Abstraktion als Denkwerkzeug

– das mit Abstand wichtigste Werkzeug der Informatiker –

Gebhard Greiter, 2011

In diesem Aufsatz soll erklärt werden, was man unter ***Abstraktion*** versteht und warum die Fähigkeit, zu abstrahieren, den Menschen – und insbesondere den erfolgreichen Software Ingenieur – zu dem macht, was ihn auszeichnet.

Geschichtlich betrachtet begann der Mensch zu abstrahieren, als er zum ersten Mal Bilder an Höhlenwände malte und als er zum ersten Mal begriffen hat, was eine Zahl ist. Seitdem sind Jahrtausende vergangen, in denen der Mensch seine Fähigkeit, zu abstrahieren, mehr und mehr erkannt und kultiviert hat. Er tat dies aber bis in die jüngste Vergangenheit hinein eher unbewusst.

Erst etwa 1950 hat sich das – aufbauend auf der Erfindung des Computers – schlagartig geändert. Ich meine den Zeitpunkt (etwa 1950), zu dem der Mensch begann, Software zu schreiben in der Absicht, die Arbeit der Computer über Programme zu steuern.

Welchen schier unglaublichen ***Quantensprung in der Entwicklung des Menschen*** dies bewirkt hat, müssen wir uns klar gemacht haben, bevor wir begreifen werden, welchen Wert Abstraktion als Denkwerkzeug hat:

Ein Computer ist eine Maschine, die man startet, indem man einen Schalter umlegt um sie mit Strom zu versorgen. So zum Leben erwacht wird das Herz des Computers – ein heute als Chip realisierter Prozessor – taktgetrieben in jedem durch diesen Takt gegebenen Schritt eine im Prozessor selbst existierende relativ kleine Menge binärer Variablen – man nennt sie Speicherzellen – aufsuchen, lesen und sofort wieder schreiben. Dies bewirkt elektronische Impulse, die (wie Morsesignale) auf Leitungen fließen, die den Prozessor mit seiner Umgebung verbinden: mit einer Umgebung, die

- noch 1950 zunächst aus blinkenden Lämpchen bestand,
- etwas später aus Maschinen, die Lochstreifen oder Lochkarten lesen und stanzen konnten
- und die heute aus *sämtlichen* über Strom- oder Funknetze an den betrachteten Computer angeschlossenen anderen Maschinen besteht (insbesondere aus über die gesamte Welt hin verteilten anderen Prozessoren).

Schon jetzt also – nur etwa 60 Jahre nachdem man erste Software zu schreiben begann – sind Milliarden von Maschinen, die Prozessoren und andere elektrisch oder elektronisch betriebene Geräte darstellen, zusammengewachsen zu einem einzigen großen Computer, den man ***das Internet*** nennt (das **WWW = World wide Web**).

Neben ihm existieren kleinere, aber dennoch ganz analog aufgebaute andere Computerumgebungen: solche, die (grundsätzlich oder nur zeitweise) nicht ans Internet angeschlossen sind. Jeder offline betrieben PC etwa ist typisches Beispiel dafür. Auch er aber besteht heute schon aus keineswegs nur einem einzigen Prozessor. Die Software, ihn sinnvoll zu steuern, ist i.W. die gleiche, mit der man auch das WWW steuert; sie besteht aus Varianten einiger weniger Betriebssysteme wie ***MS Windows*** oder ***Unix*** in Kombination mit Treibersoftware – ***Treiber*** sind gerätespezifische Betriebssysteme, deren jedes um mehrere Größenordnungen einfacher ist als Unix oder MS Windows.

In der Summe lässt sich feststellen: **Das WWW** als eine riesige Menge sich ständig neu zu einem einzigen Computer zusammenfindender Prozessoren **und MS Windows** (als das umfangreichste und komplizierteste aller sie steuernden Betriebssysteme) sind die komplexesten Konstruktionen, die der Mensch je schaffen hat.

Beide sind um viele Größenordnungen komplexer als jedes vom Menschen bis etwa 1950 geschaffene andere System.

Wie nun aber wird solche Komplexität durchschaubar und (fast perfekt) beherrschbar?

Was befähigt uns, zu durchschauen und zu beherrschen, was eine Person alleine nur hinsichtlich ganz weniger Aspekte verstehen kann?

Die Antwort ist: Nur gezielt angewandte Abstraktion macht so was möglich.

De facto aber hat erst die Informatik – Computer Science, wie man im Englischen sagt – uns gelehrt und gezwungen, Abstraktion ***bewusst und sehr gezielt*** einzusetzen. Genau das zu tun scheinen heute aber nur wenige Entwickler hinreichend geübt zu haben.

Die Ignoranz, Abstraktion ihrem Wert nach zu sehen und anzuerkennen, ist selbst unter Informatikern weit verbreitet, und das zunehmend ab etwa 1990 und ungeachtet der Tatsache, dass erst Abstraktion uns aus der 1968 festgestellten ***Software-Krise*** erfolgreich herauszuführen in der Lage war.

Die 1995 von Kent Beck begonnene Diskussion um sog. Agile Methodik etwa dauert bis heute unverändert an und hat dennoch bislang zu nur ganz dünnen Ergebnissen geführt. Solcher Misserfolg ist ganz offensichtlich darauf zurückzuführen (siehe [1]), dass man bis heute an einer allzu wenig abstrakten Definition des Begriffs „Agile Software Development“ festhält. Auf die Idee, diesen Mangel zu beheben, kam auch keiner der Wissenschaftler, die auf einer nur diesem Thema gewidmeten Konferenz im Januar 2002 den Methodenkrieg um Agile mit deutlichen Worten aber eben doch nur mit einem Waffenstillstand beendet haben (Details dazu in [2] und [3]).

David Parnas – jener Wissenschaftler also, der als erster den Wert von Abstraktion erkannt und gepredigt hat – wird sich ob dieser traurigen Situation wundern, wie wenig professionell manche Informatiker, auch solche, die im Hochschulbereich arbeiten, immer wieder agieren.

Die noch so junge Informatik ist ganz offenbar die einzige Wissenschaft, die vor dem Problem steht, dass sich hier neue Ideen schneller einstellen, als die Wissenschaftler sie zu verarbeiten in der Lage sind. Wie sonst sollte erklärbar sein, welche Perlen der Erkenntnis hier in welch kurzer Frist schon in Vergessenheit geraten: ***das Wissen um den Wert von Abstraktion*** etwa.

Das Vergessen dieser Perle mag sogar selbst Folge zu wenig abstrakten Denkens sein:

Tatsache jedenfalls ist, dass selbst Parnas Abstraktion vor allem im Umfeld von Programmiersprachen diskutiert hat – genau dort also, wo sie bis heute überlebt hat (man denke an die Begriffe *Class* und *Interface* aus Java und C# oder die Wortsymbole *private*, *public,* die in sämtlichen objekt-orientierten Programmiersprachen das WAS vom WIE trennen und somit Abstraktion unterstützen).

Während der um das Jahr 1980 geführten Diskussion um die Idee sog. ***abstrakter Daten*** und ***abstrakter Datentypen*** (***abstrakter Software*** also) ist uns aber klar geworden, dass Abstraktion vor allem im *Vorfeld* von Programmierung – dort also, wo es um Problemanalyse und Lösungsentwurf geht – in ganz besonders hohem Ausmaß Klarheit zu schaffen in der Lage ist.

Die Entdeckung dieser Tatsache hat damals zur Lösung der 1968 festgestellten Software-Krise geführt (es war zu jener Krise gekommen, da man damals erkennen musste, dass Software zunehmend komplexer wurde: so komplex, dass man fürchten musste, sie nicht mehr beherrschen zu können).

Vom Detail abstrahierende Programmiersprachen (man nennt sie „höhere“ Programmiersprachen) und zunehmend abstraktere und somit auch einfachere Schnittstellen hin zum Anwender der Betriebssysteme haben der Krise schließlich ein Ende gesetzt.

Der Versuch aber, neben der Welt der Programmiersprachen eine Welt formaler Designsprachen aufzubauen ist kläglich gescheitert: Als nahezu einziges Ergebnis dieser Bemühung hat normierte graphische Notation überlebt (UML, dann auch BPMN).

BPMN ist eine Spezialsprache zur Beschreibung von Worflow, beschreibt also, wie man Anwendungsbausteine kombinieren kann um ganze Anwendungen zu erhalten.

UML dagegen, gedacht als universell einsetzbare Sprache zur Spezifikation von Software und ihrer Interaktion mit dem Anwender (dem User), wird ihrem Anspruch keineswegs gerecht. Was natürlich kein Wunder ist, denn wie soll das Komplexeste, das der Mensch je ersonnen hat – eben Software – durch einfache Bilder ausreichend genau beschreibbar sein? Dies gilt trotz der Tatsache, dass die Erfinder von UML sich bis heute weigern, das zuzugeben.

Die beste Möglichkeit, Softwaredesign zu dokumentieren, ist nach meiner Erfahrung ein an Karteikarten erinnernder Ansatz: **Specification and Collaboration Cards** (**SCC**).

Man geht da vor wie folgt:

- Grundregel ist: Jedes Konzept – und sei es eine ganze Applikation – wird zunächst so beschrieben, dass dieses Beschreibung auf etwa einer Seite Platz hat (diese Seite ist das, was ich als „Karteikarte“ bezeichne).

- Teil dieser Beschreibung ist eine Zerlegung in Subkonzepte, die man dann wieder auf eben dieselbe Weise beschreibt.

- Vorteil dieses Verfahrens: Es zwingt zur Abstraktion, denn da die Beschreibung zunächst nur etwa eine Seite Text umfassen darf, wird man gezwungen, sich auf das Wesentliche zu konzentrieren und geeignete Subkonzepte zu definieren, die dann hin zu allen anderen notwendigen Details führen.

- Weiterer Vorteil: Es bleibt dem Leser überlassen, wie weit er in Details einsteigen möchte.

- Und noch ein Vorteil: Man kann solches Design – da es sich auf hierarchisch zueinander angeordnete Specification Cards verteilt – leicht in HTML präsentieren und, wenn geeeignete Regeln zur Notation von Namen eingehalten werden, sogar automatisch über Links verknüpfen und auf formale Vollständigkeit hin prüfen.

- Jede Karte kann, wo hilfreich, durch Bilder illustriert sein (hin und wieder sogar durch ein UML Diagramm).

- Nicht zuletzt ist derart aufgeschriebenes Design extrem änderungsfreundlich (also gut wartbar). Zu Projektanfang, wo noch wenig Details erarbeitet sind, werden derart gegliederte Entwurfspapiere sogar von Managern gerne gelesen.

- Als Collaboration Card bezeichnet man eine Karte, deren Gegenstand die Interaktion von Systemkomponenten, von User und System, oder von Menschen ist, die in bestimmten Rollen arbeiten (ein Beispiel findet sich hier).

Nochmals: Diese Technik zwingt zur Abstraktion und garantiert gute Wartbarkeit sämtlicher Konzeptdokumentation. Ich habe sie seit 1995 mit weit mehr Erfolg eingesetzt als die sonst übliche aufsatzartige Aufschreibung von Design (aufsatzartig gegebenes Design ist alles andere als gut wartbar).

Note: Das Akronym **SCC** scheint kaum bekannt, aber einen Spezialfall dieser Technik – sog. *Class Responsibility Cards* – wurde erstmals beschrieben durch Ivar Jacobson in seinem sehr lesenswerten Buch „Object Oriented Software Engineering: A Use Case Driven Approach" (1992).

Abstraktion im Software Engineering bedeutet, jedes Konzept, das man zu diskutieren hat (i.A. wird das ein System oder Service sein) in zwei Teile zu zerlegen:

- Den ersten, er sollte so klein wie nur irgend möglich sein, nennt man die Black Box Sicht auf das Konzept (auch: das ***abstrakte*** Konzept, den ***abstrakten*** Service, oder das **WAS**).

- Der gesamte Rest **–** das sog. ***Implementierungsgeheimnis*** (auch das **WIE** genannt) **–** braucht den Anwender nicht zu interessieren. Dieser Teil muss austauschbar sein, ohne den Anwender vom Austausch unterrichten zu müssen. Diese Eigenschaft ist mindestens dann wichtig, wenn das Konzept einen Web Service darstellt. Oft kennt der Service Provider in diesem Fall seine Anwender ja gar nicht.

- Java und C# beschreiben abstrakte Services als sogenannte ***Interfaces*** (ein durch diese und ähnliche Sprachen explizit unterstütztes Konzept).

Die Grenze zwischen Implementierungsgeheimnis einerseits und Interface andererseits bestimmt man heute als Software Designer selbst. Man kann zeigen, dass es stets eine eindeutig bestimmte Stelle gibt, an der diese Grenze gezogen sein muss, wenn man sich die Black Box Sicht ***maximal abstrakt*** (also ***maximal einfach aussehend***) wünscht. Jene Stelle zu errechnen ist heute – und sicher noch auf lange Zeit hin – aber nur theoretisch möglich.

Festzustellen bleibt: Der Informatiker tut gut daran, jedes Konzept – sei es ein System, ein Prozess, oder irgend etwas anderes – sich vorzustellen als ein Paar (**WAS**, **WIE**), in dem das **WAS** sich auf Unverzichtbares oder Wünscheswertes konzentrieren sollte.

Je kleiner nämlich das **WAS** gewählt wird – der Teil also, über den man sich mit anderen zu einigen hat – desto mehr Freiheit hat man als Entwickler, das **WIE** selbst zu bestimmen und ggfs. auch auszutauschen, *ohne* sich dazu mit jeder nur am **WAS** interessierten Partei auseinandersetzen zu müssen.

Für nur am **WAS** interessierte Parteien aber besteht der Vorteil darin, das **WIE** komplett ignorieren zu können (was für sie Komplexität ganz dramatisch reduziert).

Verstehen Sie jetzt, warum Abstraktion so extrem wichtig ist?

Es wird nun auch klar, warum, wer einen neuen Prozess über ein **WIE** zu definieren versucht (statt einfach nur über seinen Zweck),

- sich nicht nur unnötig Hindernisse aufbaut,
- sondern daneben auch die Chance vergibt, dass andere ein besseres **WIE** finden, sprich: einen besseren Weg hin zum gemeinsamen Ziel.

Grundsätzlich gilt: Wer darauf besteht, sein Ziel über ein **WIE** zu definieren, wo es reicht, über das **WAS** Einigkeit zu erzielen, handelt sich dadurch ernsthafte Nachteile ein und lenkt die gesamte Diskussion in eine wenig fruchtbare Richtung.

Die 17 Verfasser von Agile Manifesto machen genau diesen Fehler, und das mag der Grund dafür sein, dass die Suche nach einem für *alle* akzeptablen **WIE** Agiler Methodik auf breiter Front erst noch beginnen muss – obgleich über das **WAS** ja durchaus Einigkeit besteht und das Thema nun schon wenigstens 10 Jahre lang ständig neu besprochen wird. Eine Lösung zu finden, erscheint zunehmend als dringend, kann aber nicht gelingen, solange man das Manifesto nicht beiseite schiebt und die Anhänger klassischer Methodik sich deswegen an der Diskussion kaum beteiligen.

Man erkennt:

Wer ein **WIE** mit dem **WAS** verwechselt, blockiert Fortschritt (!).

DocHome = http://greiterweb.de/spw/dox/Abstraktion_als_Denkwerkzeug.pdf

Nochmals zurück zum als Computer betrachteten WWW:

Das WWW – und zunehmend auch MS Windows mit seiner autopmatischen Update-Funktion – sind vergleichbar mit riesigen Pflanzen: Ständig im Wachsen begriffen, aber auch ständig einem Erneuerungsprozess unterworfen. Je kleiner die einzelnen Teile (Äste, Blätter, Executables, Seiten im Web, oder gar ihr Inhalt), desto kürzere Lebensdauer haben sie.

Auch was Robustheit hinsichtlich Ausfall oder einem Abschneiden selbst größerer Teile betrifft, lässt sich das WWW gut mit einer Pflanze vergleichen. Es scheint nur noch eine Frage der Zeit zu sein, bis sich Applikationen ganz allgemein in diese Richtung entwickeln. Cloud Computing steht ja noch ganz am Anfang seiner Entwicklung.

Wer sich vor Augen führt, dass all das ***in nur 60 Jahren*** entstand, kann nicht anders als neugierig sein, wohin uns dieser Weg z.B. schon in den nächsten 60 Jahren noch führen könnte.