen, ist auf den Folien ausgeführt. Oben finden sie eine Beschreibung, die auf den Kern des jeweiligen Patterns eingeht. Außerdem werden immer alternative Lösungsmöglichkeiten angedeutet, die aber „nicht so gut sind“. Das ist Erfahrungssache. Gute Entwürfe zu machen, ist generell eine Frage von Vorkenntnissen und Erfahrung. Sie könnten zwar selbst eigene Design-Patterns entwickeln, aber bis dahin werden Sie vermutlich noch einige Jahre Erfahrung sammeln. Dabei sehen Sie verschiedene Lösungsmöglichkeiten und finden schrittweise heraus, welche Lösungsmöglichkeit die beste ist. Die meisten Entwickler erfinden aber nie ein eigenes Pattern. Das Wichtigste ist, dass Sie etablierte Pattern kennen und einsetzen, wenn sie passen.

MVC: Model-View-Controller-Pattern

Die drei oben skizzierten Patterns lösen jeweils sehr kleine Probleme, die aber dafür häufig vorkommen. Jedes Pattern wirkt sich darauf aus, welcher Teil des Programms mit der Problemsituation umgehen muss: Beim Adapter ist es die Klasse, die nicht ganz passt (weil sie eine Methode nicht bietet). Beim Composite ist es das zusammengesetzte Objekt (weil es einen Methodenaufruf an ihre Unterteile weitergeben muss). Dagegen ist es in beiden Fällen eben nicht das Anwendungssystem, das diese Objekte verwendet. Wieso sollten große Anwendungssysteme sich damit herumschlagen müssen, wie kleine verwendete Objekte aussehen? Der Aktualisierungsaufwand wäre dann größer. Das kann man nicht theoretisch ableiten, es ist eine Erkenntnis aus Erfahrung.

Bei MVC hat man es mit einer ständig auftretenden Problemklasse zu tun. Die Frage hier heißt: Wie soll man es organisieren, dass bei großen Systemen oft mehrere Oberflächen zu einem inhaltlichen System existieren? Diese Oberflächen sollen Zustände anzeigen und sie interaktiv verändern können. Wie werden die Zustände angezeigt, wie werden Interaktionen mit den Benutzern behandelt und welche Rolle spielen die dahinterstehenden Rechen- und Fachmodule?

Dafür gibt es sehr viele Möglichkeiten. Die Struktur und Zuständigkeit von inhaltlichen und von Anzeigeobjekten kann unterschiedlich festgelegt werden. Nach dem Design Pattern MVC gibt es dafür aber eine weitverbreitete Empfehlung, die auf den Folien ausführlich erklärt wird. Hier betone ich die wichtigen Teile und die Begründungen, die in der Vorlesung mündlich zu den Folien gegeben werden.

Die Grundidee ist, diese Aufgaben in drei Teile zu zerlegen und an drei verschiedene Objekte zu delegieren: Das ***Model*** ist alles, was inhaltlich ist und entsprechende Zustände hat. Das kann ein Konto sein, oder ein Schiff oder was immer. Was man davon auf dem Bildschirm sieht, sind die ***Views***. Von einem Konto sieht man am Bankomaten die Nummer und den Kontostand. Zu Hause im Online-Banking sieht man in einer anderen View auch noch die Buchungshistorie und die Möglichkeit für Überweisungen. Die Bankmitarbeiter sehen von demselben (inhaltlichen) Konto in ihrem Fenster vielleicht noch, welcher Überziehungskredit gewährt wurde. Das ist eine dritte View. Der letzte Teil beschäftigt sich damit, was man tun kann: man kann Geld abheben oder eine Überweisung tätigen. Dazu muss man von außen dem Programm etwas mitteilen. Das übernehmen die ***Controller***. In der Regel gehören immer eine View und ein Controller zusammen. Es gibt meist mehrere, oft ineinander verschachtelte, View-Controller-Paare. Beispiel: Auf dem Bankomaten gibt es Knöpfe, auf denen Sie wählen können, ob Sie 50/100/200 EURO abheben wollen. Die View des Kontos zeigt Ihnen diese Möglichkeiten an. Wer weniger auf dem Konto hat, sieht vielleicht den 200 EURO-Button nicht. Wenn Sie jetzt auf einen Knopf drücken (das ist eine Auswahl auf Touch-Screen als Benutzergeste), nimmt das der Button-Controller wahr und übersetzt es in: „Kunde möchte 100 EURO abheben“. Das schickt der Controller dem Model. Das Model ist die inhaltliche Verwaltung des Kontos; grob gesprochen ein Objekt der Klasse Konto. Sie weiß, dass dieses spezielle Konto 312 EUR enthält und reagiert auf diese inhaltliche Nachricht, indem sie die Abhebung zulässt. Sie meldet allen Views: „Mein interner Zustand ist geändert“. Das erfahren alle Views, auch die Bankomaten-View. Sie fragt nun wieder beim Konto nach und erfährt, dass nach der Abhebung nur noch 212 EURO auf dem Konto sind. Das zeigt sie an. Alle anderen Views machen es parallel genauso: Sie erfragen das, was sie darstellen wollen. Sie müssen wissen, wie sie das vom Model Konto erfahren – nicht das Konto muss von sich aus jeder View die passenden Informationen schicken. Da das Konto länger stabil bleibt, die Views sich aber öfter einmal ändern, ist diese Aufgabenverteilung sinnvoll. Das ist wiederum eine Einsicht aus Erfahrung.

Wie setzt man Design Patterns in der praktischen Entwicklung ein?

Das Prinzip ist einfach: Wenn Sie beim Entwurf an ein Problem bzw. eine Situation gelangen, die dem Problemteil eines Design Patterns entspricht, dann überlegen Sie, ob das Pattern in Ihrem Fall Vorteile bringt. Dann – und nur dann – bauen Sie es ein. Das passiert in der Praxis ständig. Design Pattern sind kein exotisches Konzept, sondern unter professionellen Entwicklern tägliches Brot.

Grundsätzlich verbessern die Gamma-Patterns sehr häufig die Flexibilität bei Änderungen. Wenn sich an einem Teil des Programms etwas ändert, kommt man damit leichter zurecht, wenn man sich an die Patterns gehalten hat. Das ist bei Adapter, Composite, Observer und MVC so, und noch bei vielen anderen Design Patterns. Wenn es aber sicher keine Änderungen mehr geben wird, dann ist eine andere Lösung eventuell genauso gut oder besser – weil sie einfacher ist. Daher muss man schon jedes Mal überlegen, ob sich der Pattern-Einsatz lohnt.

Wenn Sie entschieden haben, dass ein Pattern eingesetzt werden soll, muss jetzt das Muster in Ihren konkreten Entwurf eingebettet werden. Die Lösungsmuster (meist als kleine UML-Ausschnitte gezeichnet) müssen also auf konkrete Klassen und Objekte Ihrer Software abgebildet werden. Die Entsprechung der Teile im Design Pattern (z.B. Leaf und Composite im Composite-Pattern) müssen Sie dort suchen. In der Abbildung unten werden Cars und Suitcases (Leafs) in Boxen geladen (Composite). Diese Boxen können von Anwendungsprogrammen, wie hier dem Client Container, verwendet werden.

Ein anderes Beispiel ist weithin bekannt: In einem grafischen Editor gebe es die atomaren Symbole wie Kreis und Rechteck. Sie spielen die Rolle von Leafs: man kann sie direkt mit den entsprechenden Methoden verschieben, einfärben und löschen, denn für diese Klassen seien diese Methoden implementiert. Dagegen müssen Sie eine Composite-Klasse für „zusammengesetzte Symbole“ erst neu programmieren und in ihr dann auch die Schleife implementieren, die die Nachrichten an ihre Bestandteile weitergibt. Das ist eine Konsequenz aus der Anwendung des Composite-Patterns. Zusätzlich brauchen Sie eine abstrakte Oberklasse von Leaf und Composite und geben der einen eigenen Namen. Die Rolle im Pattern heißt „Component“. Sie können die entsprechende Klasse vielleicht *Zeichenobjekt* nennen. Wenn Sie alle Rollen des Design Patterns in Ihrer Software gefunden oder neu geschrieben haben, sind Sie (fast) fertig. Dann steckt in Ihrem Entwurf jetzt auch das Design Pattern Composite.

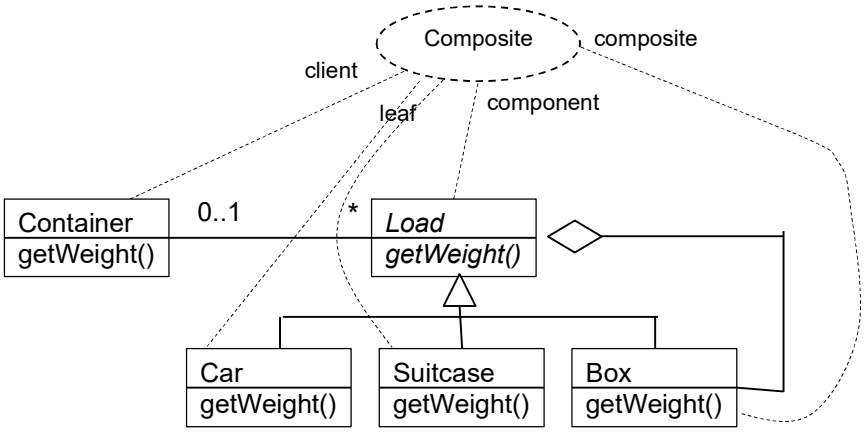
Dokumentation von eingesetzten Design Patterns

Ein Pattern zu verwenden ist gut. Fertig ist man aber erst, wenn man diese Tatsache auch dokumentiert hat, damit Kollegen und man selbst nach einiger Zeit noch weiß, wieso hier eine abstrakte Klasse steckt und eine Schleife usw. Es besteht also die Gefahr, dass die Besonderheiten des Patterns nicht erkannt und sie daher „wegoptimiert“ werden. Dann ist das Pattern kaputt und bietet nicht mehr die gewünschte Flexibilität.

Daher muss man den Einsatz von Patterns dokumentieren. Dazu gibt es in UML das gestrichtelte Oval, in das man den Pattern-Namen schreibt, z.B. Composite. Von diesem Oval gehen gestrichelte Linien zu all den Teilen des eigenen Programms, die den Rollen im allgemeinen Design Pattern entsprechen. An diesen Linien stehen die Rollennamen. Beispielsweise würde eine mit „Leaf“ beschriftete Linie im grafischen Editor zur Klasse „Kreis“ gehen, eine andere zur Klasse „Rechteck“. Eine Linie mit der Rollenbezeichnung „Component“ würde auf die Klasse Zeichenobjekt verweisen.

Im Beispiel unten ist das Composite-Pattern in der Schifsanwendung eingesetzt worden. Man sieht wieder das gestrichelte Oval für das Pattern und die Rollennamen an den gestrichelten Linien. Sie verweisen auf Teile aus dem Schiffs-Software-Entwurf. Container sind clients, Cars und Suitcases sind atomare Objekte (Leafs). Die Struktur zwischen diesen Klassen muss genau dem Composite-Pattern

entsprechen, es muss also eine abstrakte Oberklasse (hier: Load) geben und eine Assoziation vom Composite (hier: Box) zurück zu der abstrakten Oberklasse. In der entsprechenden Methode von Load, Leafs und Box muss die Schleife über alle Teile in Box stehen, die die Nachricht weitergibt. Da man auf Ebene eines Klassendiagramms (wie hier) ein Algorithmendetail wie eine Schleife nicht sieht, müsste diese entweder als Kommentar dort stehen – oder man lässt sie weg, weil ja klar ist, dass im Composite Pattern dort eine Schleife zu stehen hat.



Wenn nun jemand Ihren Entwurf sieht und diese Person UML beherrscht, ist alles klar:

1. Hier wurde ein Design Pattern verwendet, und zwar Composite
2. Die Rollen sind so belegt, wie die gestrichelten Linien andeuten
3. Damit sollte Flexibilität hergestellt werden; bitte das Pattern nicht zerstören, solange diese Anforderung noch gilt.
4. Details des Composite-Patterns muss man nun nicht mehr erklären, denn die müssen professionelle Entwickler sowieso kennen. Jetzt sparen Sie sich also die Erklärung, dass und wieso hier eine Schleife in Box.getWeight() stecken muss.
5. Wenn Sie neue atomare, zusammengesetzte Objekte oder Methoden einführen wollen, folgen Sie dem Pattern. Dann kann man z.B. auch Bikes als atomare Objekte bearbeiten oder deren Gewicht abfragen. Wenn ein Pattern einmal eingebaut ist, kann und soll man es auch bei Änderungen weiter nutzen.

Man darf voraussetzen, dass die Gamma-Patterns bekannt sind. Wer sie doch nicht genau kennt, kann nachsehen. Daher erspart man sich beim Einsatz von Patterns viel Dokumentations- und –begründungsaufwand. Inzwischen gibt es einige Bücher über Patterns, die aber alle nicht annähernd die Verbreitung des Gamma-Buchs gefunden haben. Daher muss man nicht alle diese Patterns auswendig kennen.

Welche Rolle spielt Information Hiding bei Patterns?

Information Hiding besagt in aller Kürze, dass jeweils eine Entwurfsentscheidung „versteckt“ wird. Man muss also beim Verwenden nicht wissen, was da entschieden wurde. Vorteil: Wenn sich die Entscheidung ändert, müssen Verwender bei sich nichts verändern; sie hatten sich ja sowieso nicht auf die ursprüngliche Entscheidung verlassen. Man versteckt also nicht böswillig etwas, sondern erspart den Verwendern den Aufwand, sich damit auseinandersetzen zu müssen.

Design Patterns nutzen oft Schnittstellen, abstrakte Klassen und ganz allgemein Information Hiding, um zwei Programmteile voneinander zu entkoppeln, damit sie unabhängig voneinander weiterentwickelt werden können. Wenn das Eine geändert werden muss, hat es keine Auswirkungen auf das Andere.

Einige Beispiele:

- Abstrakte Klassen wie die Component beim Composite-Pattern schreiben dem Aufrufer vor, welche Methoden zur Verfügung stehen. Umgekehrt muss jede Klasse, die Component implementiert, diese Methoden auch implementieren. Beide Seiten wissen nichts voneinander; der Verwender kann mit der Methode tun, was er will. Auch die Component-Implementierung kann jede Datenstruktur und jeden Algorithmus verwenden, den sie will – solange nur die Schnittstelle für die Methode erhalten bleibt.
- In Java gibt es das Schlüsselwort *Interface*. Es ist etwas spezieller als eine abstrakte Klasse. So müssen *alle* Methoden eines Interface abstrakt sein, dürfen also im Interface noch nicht implementiert sein. Eine abstrakte Klasse muss dagegen mindestens eine abstrakte Methode haben; andere können implementiert sein. Damit ist ein Interface ein Sonderfall einer abstrakten Klasse, bei der alle Methoden abstrakt sind. Aber häufig werden abstrakte Klassen in Design Patterns zur Interfacebildung eingesetzt. Man will sich nämlich nicht auf Java beschränken; Design Patterns sind auf alle objektorientierten Sprachen anwendbar.
- Wie oben schon ausgeführt, sollen Sonderfälle in einem Programm immer dort behandelt werden, wo sie auftreten. Die Schleife bei Composite also innerhalb des zusammengesetzten Objekts – nicht in allen Anwendungsprogrammen. Die sollen davon gar nichts mitbekommen, dass es hier einen Sonderfall gibt.

Mit dem Mittel von objektorientierten Strukturen und Schnittstellen erreichen Design Patterns Strukturen, die flexibler sind als andere, ähnliche. Das ist ihre Stärke.

Vertiefung: Von Use Cases zu User Stories und Boards

Die letzte Vertiefung zeigt, wie man traditionelle und agile Vorgehensweisen miteinander verbinden kann. Das tut man, um die Vorteile beider Vorgehensweisen zu kombinieren. Natürlich ist das nicht ganz einfach, denn man bekommt die Vorteile nicht, ohne auch einige Nachteile in Kauf zu nehmen. Das konkrete Beispiel hier ist es wert, genauer betrachtet zu werden. Denn einerseits ist es genau das hybride Vorgehensmodell, dem Sie im Software-Projekt SWP selbst folgen werden. Und zum anderen haben wir aus den letzten Jahren schon sehr viel Erfahrung mit diesem Modell gesammelt und können es aus Erfahrung bewerten.

Grundidee: Klare Übersicht und dennoch Flexibilität

In sehr vielen Projekten wünscht man sich zu Anfang eine verbindliche, für alle Projektteilnehmer verständliche Arbeitsgrundlage. Typischerweise ist dies eine Spezifikation. In traditionellen Entwicklungsprozessen werden die Spezifikationen dann unterschrieben und freigegeben. Von diesem Punkt an soll sie sich nicht mehr ändern. Sie gilt als Vertragsgrundlage und dient am Projektende dazu festzustellen, ob alle Anforderungen erfüllt wurden.

Problematisch ist dieser Ansatz nur, wenn anfangs die Anforderungen gar nicht alle bekannt oder noch unklar sind; wenn die Kunden während des Projekts ihre Meinung ändern oder präzisieren; oder wenn neue Ideen aufkommen, so dass man doch andere Prioritäten setzen möchte. Mit einem Wort: Wenn sich etwas ändern soll, ist man mit einer festgeschriebenen Spezifikation nicht mehr gut beraten. Es besteht die Gefahr, die veraltete Spezifikation trotzdem umzusetzen, um vertragstreu zu sein.

Das widerspricht jeder agilen Überzeugung. Dort würde man unbedingt versuchen, die Änderungen noch einzubringen, in der Hoffnung, dass der Kunde dann am Ende viel zufriedener ist, weil der Endstand auf Basis aller neuen Erkenntnisse und Wünsche zustande gekommen ist. Also würde man dem normalen iterativen vorgehen eines agilen Projekts folgen. Anforderungen stehen auf Story Cards

und werden im Planning Game oder in der Sprint-Vorbereitung priorisiert. Im nächsten Sprint können sie verändert, umpriorisiert oder ergänzt werden.

Im SWP und in vielen Unternehmen möchte man beides: Eine gute Übersicht am Anfang, an der sich Kunden und Entwickler orientieren können. Und die Möglichkeit, schnell auf Änderungen zu reagieren und sie zu berücksichtigen, ohne aufwändig eine Spezifikation aktualisieren zu müssen. Man möchte eigentlich am liebsten Karten hin- und herschieben und damit die Arbeiten organisieren.

Hybrider Prozess, auch im SWP

Nach langen Überlegungen und Diskussionen wurde eine Kombination entwickelt, die folglich ein hybrider Entwicklungsprozess ist:

- In der ersten der vier Iterationen wird eine Spezifikation geschrieben. Kunden und Entwickler versuchen, ein gemeinsames Verständnis zu entwickeln und dokumentieren es in diesem Dokument. Das wird dann von beiden Seiten unterschrieben und freigegeben.
- Nun möchte man flexibler werden und zerlegt die Spezifikation in Karten: Sie entstehen im Wesentlichen aus den Use Cases der Spezifikation. Jeder Use Case wird in ca. 1-5 User Stories (Karten) zerlegt. Diese kommen auf ein Agile Task Board und werden von da an immer mit den Kunden besprochen und aktualisiert.
- Ab diesem Zeitpunkt wird die Spezifikation nicht mehr aktualisiert. Änderungen auf Basis der User Stories werden nur an den Abnahmetestfällen nachvollzogen. Die Spezifikation hat ihren Zweck erfüllt: Sie hat nach der ersten Phase einen guten Überblick geboten und steht dafür auch weiterhin zur Verfügung; Ihre Bestandteile sind in den Anfangszustand der Story Cards übergegangen.
- Bei jedem Treffen mit dem Kunden ist auch das Task Board mit den Story Cards dabei. Mit dem Board planen die Entwickler, was wann zu tun ist und von wem. Mit dem Kunden besprechen sie regelmäßig, ob sich etwas geändert hat oder ob sich bei ihm oder ihr die Prioritäten der verschiedenen Karten sich verändert haben. Ab der zweiten Iteration ist die Spezifikation in den Hintergrund getreten, es werden nur noch die Karten auf dem Board verwendet.
- Beim Abnahmetest prüft man das Produkt gegen die veränderten Abnahmetests. Sie stammen ursprünglich aus der Spezifikation, wurden aber als eigenes Dokument aktualisiert, wenn eine Änderung auch Auswirkungen auf die Abnahmetests hatte. Sie haben sich daher von der Spezifikation getrennt und wurden einzeln aktualisiert.

Was erreicht man mit dem Verfahren?

Vordergründig erscheint es wie Zeitverschwendung: Wieso sollte man eine ganze Spezifikation schreiben, wenn man sie gar nicht weiterverwenden will? Dieser Einwurf übersieht aber, dass die Spezifikation durchaus weiterverwendet wird, wenn auch in ungewöhnlicher Form: Sie dient immer noch im Hintergrund als Übersicht. Diese Aufgabe ist sehr wichtig und sie ist nicht abgeschlossen, sobald die Spezifikation unterschrieben ist. Zusätzlich wird die Spezifikation aber weiterentwickelt, indem sie in User Stories übersetzt wird. Das ist ein weiterer Arbeitsschritt, aber er lohnt sich. Denn viele Kunden können nicht von Anfang an in kleinen Teilen denken. Sie brauchen erst einmal das Gesamtbild und können sich auf die Karten erst einlassen, wenn sie wissen, wie sie in dieses Gesamtbild passen.

Ein Problem hat es im SWP tatsächlich schon vereinzelt gegeben, daher muss man hier wirklich aufpassen: Manche Kunden dachten, sie würden am Ende auch genau das bekommen, was in der Spezifikation stand. Aber das ist zum Glück nicht so! Sie werden das bekommen, was sie sich zum Ende des Projekts wünschen, nicht am Anfang. Damit das funktioniert, müssen aber wirklich beide Seiten

(Entwickler und Kunden) ganz klar übereinstimmen, dass die Spezifikation nur ein Zwischenergebnis ist. Was schließlich herauskommen soll, entscheiden die Karten auf den Boards, und die Abnahmetestfälle.

Es gibt noch einen dritten Grund, der für diese Vorgehensweise spricht: Sie gibt Ihnen Gelegenheit, eine echte Spezifikation unter realen Bedingungen zu schreiben und mit dem Kunden zu diskutieren. Sogar die Spezifikation aus der Teamaufgabe ist kurz und künstlich im Vergleich zu dem Dokument aus dem SWP. Es gibt also zwei gute Gründe, mit Spezifikation zu beginnen, sie dann aber bald aufzulösen und iterativ/agiler weiterzuarbeiten. (1) Sie lernen zu spezifizieren, und (2) die Projekte im SWP erhalten eine maßgeschneiderte Kombination aus Übersicht und Flexibilität. Jeder dieser Gründe würde schon für sich ausreichen, die hybride Vorgehensweise durchzuführen. Es gibt auch Nachteile, aber die sind im Vergleich damit gering und werden in Kauf genommen.

Wie zerlegt man die Spezifikation in User Stories?

Dabei kann man folgenden Daumenregeln folgen:

- Beginnen Sie mit den Use Cases
    a. Machen Sie aus jedem Use Case zuerst ein „Epic“, das ist eine Überschrift. Aus dem Hauptszenario machen Sie nun (mindestens) eine Story Card.
    b. Wenn das Hauptszenario zu lang oder zu kompliziert ist, teilen Sie es in 2-3 Teile auf. Jeder Teil sollte für sich schon eine sinnvolle und nützliche Aktion sein und ist eine eigene Story Card.
    c. Machen Sie jetzt aus Extensions und möglicherweise anderen Teilen des UC weitere Story Cards. Sie können jede Extension in eine eigene Story Card übernehmen oder mehrere zusammen auf eine, wenn sie sehr einfach waren.

- Lassen Sie den Kunden nun mit A/B/C/D bewerten, ob jede Karte eine besonders wichtige, eine mittel-wichtige oder eine Funktion darstellt, die schön zu haben wäre (nice to have), oder sogar eher verzichtbar (D). Dann ordnen Sie die Karten auf einem Board an:
    a. Schreiben Sie den Namen des Use Case als Spaltentitel („Epic“) auf
    b. Hängen Sie darunter die mit A bewerteten Karten, die aus diesem Use Case hervorgegangen sind.
    c. Dann halten Sie etwas Abstand und hängen drunter die B-Karten dieses Use Cases.
    d. Die C-Karten kommen ganz zuunterst.
    e. Das tun Sie mit allen User Stories: Nach rechts als Spalten die User Stories, darunter jeweils die daraus hervorgegangenen User Stories in A/B/C-Priorisierung.
    f. Nun betrachten Sie alle A-Karten aus allen Use Cases als den Inhalt der ersten Programmieriteration (zuvor hatten Sie nur Anforderungen erhoben und die Spezifikation geschrieben). Die B-Karten gehören zur Iteration-2, und die C-Karten hebt man sich zwar auf, ordnet sie aber noch keiner Iteration zu.

- Jetzt kommt noch einmal der Plausibilitäts-Check:
    a. Wichtig: Betrachten Sie das Ergebnis zusammen mit dem Kunden. Die Entwickler sollen abschätzen, ob sie alle A-Karten in einer Iteration umsetzen können. Falls nicht, bitten Sie den Kunden, die unwichtigsten A-Karten nach unten zu hängen. Haben Sie mehr Zeit, können B- oder C-Karten von unten nach oben befördert werden.
    b. Wie viel sie schaffen, schätzen die Entwickler. Der Kunde kann zwar mit ihnen diskutieren, aber nicht erzwingen, dass sie sich mehr Karten in eine Iteration laden, als sie schaffen können.

c. Das kann im Endeffekt dazu führen, dass der Kunde nicht alles bekommen wird, was er sich wünscht. Die ständigen Gespräche und Diskussionen dienen auch dazu, den Kunden zu verdeutlichen, dass Entwickler eine Grenze haben. Andererseits darf der Kunde immer wieder neu priorisieren, welche Funktionen ihm oder ihr besonders wichtig sind. So profitieren beide Seiten. Das funktioniert aber nur, wenn wirklich bei jedem Treffen das Board verwendet und bei Änderungen aktualisiert wird. Das Board ist die Kontrolltafel eines agilen Projekts.

- Wann ist eine Karte fertig?
    a. Wenn dem Kunden die Funktion vorgeführt wurde und er bestätigt, dass sie seinen Vorstellungen entspricht. Das ist ein informeller Vorgang ohne Unterschrift.
    b. Ganz am Ende führen wir die Abnahmetests durch. Wenn sie korrekt aktualisiert wurden, sollte es keine Probleme geben. Dann sind alle vorher schon abgenommenen Karten korrekt erfüllt.
    c. Fertige Karten kann man wegwerfen. Oft hebt man sie doch auf, aber die Methode verlangt das nicht. Niemand wird mehr nach den alten Karten fragen.

Tipps für die Transformation in User Stories

- Was tut man, wenn der Kunde mehr will?
    a. Verdeutlichen Sie ihm, dass innerhalb der zeitlichen Möglichkeiten eine neue Funktion nur dann hereinkommen kann, wenn eine ähnlich aufwändige herausfällt, also niedriger priorisiert wird. Das kann man leichter verständlich machen als am Ende zu sagen: Für diese anderen Funktionen hatten wir dann keine Zeit mehr.

- Weitere Karten, über die Use Cases hinaus
    a. Die Spezifikation besteht ja nicht nur aus Use Cases. Es kann sein, dass auch in anderen Teilen noch funktionale Anforderungen stehen, die sie in die iterative Umsetzung retten wollen. Das ist möglich, lässt sich aber nicht so gut als geregelten Ablauf beschreiben. Wenn Sie also meinen, noch eine Aufgabe hinzufügen zu wollen: Tun Sie es! Das könnte beispielsweise eine sehr komplizierte Oberfläche sein. Die stünde ja nicht in einem Use Case, sondern vielleicht als eine Menge von Mockups in der Spezifikation.
    b. Manche Spezifikationsteile sind für den Überblick sehr wichtig, sie liefern aber nicht direkt Story Cards; so zum Beispiel die nicht-funktionalen Anforderungen oder die Mission zu Beginn der Spezifikation. Die Spezifikation bleibt ja erhalten, man kann diese Teile immer noch einmal durchsehen.
    c. Dem Kunden fallen noch mehr Funktionen oder Sonderfälle ein, wenn er/sie sieht, wie die Use Cases zerlegt werden. Nehmen Sie auch diese zusätzlichen Story Cards auf, balancieren Sie dann aber noch die Aufwände (was in einer Iteration zu schaffen ist).