Softwareentwicklung

Mario Konrad
Mario.Konrad@gmx.net

8. Oktober 2003

Inhaltsverzeichnis

Abbildungsverzeichnis

Literatur

[1] Eric S. Raymond. *The Cathedral and the Bazaar*. O'Reilly, 2nd edition, Januar 2001.

[2] Frederick P. Brooks Jr. *The Mythical Man-Month*. Addison-Wesley, May 2001.

[3] The Standish Group. CHAOS Report, 2001.

[4] Tom DeMarco and Timothy Lister. *Peopleware - Productive Projects and Teams*. Dorset House Publishing, 2nd edition, 1999.

Glossar

ASCII *American Standard for Character Information Interchange.* Dieses Datenformat gilt als der am weitesten verbreiteste, meistgenutzte und flexibelste Format. Dateiformate wie HTML- oder auch XML-Dateien beruhen auf diesem Format.

Closed Source Der Quellcode der Software ist nur dem Hersteller bekannt. Das Gegenteil von *Open Source*.

CPU *central processing unit*, der Hauptprozessor des Computers der die hauptsächlichen Arbeitsschritte durchführt.

GUI Das GUI, *graphical user interface*, ist eine grafische Benutzeroberfläche, in der mit der Tastatur und Maus der Computer bedient werden kann.

Hacker Enthusiastischer Programmierer. Jemand der starkes Interesse an der Funktionsweise von technischen Mitteln, wie z.B. dem Computer, zeigt. **Nicht** jemand der, wie von den Medien falsch interpretiert, in Computersysteme einbricht, Daten stiehlt, löscht, verändert, verkauft oder einen Kopierschutz knackt. Die wäre ein *Cracker*.

IDE *integrated development environment.* Dies ist eine (meist) grafische Oberfläche, welche dem Programmierer Arbeit abnehmen soll. Die IDE hat auch den Zweck die einzelnen Schritte die notwendig sind um Software zu erstellen vor dem Programmierer zu verstecken, als Gegenleistung aber Komfort bieten soll.

KISS Abkürzung für *"keep it simple, stupid"*, was soviel bedeutet wie die Software einfach und funktionell zu halten. Dieser Grundsatz findet in vielen Programmen (vorwiegend unter Unix oder Derivate) anwendung, wobei jedes Programm nur eine bestimme Funktion besitzt, diese aber sehr gut beherrscht. Unter Verwendung von mehrerer solcher "simpler" Programme, kann ein übergeordnetes Ziel erreicht werden (Kombination der Einzelfunktionen).

Library Softwarebibliothek. Eine Sammlung von, meist logisch zusammenhängender, Funktionen. Ein modularer Teil eines Programms, das als abgeschlossene Einheit behandelt werden kann.

Open Source Software bei der der Quellcode, oder auch Sourcecode, offen liegt. Dies bietet die Möglichkeit der Weiterentwicklung durch dritte oder zur besseren Zusammenarbeit. Das Gegenteil von *Closed Source*.

Rapid Prototyping Der Vorgang, einen Prototyp in sehr kurzer Zeit zu erstellen. Dieser Prototyp hat meist keine oder nur wenige Funktionen.

Wird meistens dazu eingesetzt einen Eindruck über die Benutzerschnittstelle zu geben.

RUP *Rational Unified Process*. Iterativer Entwicklungprozess, spezifiziert von der Firma *Rational*.

Tool Chain Kette von Werkzeugen die hintereinander verwendet werden um ein Bestimmtes Ziel zu erreichen, wobei jeweils das Resultat eines Werkzeuges als Rohstoff des nächsten Werkzeugs dient.

User Interface Schnittstelle zum Benutzer. Diese kann in verschiedenen Formen auftreten, sowohl als Textschnittstelle als auch als grafische Schnittstelle, siehe *GUI*.

Vorwort

In unzählige Diskussionen wurde ich schon verwickelt in denen es um Softwareentwicklung, Programmierung und Softwareentwicklungs-Projekte ging. Wie in vielen anderen Bereichen existieren auch hier unzählige Meinungen und Erfahrungen. Verschiedene Meinungen und Erfahrungen sind Grundsätzlich eine gute Sache, nur erlebe ich leider viel zu oft, dass Erfahrungen die gemacht wurden nicht berücksichtigt werden und Folgen von Fehlern wiederholt durchlitten werden müssen. Oft passiert es auch, dass lehrreiche Erfahrungen anderer Leute keine Beachtung finden. Gründe und Folgen werden in diesem Dokument näher erläutert.

Diese Diskussionen werden so oft geführt, auch von und mit mir, dass ich mich entschlossen habe die Gedanken schriftlich festzuhalten. Ich erhoffe mir dadurch, dass so die Diskussionen nicht verschwinden aber dass eine erweiterte Basis geschaffen wird und nicht immer wieder von vorn begonnen werden muss.

Teil I
Einleitung

Dieses Kapitel gibt eine kurze Einführung zu den Fragen: was ist Software und was ist Softwareentwicklung? Ferner wird kurz auf eine kleine Anzahl von verschiedenen Werkzeugen eingegangen, die zur Softwareentwicklung eingesetzt werden. Natürlich ist dies keine vollständige Liste, soll aber einen Überblick über die verschiedenen Werkzeuge und ihre Funktionen geben.

1 Software und Hardware

Der Begiff *Software* ist heute sehr weit verbreitet, doch was verbirgt sich dahinter? Was ist denn überhaupt diese "weiche Ware"?

Computer, z.B. PCs, funktionieren durch zwei Dinge: die Hardware und die Software. Als Hardware können all jene Teile des Computers bezeichnet werden, die man anfassen kann: den Monitor, die Tastatur, die Maus, das Gehäuse und sogar die CPU (central processing unit, der Hauptprozessor). Um dem Computer Leben einzuhauchen, dafür ist die Software verantwortlich. Die Software steuert die Hardware und stellt die Schnittstelle zum Benutzer zur Verfügung (user interface).

Ein Blick hinter die Kulissen bringt noch eine klarere Definition des Begriffs *Software*. Ausführbare Software (im Allgemeinen Software oder Programme genannt), besteht aus einer Folge von Anweisungen die der Prozessor ausführen soll, d.h. das Programm diktiert der CPU die auszuführenden Arbeitsschritte. Dies zeigt auch sehr schön, dass die Software die Hardware steuert.

Die Software kann der Computer im Allgemeinen nicht selbst erstellen, sich also nicht selbst die Arbeitsschritte diktieren. Dies ist bis heute immer noch die Arbeit des Menschen. Wie er die Software er die Software und welche Probleme ihm dabei gestellt werden, wird in den folgenden Kapiteln erörtert.

2 Die Werkzeuge und Artefakte

Wie jede andere Arbeit auch, stehen um die Software zu erstellen auch verschiedene Werkzeuge zur Verfügung, die verschiedene Arbeiten unterstützen oder z.T. sogar gänzlich übernehmen. Um nun ein Programm zu erstellen bedient man sich im Normalfall mehrerer Werkzeuge, die je einen Teil erledigen, in Serie. Dies nennt man dann die *tool chain* (Werkzeugkette).

Als Artefakte werden Dateien bezeichnet, die während der gesamten Vorgangs des Erstellens von Software involviert sind, vom Ausgangspunkt bis hin zum fertigen Produkt.

Betrachten wir zunächst mal die Abbildung 1. Sie zeigt den generellen Ablauf der Entstehung von ausführbarer Software bzw. Programmen.

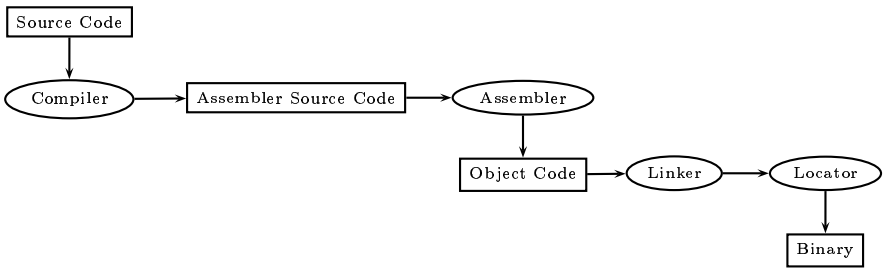

Abbildung 1: Toolchain

Die einzelnen Werkzeuge und Artefakte werden in den folgenden Kapiteln beschrieben.

2.1 Werkzeuge

Dies ist, wie schon kurz erwähnt, keine vollständige Liste. Es wird auch nicht auf Produkte von verschiedenen Herstellern eingegangen. Es soll lediglich gezeigt werden, welche Werkzeuge massgeblich beteiligt und was deren Aufgaben sind.

2.1.1 Der Compiler

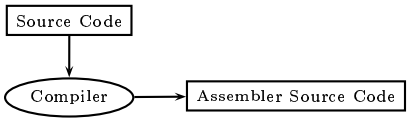

Abbildung 2: Toolchain: Compiler

Die Aufgabe des Compilers liegt darin, den Quellcode, der in einer Hochsprache geschrieben ist, in Assembler-Quellcode zu übersetzen. Die vielen verschiedenen, für Menschen lesbaren, Hochsprachen, müssen von Compilern in eine Sprache übersetzt werden die der Computer versteht. Dies ist nicht direkt der Assembler-Sourcecode, jedoch ist dieser sehr Prozessorspezifisch.

Der Compiler macht es also möglich, dass wir dem Computer Anweisungen geben können, die für Menschen bequem anzugeben und verständlich sind. Er macht es aber auch möglich, dass grössere und komplexere Programme geschrieben werden können.

2.1.2 Der Assembler

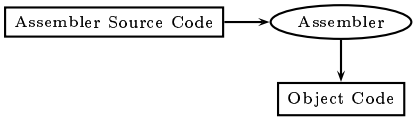

Abbildung 3: Toolchain: Assembler

Der Assembler ist das Werkzeug, das den Assembler-Sourcecode (meistens erzeugt vom Compiler) in Maschinensprache übersetzt, in einen sogenannten Objektcode. Dies ist die Sprache des Prozessors, für Menschen aber nur sehr schwer bis gar nicht lesbar.

Der Zwischenschritt von Compiler und Assembler über den Assembler-Sourcecode ist auf den ersten Blick unnötig. Tatsächlich gibt es ältere Compiler die diesen Schritt auslassen. Es hat sich jedoch gezeigt, dass es ein sinnvoller Schritt ist, da sehr viel Flexibilität erreicht werden kann. Zum Beispiel ist es möglich Teile von Programmen in verschiedenen Hochsprachen zu schreiben, diese mit entsprechenden Compilern in Assembler-Sourcecode zu übersetzen und dann mit einem Assembler den Objektcode für den Prozessor zu erzeugen. Dies kann von unschätzbarem Wert sein, obwohl in der Praxis selten eingesetzt.

Es wird so auch möglich Optimierungen oder Fehlerbehebung durchzuführen, die der Compiler allenfalls nicht in der Lage ist.

2.1.3 Der Linker

Abbildung 4: Toolchain: Linker

Der vom Assembler generierte Objektcode liegt in Bruchstücken vor. Die Aufgabe des Linkers besteht nun darin alle *benötigen* Stücke zu sammeln und zusammenzufügen (engl. *to link* = verbinden, koppeln). Stücke die nicht benötigt werden, z.B. unbenutzte Teile einer Softwarebibliothek, können so weggelassen werden, um das ausführbare Programm nicht unnötig aufzublasen.

Das Resultat des Linkers sind Gruppierungen von Bruchstücken die Gemeinsamkeiten aufweisen wie z.B. gemeinsam genutzer Speicher, etc. Dies

kann die Vorstufe zu einem ausführbaren Programm oder aber eine Softwarebibliothek (*library*) sein.

2.1.4 Der Locator

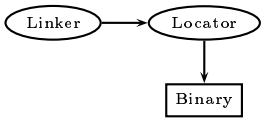

Abbildung 5: Toolchain: Locator

Als letzte Station tritt der Locator in Aktion. Er ist es der die vom Linker erzeugten Gruppen an die richtigen Stellen platziert (engl. *to locate* = fixieren, siedeln).

Somit ist ein ausführbares Programm erzeugt und bereit zur Ausführung.

2.1.5 Der Debugger

Der Begriff *debugging* rührt daher Softwarefehler, die sog. *bugs*, zu finden und zu entfernen (*de-bugging* oder auch *debugging*). Ein sehr wertvolles und mächtiges Werkzeug wenn es darum geht Fehler aufzuspüren, da der Entwickler das ausführbare Programm Schritt für Schritt ausführen kann und alle Abläufe überprüfen kann.

2.1.6 Die IDE

Als integrierte Softwareentwicklungs-Umgebung (*integrated development environment*) stellt sie dem Entwickler eine grafische Benutzeroberfläche zur Verfügung, um die einzelnen Schritte (Compiler bis Debugger) zu verbergen. Ein Editor zur Erstellung und Bearbeitung des Sourcecodes ist enthalten, sowie meistens eine Unmenge von Optionen für diverse Einstellungen.

IDEs sind ein zweischneidiges Schwert. Zum Einen bieten sie dem Entwickler mehr oder weniger Komfort, zum Andern entmündigen sie ihn jedoch. Der gebotene Komfort wird oft von Einsteigern bevorzugt, da in den meisten Fällen wenig oder keine Kenntnis über obigen Werkzeute (Compiler bis Debugger) vorhanden sind. Je nach grösse des Softwareentwicklungs-Projekts kann (nicht zwingend) die IDE zur Übersicht beitragen.

Die andere Seite hat leider versteckte aber sehr ernst zu nehmende Konsequenzen. Da die verschiedenen Abläufe vor dem Entwickler versteckt werden, hat er in den meisten Fällen, keinen oder nur wenig Einfluss auf den selben. Die fehlende Kontrolle führt oft zum Verlust der Übersicht und schnell zu einer Arbeitsweise die als *try and error*, d.h. Probieren bis es klappt, bekannt

ist. Erfahrene Softwareentwickler lehnen oft IDEs deswegen ab. Kontrolle wird von erfahrenen Entwicklern gegenüber dem, oft gar nicht so grösseren, Komfort bevorzugt.

Ironischerweise führt dies oft zu Unverständnis. Vor allem bei Einsteigern und Führungskräften, da diese Arbeitsweise als archaisch angesehen wird. Auf diesen Punkt wird weiter unten noch ausführlicher eingegangen. Ein anderer Punkt sind die entstehenden Abhängigkeiten, mehr dazu weiter unten.

2.2 Artefakte

Während der Softwareentwicklung entstehen verschiedene Artefakte oder Erzeugnisse, die jeweils einen andern Inhalt haben und Zewck erfüllen. Diese Artefakte werden zum Teil von Menschen erzeugt, zum Teil aber auch von verschiedenen Werkzeugen (siehe oben).

Es liess sich leider nicht verhindern, bereits die Begriffe *Quellcode* und *Objektcode* zu verwenden. Diese werden nun in diesem Kapitel näher erklärt.

2.2.1 Quellcode - Source code

Der Mensch bedient sich, unter anderem, der Sprache zur Kommunikation. Es gibt verschiedene Sprachen mit verschiedenen Eigenschaften. Manche haben eine komplexe Grammatik, andere haben einen sehr grossen Wortschatz und wieder andere variiren die Betonung.

Eine Programmiersprache ist im Wesentlichen nichts Anderes als eine gesprochene Sprache der Menschen. Sie weist auch einen Wortschatz sowie eine Grammatik auf, und sie dient zur Kommunikation zwischen dem Entwickler und dem Computer. Mit Hilfe einer Programmiersprache gibt der Entwickler dem Computer Anweisungen. Diese Anweisungen werden aber vom Computer nicht direkt verstanden also müssen sie zuerst übersetzt werden (siehe Compiler).

Es braucht also ein Artifakt (ein Dokument), welches diese Anweisungen enthält. Da dies die Quelle der Anweisungen ist, wird dieses Artifakt auch Quellcode oder Sourcecode genannt (engl. *source* = Quelle).

Dieser Sorucecode ist vergleichbar mit jedem andern Dokument. Der Code existiert nicht des Codes willen. Er besitzt einen Inhalt wie ein literarisches Werk, welcher gelesen und verstanden werden kann. Es ist eine Form des Ausdrucks um dem Computer die gewünschten Arbeitsschritte mitzuteilen. Wie auch in andern Werken stecken Ideen und Gedanken, die mit Hilfe einer Sprache sich in im Dokument manifestieren.

Diese etwas philosophische Erklärung hat nichts mit Begriffen *hacker*, *cracker*, etc. oder religiösem Gedankengut zu tun. Es soll lediglich das Verständnis für den Begriff *sourcecode* und dessen Bedeutung fördern.

2.2.2 Objektcode

Der Objektcode ist ein Artifakt, dass den Code enthält den der Prozessor direkt versteht, in sog. Maschinensprache. Allerdings liegt der Objektcode (in den allermeisten Fällen) als eine Sammlung von Teilen vor; ungeordnet und undplatziert. Es ist das Erzeugnis des Assemblers.

Es ist durchaus möglich, dass Objektcode geordnet, jedoch noch nicht richtig platziert ist. In diesem Zustand ist der Objektcode nach der Verarbeitung durch den Linker. Softwarebibliotheken (*libraries*) liegen in dieser Form vor.

Der Objektcode kann, obwohl er in der Sprache des Prozessors vorliegt, nocht nicht ausgeführt werden.

2.2.3 Binärcode

Dies ist der Code in dem jedes ausführbare Programm vorliegt. Es kann vom Prozessor ausgeführt werden, und beinhaltet alle nötigen Teile, die auch am richtigen Ort liegen. Im Fachjargon wird dieser Code oft einfach als *binary* bezeichnet.

Dieser Code ist für einen Menschen kaum lesbar. Nur mit eklatantem Aufwand wäre es möglich diesen Code zu lesen und zu verstehen.

2.2.4 Dokumentation

Dieses Arifakt wird zu oft unterschätzt, sowohl in seiner Bedeutung als auch in seinem Aufwand. Natürlich liegt die Dokumentation in einer Form vor, die für Menschen lesbar ist, oft in der Muttersprache der Entwickler, meistens allerdings in englischer Sprache, dem Quasi-Standard in der Computerindustrie.

Gründe und Konsequenzen für die chronische Unterschätzung der Dokumentation werden weiter unten erörtert.

3 Forschung und Entwicklung

Innerhalb des Erstellens einer Software bestehen Unklarheiten bezüglich verschiedenen Begriffen. Hier wird nun versucht diese Unklarheiten zu beseitigen.

Benutzt werden drei Begriffe: *Forschung*, *Entwicklung* und *Produktion*. Setzen wir diese drei Begriffe in ein anderes Umfeld als die Softwareindustrie, z.B. in die Abteilung Elektronik/Elektrotechnik, dann scheinen keine Unklarheiten zu bestehen:

Forschung: das Interesse gilt physikalischen Mechanismen und verschiedenen Materialien um neue Technologien zu erhalten. Es geht darum physikalishe Vorgänge zu optimieren oder sich neue Vorgänge zu nutze

zu machen. Wie lange Forschung dauert kann meistens nicht im Voraus abgeschätzt werden, da man sich auf völlig neuem Terrain bewegt.

Entwicklung: Prozessoren, Geräte und ganze Systeme werden von einer Anforderung oder auch einer Idee von Grund auf erstellt, wobei noch nicht alles was im Verlauf der Entwicklung zu Beginn schon bekannt ist. Hier werden Technologien, welche die Forschung entdeckt hat, umgesetzt. Die Dauer der Entwicklung kann Teilweise abgeschätzt werden, da die Technologie vorhanden ist und "nur" Erfahrungen mit der Umsetzbarkeit der Technologien gemacht werden muss.

Produktion: ein entwickeltes Gerät wird Produziert, dies meist in grösseren Mengen. Ist die Produktion einmal im Gang, gibt es praktisch keine (grossen) Überraschungen mehr, die Dauer der Produktion kann sehr gut abgeschätzt werden.

Versuchen wir nun diese Definitionen, diese verschiedene Arbeitsschritte auf die Softwareindustrie zu übertragen. In dieser Industrie gibt es viele Analogien, die jedoch nicht immer auf den ersten Blick zu erkennen sind.

Forschung: hier werden Algorithmen erarbeitet und optimiert, es werden gurndsätzliche Gedanken über das Funktionieren und den Ablauf von Software gemacht, usw. Dies beinhaltet die theoretische wie auch die praktische Informatik. Auch hier, wie in andern Industrien, werden Grundsteine für Technologien gelegt.

Entwicklung: die Softwareentwicklung befasst sich mit dem Thema wie Technologien in Produkte umgesetzt werden können. Dabei treten möglichweise Probeleme auf, gegeben durch Randbedingungen wie z.B. Hardware, die gelöst werden müssen, meistens durch einen evolutionären Prozess. So werden Erfahrungen mit Technologien gemacht. Das Resultat einer Softwareentwicklung ist im Allgemeinen eine Studie oder ein Produkt. Die Verwendung des Begriffs "meistens" deutet darauf hin, dass viele Entwicklungen nicht so enden wie geplant waren, mehr dazu weiter unten.

Produktion: existiert viel Erfahrung auf dem Gebiet einer Technologie, so dass keine Neuerungen und keine unbekannten, vielleicht noch nicht gelöste, Probleme mehr auftauchen können, nennt man dies Produktion. Dies sind Softwaresysteme die schon mehrfach existieren, nun wird eine weitere Version hergestellt, oft massgeschneidert auf eine bestimmte Zielgruppe. Von einem evolutionärem Prozess ist hier nicht mehr viel übrig und es gibt eine Vielzahl von Werkzeugen die Teile der Produktion dem Menschen abnehmen. **Achtung:** dies hat **Nichts** mit Softwareentwicklung zu tun!

Oft werden Entwicklung und Produktion zusammengefasst oder gleichgesetzt, dies ist ein Fehler. Der Fokus dieser unterschiedlichen Prozesse ist ganz anders gerichtet und die Arbeitsweise hat auch nicht viel Gemeinsamkeiten. Trotzdem wird häufig nur von *Softwareentwicklung* gesprochen, dies mit oft fatalen Folgen (siehe Risiken).

Teil II
Anforderungen

Dieses Kapitel beschäftigt sich mit der Frage der Anforderungen an eine Software. Warum diese so wichtig sind wird ebenso betrachtet wie deren Konsequenzen, vor allem die der *late requirements* (späten Anforderungen, spät im Sinne von weit forgeschrittenem Projekt).

4 Wunschkonzert

Jeder der schon selbst Hand angelegt hat und ein Stück Software geschrieben hat kennt wahrscheinlich das Problem: die Software muss alle möglichen und unmöglichen Funktionen, zusätzlich eine schöne und einfache Oberfläche bieten, muss offen sein um weitere *Features* aufzunehmen und muss mit sämtlichen bekannten Applikationen Mühelos zusammenspielen.

Sich etwas zu wünschen ist sicher sehr schön, das kennen selbst die Kinder wenn es auf die Weihnachtszeit zugeht. Das Problem bei den Wünschen sind die resultierenden Konsequenzen. Jeder ausgesprochene Wunsch, also eine Anforderung, bedeutet erstens einen gewissen Aufwand und zweitens eine erhöhte Komplexität des Systems. Im Folgenden wird auf diese zwei Konsequenzen eingegangen.

4.1 Aufwand

Jede zusätzliche Anfoderung bringt zusätzlichen Aufwand. Angenommen eine Software soll 10 Anforderungen genügen ($n = 10$) und hat einen Gesamtaufwand von A. Das würde bedeuten, dass jede einzelne Anforderung im Durchschnitt einen Aufwand von $\frac{A}{n}$ haben müsste. Dies ist jedoch leider **falsch**!

Nicht jede Anforderung hat den gleichen Aufwand um Umgesetzt werden zu können. Das Problem ist die Annahme der *Normalverteilung* des Gesamtaufwands auf die einzelnen Anforderungen.

Eine Software besteht aus einer Anzahl n von Funktionen die sich jedoch eine bestimmte Basis teilen. Man kann dies als Infrastruktur innerhalb der Software bezeichnen. Nennen wir den Aufwand für eine Anforderung a_i wobei i mit für die Nummer der Anforderung steht, also von 1 bis n, mathematisch ausgedrückt: $i = [1, n]$. Der Aufwand für die Basis bezeichnen wir als a_b. Demnach ist der Gesamtaufwand für die Software:

$$A = a_b + \sum_{i=1}^{n} a_i$$

Die Abbildung 6 soll dies verdeutlichen.

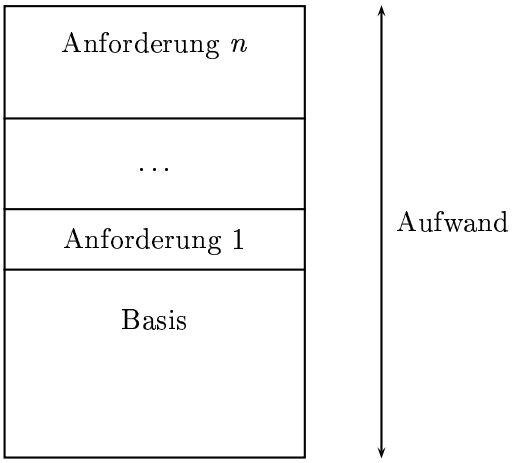

Abbildung 6: Aufwände

4.2 Komplexität

Wir haben nun gesehen, dass die Planung des Aufwands eine Software zu erstellen nicht ganz so trivial ist wie sie zunächst erscheint. Die Komplexität verstärkt diesen Effekt nochmals.

Im Normalfall ist eine Software die wenige Anfordrungen erfüllen muss weniger Komplex als eine die viele Anforderungen erfüllen muss, vorausgesetzt wir gehen von einer durchschnittlich komplizierten Anforderung aus. Eine Software die nur aus komplizierten Anforderungen besteht ist offensichtlich auch komplexer als eine die nur aus trivialen Anforderungen besteht. Dies ist aber nicht das grösste Problem.

Verschiedene Anforderungen haben mehr oder weniger Abhäigkeiten zueinander, z.B. sind das gemeinsam genutzte Daten oder ein sequentieller Ablauf. Bei einer Software A mit wenigen Anforderungen n_A gibt es wenige Abhängikeiten und eine kleine Basis d_A. Eine Software B mit vielen Anforderungen n_B gibt es somit auch viele Abhängikeiten und eine grosse Basis d_B. Somit können wir also sagen, dass n_B sehr viel grösser ist als n_A (sprich $n_1 << n_B$). Die Komplexität O wächst aber Aufgrund der wachsenden anzahl Abhängigkeiten und der stetig grösser werdenden Basis nicht linear, sondern exponentiell. Somit erhalten wir eine Komplexität von $O(e^n)$. Die Abbildung 7 zeigt diese exponentielle Ansteigung der Komplexität bei linearer Ansteigung der Anforderungen.

5 Wohin?

Mit den Anforderungen steht und fällt die Software. Keine Anfoderungen, keine Software. Eine Software zu erstellen ist wie eine Reise zu unternehmen.

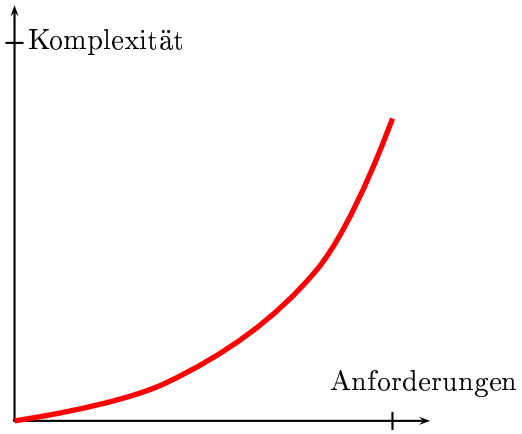

Abbildung 7: Anforderungen und Komplexität

Bevor man ins Auto oder in die Bahn steigt überlegt man sich zuerst wohin die Reise gehen soll. Tut man dies nicht, dann sollte man sich auch nicht darüber beklagen, wenn der Zielort nicht gefällt.

Eine Software wird nicht nur so zum Vergnügen erstellt, jedenfalls nicht von einer gewinnorientierten Firma. Die Software soll ja eine Zielgruppe ansprechen und vor allem einen Zweck erfüllen. Hobbisten stehen nicht so im Zwang der Finanzen oder des Marketing, einen sinnvollen Zweck der Software wird aber durchaus verfolgt.

Auf jeden Fall ist es besser sich richtig Gedanken über das Ziel zu machen anstelle von ständigen Richtungsänderungen. Zu oft wird dies ignoriert oder unterschätzt. Die Verlockung schnell mit der Umsetzung zu beginnen ist oft sehr gross. Die Konsequenzen sind dann entsprechend schlimm, da Anforderungen die erst spät einfliessen sehr schwierig (und dann auch teuer) sind.

Die Abbildung 8 soll verdeutlichen, wie sich späte Anforderungen auf den Aufwand auswirken.

Zu Beginn des Projekts haben Anforderungen einen relativ kleinen Einfluss auf den Aufwand. Natürlich bedeutet jede neue Anfoerderung mehr Aufwand, die Grafik zeigt neu dazugekommene Aufwände, die zu Beginn noch nicht bekannt waren. Ist das Projekt noch nicht weit forgeschritten so haben neue Anforderungen einen kleinen Einfluss. Zuletzt hinzugekommene Anforderungen sind fatal. Diese Mehraufwände können so erheblich werden, dass die Projektdauer auf das Doppelte oder mehr ansteigt von den Projektkosten ganz zu schweigen.

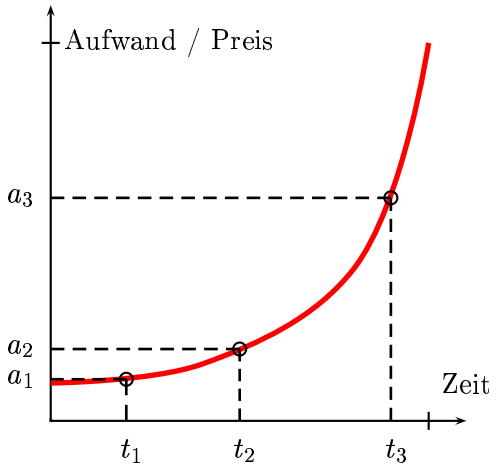

Abbildung 8: Zeit und Aufwand

Teil III
Die Entwicklung

Dieses Kapitel beschäftigt sich nun mit der Entwicklung von Software. Wie schon zuvor erwähnt, sollte dies nicht verwechselt werden mit Forschung oder gar Produktion von Software. Es wird auch gezeigt, welchen Stellenwert Projekte und Prozesse haben. Als Beispiel wird kurz auf das Open Source Entwicklungsmodell eingegangen.

6 Projekte

Das Wort *Projekt* wird oft mystifiziert. Viele Legenden existieren um diesen sagenumwogenen Ablauf der Entstehung von Software. Es gibt zwei weitverbreitete Meinungen über Projekte, die ich gerne zeigen und relativieren möchte. Selbstverständlich gibt es noch weitere Meinungen, doch möchte ich die folgenden zwei kurz erörtern:

1. Projekte sind schwierig, problematisch in der Handhabung. Sie sind komplex, kaum zu überschauen und man muss immer auf Hut sein um sich nicht zu verirren.

2. Projekte sind dazu da sich zu profilieren. Der Abschluss eines Projekts ist nicht so wichtig wie die Grösse, er jeweilige Start und die Präsentation.

Die erste Meinung tritt vor allem dann auf, wenn entsprechende Personen, Projektleiter wie andere, überfordert sind. Das entsprechende Projekt ist

möglichweise bereits ausser Kontrolle geraten, der Projektstart wurde nicht kontrolliert durchgeführt oder die Umsetzung wurde zu früh vorangetrieben. Dies und andere sind mögliche Ursachen.

Die zweite Meinung tritt vor allem dort auf, wo sich Leute befinden, die Ambitionen bezüglich der Karriere haben, sich überschätzen oder möglicherweise mit chaotischen Arbeitsmethoden.

Im Grunde genommen liegt die Wahrheit zwischen den beiden, oben erwähnten, Extremen. Der "gesunde" Menschenverstand ist hier ein erfolgreiches Hilfmittel. Vorausdenken ist die Devise, *Agieren* statt *Reagieren* heisst das Gebot der Stunde. Dies ist keine Anleitung für erfolgreiche Projekte, es soll lediglich aufgezeigt werden, dass man vor dem Begriff *Projekt* keine Angst haben sollte, jedoch durchaus Respekt.

Die Wichtigkeit eines Projekts hat eine noch grössere Tragweite. Es hat psychologische Einflüsse auf die Menschen, die in einem solchen arbeiten. Definierte Rahmenbedingungen (Start und Ende eine Projekts, Zeit und Budget) und ein Zugehörigkeitsgefühl sind nicht zu unterschätzen, sie haben einfluss auf die Motivation, einer der wichtigsten Einflüsse überhaupt für ein erfolgreiches Projekt.

7 Prozesse

So wie das Projekt einen Rahmen vorgibt, so bestimmt der Prozess den Ablauf des Selbigen. Der Prozess definiert, welche Dokumente von welchen Personen geschrieben werden, wie die Qualitätssicherung durchgeführt wird und auch wie das Erstellen der Software geschehen soll. Der Prozess ist vergleichbar mit einer Konvention, die besagt was, wie, zu welchem Zeitpunkt getan wird. Dies schafft eine gewisse Ordnung und Transparenz, verhindert Wildwuchs. So wird auch eine Infrastruktur geschaffen, damit nicht jeder das Rad von Neuem erfindet und die Varianz überhand nimmt.

Es gibt verschiedene Softwareentwicklungs-Prozesse. Die Meisten beruhen allerdings auf einem der folgenden Prinzipien:

- Wasserfall
- Iterativ
- Ungeordnet / Unstrukturiert
- Open Source

Werfen wir zunächst einen Blick auf das Wasserfallmodell (Abbildung 9). Jeder Schritt wird separat für sich ausgeführt. Ist er abgeschlossen und zum nächsten Schritt weitergegangen, so wird nicht wieder zurückgeschaut. Fehler die sich früh eingeschlichen haben, werden so kaum mehr verbessert und bis zum Schluss weitergezogen. Zum Teil wurden die einzelnen Schritte

von separaten Teams durchgeführt, die Qualität war dann dementsprechend schlecht. Eine Variante des Wasserfallmodells zeigt die Abbildung 10.

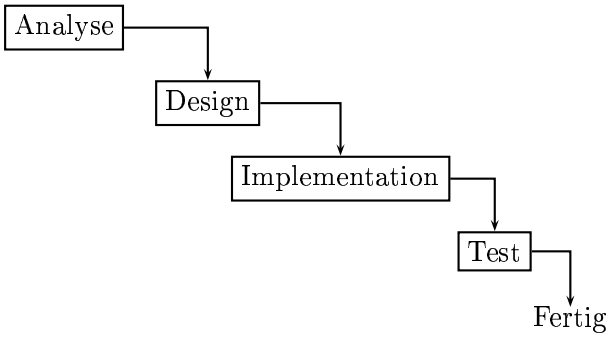

Abbildung 9: Wasserfallmodell

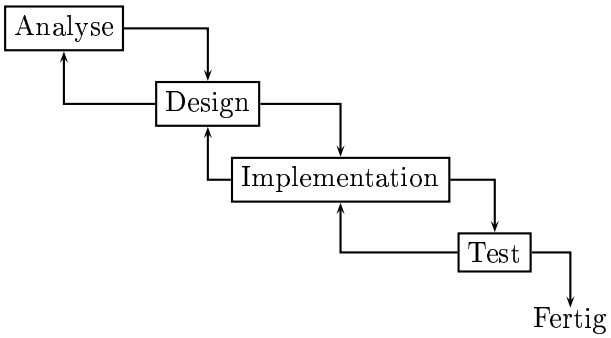

Abbildung 10: Variante des Wasserfallmodells

Trotz gewisser Gemeinsamkeiten doch recht unterschiedlich: der iterative Prozess (Abbildung 11). Jede Iteration besteht für sich als kleiner Wasserfall, jedoch durch die aufeinanderfolgenden Iterationen mit einem gewissen Controlling. Fehler, die erkannt werden, können in der folgenden Iteration behoben werden. Trotz dem iterativen Vorgehen, wird der Schwerpunkt in den ersten Iterationen auf Analyse und Design gesetzt. In späteren Iterationen dann auf Implementation und Test. So kann verhindert werden, dass das Projekt sich ewig im Kreis dreht. Ein bekannter Vertreter ist *RUP*.

Risikomanagement ist einer wichtigsten Punkte für den Ablauf eines Projektes, unabhängig vom Prozess. Lesen Sie mehr dazu weiter unten. Auf den *Open Source* Prozess wird im nächsten Kapitel eingegangen.

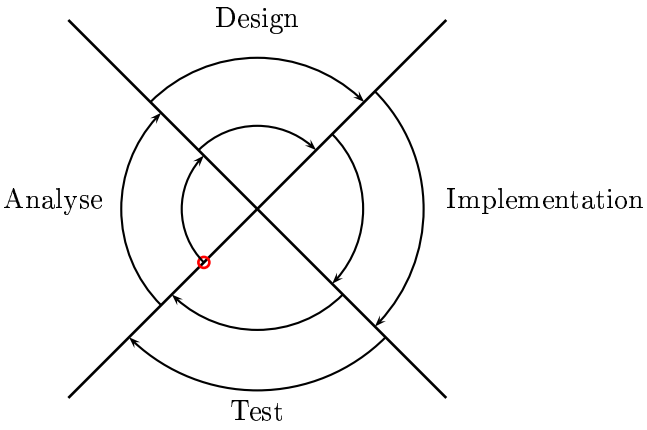

Abbildung 11: Iteratives Modell

8 Open Source

Als Beispiel für einen Entwicklungsprozess möchte ich die Open Source Szene hinzuziehen. In ihr können verschiedene Abläufe betrachtet werden, ohne die "Verschmutzung" des kommerziellen Druckes. Diese Software unterliegt einem Prozess, den man wirklich als Entwicklung in Reinkultur bezeichnen kann. Das Buch [1] beschreibt dies ausführlich.

Jemand, eine Person oder eine kleine Gruppe von Personen, schreibt ein Programm. Oft ist es nicht ein vollständiges Programm, sondern eine Entwicklerversion mit minimaler Funktionalität. Das Programm, oder deren Verwendung, findet Anklang bei verschiedenen Leuten, die ihrerseits die Entwicklung unterstützen. Diese Unterstützung geschieht in vielerlei Form: Bugreports (ein Report über Softwarefehler), Patches (engl. *patch* = Pflaster, Verband; Behebung eines Fehlers als Sourcecodefragment), Dokumentation, etc. Diese Beiträge werden vom *Maintainer* (wiederum eine oder mehrere Personen) ins Projekt eingefügt. Auf diesem Weg entwickelt sich die Software Schritt für Schritt.

Es entstehen nach einer gewissen Anzahl von Beiträgen, sog. *contributions*, eine neue Entwicklerversion, die dann getestet und wiederum mit neuen Beiträgen weitergebracht wird. Dies geschieht solange, bis die Maintainer das Programm für stabil genug halten um einen *stable release* zu generieren. Dieser ist im Wesentlichen die letzte, getestete Entwicklerversion. Dabei spielen marketingstrategische Entscheide *keine* Rolle. Die Software wird solange weiterentwickelt bis sie ein genügend stabiler Zustand erreicht hat.

Während der gesamten Entwicklung wird immer das Konzept *KISS* vor Augen gehalten. Nicht nur kleine Programme werden auf diesem Weg entwickelt, es gibt zahlreiche grosse Programme (Linux, Gnome, KDE, etc.).

Dies ist nicht eine vollständige Beschreibung aller Eventualitäten, gibt aber eine kurze Übersicht und es ist trotz dieses kurzen Ausflugs ein Pro-

zess erkennbar. Sehr schön zu sehen ist die Entwicklung, die stattfindet. Es werden (praktisch) keine unbenutzten Funktionen eingebaut, nur weil sie marketingstrategische Relevanz haben.

Wird versucht eine Software mit Ansätzen der Software-Produktion (siehe weiter oben), wird diese nach guter alter darwinistischen Evolutionstheorie früher oder später ausscheiden. Eventuell wird ein neuer Zweig entstehen (*fork*), in welchem eine Evolution statfinden wird und das ausgeschiedene Programm als Basis dient.

Teil IV
Risiken

In diesem Kapitel werden verschiedene Risiken erörtert, die während der Entwicklung einer Software auftauchen (können). Es auch Wege aufgezeigt, wie diese Risiken vermieden oder gelöst werden können. Dieses Kapitel enthält *keine* Patentlösung für alle Probleme der Softwareentwicklung, es beschäftigt sich lediglich mit den genannten Risiken.

Die hier genannten Risiken sind in sehr vielen Projekten zu finden. Leider werden verschiedene Risiken oft zu wenig ernst genommen. Interessant ist, dass wenn sich Risiken manifestieren, dass im nächsten Projekt genau die gleichen Risiken wieder unterschätzt werden. Dies widerspricht in jeder Hinsicht der Weisheit: *"Fehler sind die besten Lehrer."* Dafür kann ich nur zwei Gründe erkennen:

- Menschliches Versagen
- Für jedes Projekt einen neuen Projektleiter ohne entsprechenden Erfahrungen

9 Fehler

Fehler sind eine unschöne Sache. Schlimmer noch: es kommt auf den Zeitpunkt an, wann sie entdeckt werden. Je später im Projekt ein Fehler entdeckt wird um so Aufwendiger ist seine Behebung. Die Abbildung 12 soll verdeutlichen, wie hoch der Aufwand wird Bezug auf die Phase des Projekts und den Art des Fehlers. Die Achse *Fortschritt* zeigt die Schwerpunkte in einem Projekt mit iterativem Prozess. Im klassischen Wasserfallmodell entspricht dies den einzelnen Stufen. Die Achse *Fehler* zeigt die Art des Fehlers. Ein *Analysefehler* ist ein Fehler der gemacht wurde in der Analysephase. Dessen Kurve zeigt, dass je später er entdeckt wird, um so grösser ist der Schaden. Analog für die beiden andern Kurven. Deutlich zu sehen ist, dass Implementationsfehler im Gegensatz zu einem Analysefehler sich nicht so stark auf den Aufwand auswirkt. Also sollte vor allem in den Phasen der Analyse und Design sorgfältig gearbeitet und Nichts überstützt werden. Hier bietet sich an häufige Reviews durchzuführen.

10 Späte Anforderungen

Wie bereits im Kapitel 5 erwähnt wurde, haben Anforderungen die erst später in ein Projekt einfliessen grosse Auswirkungen auf den Gesamtaufwand. Die eigentliche Lösung für dieses Problem heisst: *Prozess*. Die meisten Entwicklungsprozesse sehen es vor, Anforderungen in der Anfansphase des Pro-

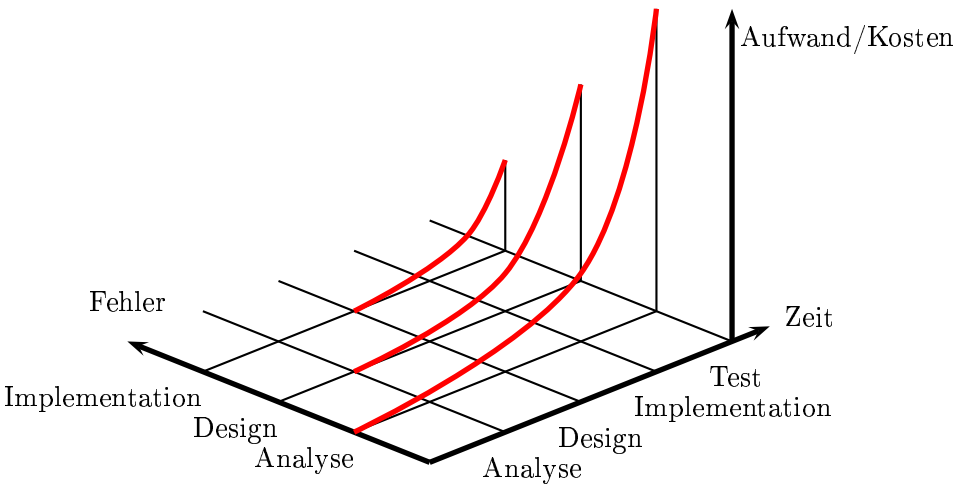

Abbildung 12: Fehler

jekts zu sammeln und zu bewerten. Spätere Anforderungen werden über einen definierten Ablauf (*Change Request Management*), ins Projekt eingebunden oder verworfen.

Interessant ist, dass die Spezifikationen der Prozesse eigentlich nur den "gesunden" Menschenverstand versuchen nachzubilden. Im konkreten Fall heisst dies, dass jemand der eine Software entwickeln möchte, sich genügend mit dem Umfeld des Problems auseinandersetzen muss. Klingt logisch, wird aber oft vergessen. Ein weiterer Punkt der im Prinzip logisch erscheint, ist, dass mangelnde Anforderungen nicht mit Softwareentwicklungsmethoden komplensiert werden können. Die besten Softwareentwickler können keine Software entwickeln, wenn dessen Zweck nicht bekannt ist.

11 Abhängigkeiten

Soll eine Software entwickelt werden, so sind Abhängigkeiten unvermeidbar. Sie sind deshalb ein Risiko, da sie nicht unmittelbar unter eigener Kontrolle sind. Es ist denkbar, dass eine Firma, zu der eine Abhängigkeit besteht, ein Produkt einstellt, Konkurs anmelden muss oder der einzige Hersteller von benutzten Softwarewerkzeugen, sog. *Tools*, ist. Abhängigkeiten können in verschiedenen Formen auftreten, z.B.:

- zum Betriebssystem
- zu diversen Softwarebibliotheken
- zu Programmiersprachen

- zu Entwicklungsumgebungen (IDEs)
- zu diversen Entwicklungswerkzeugen
- zu Personen
- zu externen Firmen

Wie schon erwähnt sind nicht immer alle diese Abhängigkeiten immer vermeidbar. Sie machen sich dann negativ bemerkbar, wenn Veränderungen eintreten, und die eignene Softwareentwicklung durch die Abhängigkeit in Mitleidenschaft gezogen wird. Mit anderen Worten können zusätzliche, späte Anforderungen entstehen, die nicht geplant werden und sehr hohe Kosten verursachen können. Im Extremfall können Veränderungen das vorzeitige Ende eines Projektes bedeuten.

Die einzige Massnahme um dieses Risiko zu vermindern, ist Abhängigkeiten zu vermindern. Es folgen ein paar Gedanken dazu, wie dies bewältigt werden könnte. Natürlich gibt es auch hier kein garantiertes Rezept.

11.1 Kleinste gemeinsame Basis

Proprietäre Dateiformate sind recht weit verbreitet, jedoch wenn es auf die Interoperabilität von verschiedenen Tools ankommt, machen diese einen Strich durch die Rechnung. Kein Tool kann alles, also ist man früher oder später auf diese Interoperabilität angewiesen.

Trotz fleissigen Sicherheitskopien, den *Backups*, ist ein Absturz der Infrastruktur, angefangen vom einzelnen Rechner über einen Server bis hin zum Netzwerk, durchaus denkbar. Bei Verwendung propritärer Dateiformate, braucht es viel Zeit um die Softwareentwicklung wieder aufzunehmen.

Bei offenen Standards, z.B. ASCII-Dateien, können die Daten im schlimmsten Fall immer noch mit einem simplen Texteditor gerettet werden. Die Interoperabilität ist auch gewährleistet, wenn alle verwendeten Tools ASCII-Dateien lesen und schreiben können. Die Chance für eine Unterstützung durch Tools ist auf jeden Fall bei ASCII-Dateien grösser als bei einem beliebigen andern Format.

11.2 Programmiersprache

Dies ist kein Votum gegen oder für bestimmte Programmiersprachen, diese Glaugenskriege werden an anderen Stellen geführt. Ein vielgelobter Ansatz ist: *"Für jede Arbeit die geeignete Sprache"*. Auf den ersten Blick leuchtet dieses Prinzip jedem ein. Ein Blick hinter die Kulissen bringt gewisse Gefahren ans Tageslicht.

Zum Einen ist zu beachten, dass die eingesetzte Programmiersprache nicht nur von einem einzigen Hersteller unterstützt wird. Meldet dieser Konkurs an oder ändert er bezüglich des Produkts radikal seine Strategie, so

findet man sich mit der eigenen Softwareentwicklung plötzlich in einer Sackgasse.

Ein anderes Problem des *single source* (ein einizger Hersteller) ist auch, dass in diesem Fall keine Konkurrenz vorhanden ist, die eine Weiterentwicklung der entsprechenden Werkzeuge (Compiler, Linker, etc) vorantreibt, ein technologisches Manko.

Vorzugsweise entscheidet man sich für eine Programmiersprache, die auf vielen verschiedenen Platformen, Entwicklungswerkzeuge von verschienen Herstellen verfügbar sind.

11.3 Softwarebibliotheken

Der Einsatz einer bestehenden Softwarebibliothek kann unter Umständen viel Zeit und Aufwand ersparen. Die gleichzeitige Verwendung mehrerer *libraries* im gleichen Projekt kann aber zu Problemen führen, z.B. redundate Funktionen die auf andern Datentypen basieren. Eine sorgfältige Auswahl ist also unerlässlich. Eventuell kann es auch Sinnvoll sein, ein paar Funktionalitäten auch selbst zu implementieren.

Ein anderes Problem mit Libraries kann auftreten, wenn sog. *closed source* libraries verwendet werden. Die Flexibilität der eignen Entwicklung wird eingeschränkt und oft wird es unerlässlich den Support der Herstellerfirma in Anspuch zu nehmen. Dies bedeutet einen Supportvertrag oder Supportgebühren, auf jeden Fall finanzielle Konsequenzen.

Die Abhängigkeit zu einer library ist auch nicht zu unterschätzen. Hat man sich in einem Projekt für eine entschieden ist es schwer und aufwändig diese später wieder zu ersetzen, falls dies nötig würde.

11.4 Betriebssysteme

Interessant ist die Abhängigkeit zum Betriebssystem. Die heute eingesetzte Software beschränkt sich in den häufigsten Fällen auf eine einzige Plattform. Dies muss nicht unbedingt ein Nachteil bedeuten. Der interessante Punkt dabei ist aber, dass Software die *portabel* entwickelt wurde, d.h. möglichst Betriebssystemunabängig, sehr viel höhere Qualität ausweist als *quick and dirty* Lösungen.

Ein anderer Vorteil ergibt sich dadurch, dass für die entwickelte Software ein grösseres potentielles Publikum (sprich: Kunden) angesprochen werden kann.

12 Der Weg ist auch das Ziel

Der Titel dieses Kapitels ist eine Abwandlung des asiatischen Sprichwortes: *"Der Weg ist das Ziel"*. Im Fall der Softwareentwicklung ist dies ein wenig anders. Wie schon weiter ober erwähnt, wird eine Software nicht entwickelt

um des entwickelns Willen (es gibt Ausnahmen), vor allem nicht in der kommeriziellen Softwareenwicklung. Gerade aber in diesem Umfeld wird aber oft nur noch das Ziel betrachtet ohne viel über den Weg dorthin nachzudenken. Ironischerweise wird das Ziel nur über den Weg erreicht.

Um die Sache ein bischen weniger philosophisch zu betrachten: oft ist nur das Ziel im Blickfeld ohne eine Selbstkontrolle oder eine Qualitätssicherung der geleisteten Arbeit durchzuführen. Ein "schönes" Design und eine entsprechende Dokumentation erlauben auch in Zukunft eine Weiterentwicklung der Software. Sind diese Kritierien nicht erfüllt, muss früher oder später die Software von Grund auf neu gemacht werden und dies ist ein viel grösserer Aufwand.

Häufig wird nur über unmittelbar anfallende Kosten, nicht aber mittel- und längerfristige, betrachtet. Jetzt ein wenig teurer, dafür nicht zwei Mal machen, könnte durchaus als Leitfaden dienen.

13 Eile mit Weile

Wer es immer eilig hat, kommt nirgendwo hin. Zu oft wird *rapid prototyping* und Softwareproduktion mit Softwareentwicklung verwechselt (siehe Kapitel 3).

Einen Prototypen zum Produkt zu erheben hat sich als problematisch erwiesen, da die Aufwände in grösserer Form reflektiert werden. Es entstehen mehr Aufwand und Kosten durch die Folgen als durch die schnelle Fertigstellung des vermeindlichen Produktes gespart wird. Mittel- und Längerfristig ist dies ein Eigentor.

Wie auch schon oben erwähnt, kann die Dauer einer Softwareentwicklung nicht immer eingeschätzt werden. Ungleich wie in der Produktion kann die Dauer der Softwareentwicklung nicht durch Veränderung des Prozesses oder duch den Einsatz weiterer oder anderer Tools herabgesetzt werden. Im Klartext heisst dies: nur Technologien einsetzen die schon längere Zeit bekannt und etabliert sind. Für solche Technologien gibt es unterstützende Werkzeuge, es gibt genügend qualifiziertes Personal und keine Überrschungen während des Projektes (sofern keine andern Risiken sich manifestieren).

In diesem Fall muss man sich allerdings davon lösen es als *Softwareentwicklung* zu taxieren. Schlagwörter wie *time to market* oder *just in time* funktioniert mit Softwareentwicklung **nicht!**

14 Nicht für Jedermann

Für jeden Fachbereich braucht es entsprechendes Fachpersonal. Dies gilt selbstverständlich auch für die Softwareentwicklung. Leider taucht immer wieder die Meinung auf, auch unqualifiziertes (dafür günstigeres) Personal

kann mit entsprechend intelligenten Tools eine Software entwickeln. Das einzige was dann aber entwickelt wird ist das Personal, ganz zum Unmut der Vertreter obiger Meinung. Nicht dass das Personal keinen Anspuch hätte, sich weiterzuentwickeln, um eine komplexe Software entwickeln zu können braucht es aber erfahrenes Personal. Idealerweise arbeiten erfahrene Leute mit weniger Erfahrenen zusammen um Nachwuchs zu fördern. Um es mit einem Sprichwort auf den Punkt zu bringen:

"A fool with a tool is just a fool!"

(*"Ein Dummkopf mit einem Werkzeug ist trotzdem ein Dummkopf!*)

Oft vergleiche ich die Softwareentwicklung mit der Automobilbranche, da viele Leute mit den dortigen Begriffen besser zurecht kommen. Wer würde schon ein Auto kaufen, das von einem Fahrradhersteller gebaut wurde. Nicht dass die Leute beim Fahrradhersteller schlecht wären, sie haben nur mit dem Bau eines Autos (noch) keine Erfahrung.

Bei der Softwareentwicklung ist es, analog zum Beispiel mit dem Auto, auch so, dass die Software, gemacht von Unerfahrenen, nie richtig funktionieren würde, und somit auch (wenn möglich) gemieden. Ganz zu schweigen von den finanziellen und zeitlichen Resourcen, die verbraucht werden.

Zugegeben ist es nicht einfach ensprechendes Fachpersonal zu finden. Auf jeden Fall lohnt es sich die bestehende Belegschaft entsprechend auszubilden. Hier ein wenig Kosten einsparen zu wollen dann geht die Rechnung breits Mittelfristig schon nicht mehr auf.

Wer zu diesem Thema weitere Literatur konsultieren möchte, dem sei [2] und [4] empfohlen.

Teil V
The Good, the Bad and the Ugly

Dieses Kapitel soll Dinge aufzeigen, die die Softwareentwicklung für so Viele Leute interessant machen. Es soll aber auch die Schattenseiten und die hässlichen Dinge aufzeigen.

15 The Good

Ein zweifellos interessanter Aspekt an der Softwareentwicklung ist, dass in relativ kurzer Zeit viele neue Technologien entstehen. Langweilig wird es also in absehbarer Zeit sicher nicht und für Abwechslung ist auch gesorgt.

Der Begriff *Informatik* beschreibt ein sehr grosses Spektrum an Gebieten und Themen. Das Gebiet ist derart gross, dass man auf keinen Fall sich mit allen Aspekten befassen kann. Es ist sogar so gross, dass man sich das Teilgebiet aussuchen muss/kann/darf.

Nicht zuletzt ist es sehr interessant, da es in den Industrieländern die Hardware sehr günstig zu kaufen gibt. Mittlerweile hat im Durchschnitt jeder Haushalt min. ein Gerät. Die Hürde sich mit der Informatik zu beschäftigen ist dadurch wesentlich kleiner als auf vielen andern Gebieten von Wissenschaft und Technik.

16 The Bad

Das Schlechte and der Softwareentwicklung ist, dass sich die oben genannten, und auch andere, Risiken zu oft manifestieren. Untersuchungen zu folge ist die Situation sogar nocht schlimmer (siehe Abbildung 13). Dies ist nur eine

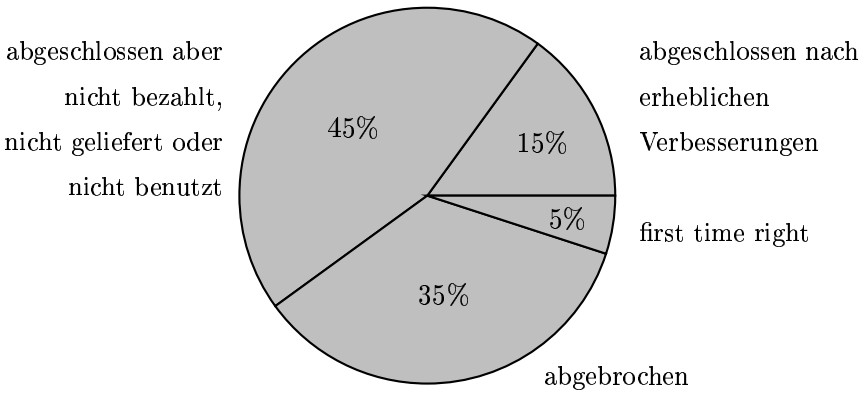

Abbildung 13: Projekt-Statistik

Statistik [3], d.h. die Zahlen dürfen nicht als Absolutwerte herangezogen werden. Jedoch wenn wir sehr grosszügige Toleranzen für einzelnen Werte einführen, sind diese Zahlen immer noch erschreckend. Dies führt zu einem schlechten Ruf der Softwareentwicklung. Wie oben auch schon erwähnt, ist es in den meisten Fällen meschnliches Versagen, das zu solchen Verhältnissen führt.

17 The Ugly

Der hässliche Teil der Softwareentwicklung zeigt sich immer dann, wenn der Entwickler zum Sklaven wird. Wie schon mehrfach erwähnt kann die Zeit, die benötigt wird für eine Entwicklung, nicht immer abgeschätzt oder vorausgesagt werden. Wenn nun noch Druck wegen einer *deadline* (Verkaufsstart oder ähnliches) gemacht wird, dann Endet meistens die Entwicklungsarbeit und mutiert zum Gebastel. Druck ist nicht grundsätzlich etwas Schlechtes, jedoch hat es auch Grenzen. Zu oft werden diese Grenzen überschritten. Ist dieser Zeitpunkt gekommen, tauchen vermehrt Sprüche auf wie: *"Die Dokumentation kann später noch gemacht werden, jetzt müssen wir uns um die Features kümmern."* Die Folgeschäden sind kaum zu beziffern.

Die fehlende Dokumentation ist allerdings nicht nur eine Folge einer Marketingstrategie oder dem Unvermögen von vermeindlichen Projektleitern, sondern auch von der Nachlässigkeit von Entwicklern. Diese beschäftigen sich oft lieber mit dem Programm (Design, Programmierung) als mit Dokumentation. Leider, so meine Erfahrung, ist es die fehlende Kommunikation untereinander und auch die fehlende Bereitschaft zum gegenseitigen Verständnis.

Alle diese hässlichen Seiten führen oft zum Ärger und Stress für alle Beteiligten. Dieser Ärger endent (siehe Statistik) oft in Mehrkosten und zeitliche Verzögerungen.

Teil VI
Fazit

Dieses Kapitel fasst die wichtigsten Punkte der vorangegangenen Kapitel zusammen. Es zeigt nochmals solche Dinge die nicht vergessen werden solten.

Vorausdenken Das asiatische Sprichwort: *"Bevor Du beginnst, bedenke das Ende."* dient sehr gut als Leitsatz. Man sollte sich immer im Klaren darüber sein, welche Ziele man verfolgt, diese nicht aus den Augen lassen.

- Handelt es sich um eine *Entwicklungs*-Charaktor oder hat es eher einen *Produktions*-Charakter?
- Was soll die Software für einen Zweck erfüllen?

Dies sind Fragen die man sich zu Beginn des Projkets gut überlegen sollte.

Vorgehen Jeder Prozess hat seine Stärken und seine Schwächen, genau so wie sie jeder Mensch hat. Bei den Menschen kommen zusätzlich noch Faktoren wie Motivation und Tagesform hinzu. Um im Projekt klare Verhältnisse zu schaffen, die Schwächen ein wenig zu Kompensieren und die Stärken zu fördern empfiehlt sich die Verwendung einer *klar definierten* (und strukturierten) Vorgehensweise.

Risikomanagement Behalten Sie immer die Übersicht über die Risiken. Versuchen Sie die Risiken in den Griff, d.h. unter Ihre Kontrolle, zu bekommen. Es gibt nichts Schlimmeres, als wenn sich Risiken bemerkbar machen und Sie nur nor reagieren können, an Stelle zu agieren. Versuchen Sie:

- Abhängigkeiten zu minimieren.
- Keine allzu exotischen Werkzeuge und Programmiersprachen einzusetzen.
- Ein Team zusammenzustellen, in dem auch erfahrene Leute mitarbeiten.
- Nicht in zu grosser Eile ein Ziel erreichen zu wollen. Die Chance ist sehr gross es zu verfehlen oder darüber hinauszuschiessen. Sich Zeit zu nehmen um Vorauszudenken ist **immer** eine gute Investition!

Ein vorausgesehenes Risiko birgt keine bösen Überraschungen!

Motivation Mit grosser Motivation ist scheinbar jedes Ziel erreichbar. Achtung: Geld ist nicht immer ein Motivationsfaktor!

Fachliche Kompetenz In der Effizienz zwischen "guten" und "schlechten" Fachleuten liegt meistens ein mehrstelliger Faktor.