

# 1 Thesen

In diesem einführenden Kapitel präsentieren wir einen Überblick über unsere Thesen. Wir beschreiben grundsätzliche Strategien zur Erhöhung der Effizienz der Softwareentwicklung, und diskutieren bestehende Probleme und Lösungsansätze. Im letzten Teil dieses Kapitels führen wir zusammenfassend alle Thesen auf, die wir in diesem Buch definieren. Der Inhalt dieses Kapitels ist für das weitere Verständnis wichtig und sollte daher von beiden Zielgruppen, Managern und Entwicklern, gelesen werden.

## 1.1 Einführung in das Konzept

Unsere Thesen betreffen strategische und organisatorische Maßnahmen zur Verbesserung der Effizienz, Flexibilität und Qualität der Softwareentwicklung. Sie sind keine Wunschträume, sondern sie wurden bzw. werden bereits in der Praxis angewendet. Ergebnisse und Analysen zur erzielten Effizienz sind bereits verfügbar, u.a. wurde eine komplexe Anwendung für die internationale Raumstation ISS mit einem (automatischen) Softwareproduktionsprozess<sup>1</sup> erfolgreich entwickelt. Unsere Thesen definieren eine neue Strategie der Softwareentwicklung, die die vollständige Optimierung des Entwicklungsprozesses zum Ziel hat.

---

<sup>1</sup> Unter einem "Softwareproduktionsprozess" verstehen wir nicht nur einen Prozess, der Code generiert, sondern einen, der alle Entwicklungsphasen von der Spezifikation bis zur Abnahme einschließt, insbesondere berücksichtigt er Integration, Test, Verifikation, Validierung, sowie die (frühzeitige) Identifizierung von Anwenderfehlern. Ein automatischer Produktionsprozess erfordert die Mitwirkung eines Entwicklers nur bei der Spezifikation und der Abnahme, aber nicht während der Zwischenphasen. Eine charakteristische Eigenschaft eines solchen Prozesses ist, dass die Produktionszeit um ca.  $1/x$  sinkt, wenn die Rechnerleistung um den Faktor  $x$  steigt.

*These 23  
Eine geänderte  
Rollenverteilung  
zwischen Ent-  
wickler und  
Rechner kann  
die Effizienz  
wesentlich  
erhöhen.*

Ein wichtiger Aspekt ist die Forderung nach einer geänderten Aufgabenverteilung zwischen Softwareentwickler und Rechner als Hilfsmittel der Entwicklung. Bei der jetzigen Vorgehensweise werden vom Entwickler Tätigkeiten ausgeführt, die ein Rechner bei entsprechender Organisation viel schneller und zuverlässiger ausführen könnte, während ein Entwickler grundsätzliche Probleme hat – aufgrund der menschlichen Unzulänglichkeit, sie regelkonform umzusetzen.

Zur Zeit sehen wir zwei prinzipielle Konflikte:

- Entwickler möchten und müssen kreativ sein, während sie bei der Umsetzung ihrer Ideen in ein Softwareprodukt Regeln einhalten müssen. Kreativität und Regelkonformität widersprechen sich aber.

Die zu bewältigenden Aufgaben und Regeln sind üblicherweise so komplex, dass die Entwickler das gesamte Problem – Entwicklungsziel und Umsetzungsregeln – nicht mehr überblicken können, und Regelverstöße Teil der Problemlösung werden.

Um Regelverstöße zu finden, sind weitere Entwickler zur Qualitätssicherung, viel Aufwand und Zeit erforderlich, wohl wissend, dass die Qualitätssicherer prinzipiell nicht alle Regelverstöße finden können.

- Entwickler suchen Herausforderungen, sehen aber als die größte Herausforderung – leider – die Tätigkeit an, bei der sie prinzipiell nicht gewinnen können: die regelkonforme und korrekte Umsetzung ihrer Ideen durch sie selbst.

In der Annahme, die korrekte Umsetzung nur selbst durchführen zu können, konzentrieren sich die Verbesserungsmaßnahmen auf die Unterstützung der Entwicklungsarbeit durch Rechner, wobei die Last dennoch bei den Entwicklern bleibt. Der Weg zur Alternative, einem Rechner die regelkonforme Umsetzung zu überlassen, ist somit blockiert.

*These 26  
Eine klare Trennung der Aufgaben zwischen Entwickler und Rechner ist erforderlich*

Die Lösung dieser grundsätzlichen Konflikte erfordert daher eine klar definierte, neue Rollenverteilung zwischen Entwicklern und Rechnern entsprechend ihrer Fähigkeiten:

- kreative Aufgaben übernimmt der Entwickler,
- die Umsetzung nach festen Regeln der Rechner, wobei der Rechner einen Entwickler auch bei der Organisation seiner Ideen unterstützen kann.

Ein Rechner wird dann nicht mehr hauptsächlich zur Ausführung von Programmen eingesetzt, sondern für die Produktion der Programme selbst. Der Entwickler definiert das Ziel, und der Rechner transformiert seine Ideen in korrekte und ausführbare Programme. Durch seine großen Ressourcen verstärkt er dabei die Softwareent-

wicklungsfähigkeiten seines "Meisters" in unvorstellbarem Maße, ohne Fehler zu übernehmen oder zu erzeugen. Aber der "Meister" muss lernen, seinem "Besen" die richtigen Instruktionen zu geben, sonst bleibt der Erfolg aus, wie beim "Zauberlehrling" von Johann Wolfgang von Goethe.

Vereinfacht ausgedrückt setzt ein Entwickler bei der bisherigen Vorgehensweise seine Ideen selbst um, und der Rechner kontrolliert meistens nur, ob er die Regeln einhält, und dies auch nicht vollständig. Bereits realisiert wurde eine solche notwendige Unterstützung durch den Übergang von Assemblern zu Compilern. Compiler sind ein gutes Beispiel für die effiziente Nutzung der "Verstärkereigenschaften" eines Rechners. Aber die Probleme auf höheren Abstraktionsebenen werden damit nicht gelöst. Durch die Mächtigkeit der Compiler kann mehr ausführbarer Code als früher erzeugt werden, aber dadurch sind neue Probleme entstanden. Die große Menge an Quellcode muss auch auf Korrektheit überprüft werden, aber dazu reichen die Mittel eines Compilers nicht aus. Jetzt muss die Erzeugung von Quellcode für die Compiler organisiert und unterstützt werden.

*These 24  
Die "Verstärkereigenschaften" eines Rechners können effizient für die Softwareentwicklung eingesetzt werden.*

Bei der Umsetzung der Eingaben oder Vorgaben in ein Ergebnis werden nur zu einem geringen Teil die Ressourcen eines Rechners genutzt. Die Ergebnismenge beträgt üblicherweise ein Vielfaches der Eingabemenge, und der Entwickler selbst übernimmt diese Vervielfachung. Bei den meisten aktuell angewandten Verfahren zur Testautomation leitet beispielsweise der Entwickler die (zahlreichen) Testfälle ab und benutzt erst zur Ausführung einen Rechner.

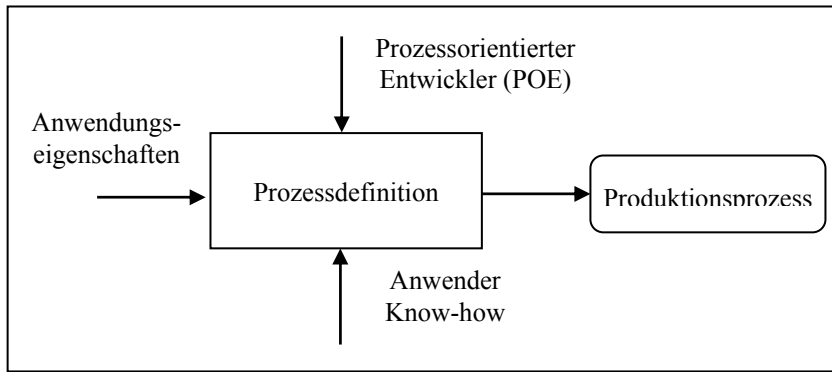
Die geänderte Rollenverteilung erfordert ein Umdenken und eine Organisation, die neue Abläufe unterstützt.

Aus unserer Sicht wird es dann die folgenden Entwicklerprofile geben (Abb. 1-1 und Abb. 1-2):

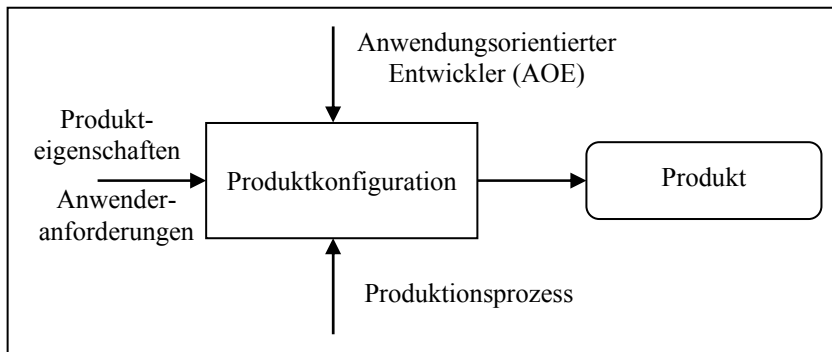
- den anwendungsorientierten Entwickler ("AOE"),
- den prozessorientierten Entwickler ("POE"), und
- den herkömmlichen Entwickler.

Alle drei Profile können natürlich auch in einer Person bzw. Firma vereint sein.

Der prozessorientierte Entwickler (POE) (Abb. 1-1) definiert den Softwareproduktionsprozess, wobei er auf Information über Produkteigenschaften und spezifisches Know-how des Anwenders angewiesen ist, um den Prozess optimal auf die Anforderungen abstimmen zu können.



**Abb. 1-1 Prozessdefinition<sup>2</sup>** Der anwendungsorientierte Entwickler (AOE) (Abb. 1-2) definiert – kreativ – die Anwendung, und setzt sie durch einen Produktionsprozess mit einem Rechner in ein reales Ergebnis um, ohne an der Produktion (Erstellung von Quellcode usw.) selbst beteiligt zu sein. Er konfiguriert den Produktionsprozess entsprechend den Anwenderanforderungen über die Produkteigenschaften.



**Abb. 1-2 Produkt-konfiguration** Der "herkömmliche" Entwickler entwickelt Software, die nicht durch Produktionsprozesse erzeugt wird. Nach einer Übergangszeit wird diese Tätigkeit an Bedeutung verlieren. Dies ist vergleichbar zum Übergang von Assembler auf Compiler. Der Anteil an Assemblerprogrammen ist heute sehr gering. Nur für sehr spezifische Probleme wird weiterhin die herkömmliche "manuelle" Entwicklung notwendig sein, ähnlich wie sich heutzutage die Anwendung manuell erzeugten Assemblercodes auf wenige einfache Teilaufgaben beschränkt, etwa beim Laden eines Betriebssystems.

<sup>2</sup> Wir benutzen hier die grafische Notation von SADT™ (Structured Analysis and Design Technique): von links: Eingabedaten, von oben: Kontrolltätigkeit, von unten: Methoden, nach rechts: Ergebnis

Die eigentliche Softwareentwicklung – nach bisherigem Verständnis – findet somit nur noch bei der Entwicklung der Produktionsprozesse statt. Die bisherige Erfahrung zeigt, dass der Anteil der erforderlichen Neuentwicklungen sinkt, wenn schon Produktionsprozesse vorhanden sind. Die Wiederverwendbarkeit der Produktionssoftware ist durch die strikte Organisation erheblich höher als bei der bisherigen Anwendungssoftware – dies lässt sich an den bisher von uns realisierten Prozessen nachweisen. Durch die angewandte Organisationsform wird die Software des Produktionsprozesses sogar 100% wiederverwendbar. Die Wiederverwendbarkeit impliziert auch eine Optimierung des Produktionsprozesses und der Wartung: die Entwicklung geht in ständige Wartung über.

Dieser hier skizzierte Ansatz kann natürlich nur schrittweise eingeführt werden d.h. es wird eine Koexistenz von "automatischer" und "manueller" Softwareentwicklung für eine Übergangsphase geben. Die Integration manuell neu erstellter bzw. bereits vorhandener Software kann und sollte ein Produktionsprozess ebenfalls unterstützen.

## **1.2 Mehr Chancen durch bessere Strategien**

Den Begriff "Strategie" interpretieren wir in zweifacher Weise:

1. Strategien zur systematischen Optimierung der Softwareentwicklung,
2. Strategien der Softwarenutzer zum Erreichen ihrer eigenen Ziele.

Mit besseren Strategien können die Softwareentwickler wettbewerbsfähiger werden und die Qualität ihrer Produkte erhöhen, aber auch flexibler und agiler werden, damit ihre Kunden durch Einsatz von guter Software ihre eigenen strategischen Ziele besser erreichen können. Bei kundenspezifischen Entwicklungen sollte der Anwender als Teil seiner Strategie sich mit den Softwareentwicklern abstimmen, um durch Synergie einen höheren Nutzen zu erhalten.

In zunehmenden Maße wird Software eingesetzt, ihr kommt immer größere Bedeutung zu. Sie kann den Nutzer begünstigen oder auch behindern. Als Behinderung verstehen wir hier Abweichungen zur versprochenen Leistung, und unvollständige Abdeckung von Anwenderanforderungen. Entsprechend der größeren Verbreitung treten auch immer öfter Behinderungen auf. Unser Ziel ist es, die Softwareentwicklungsmethodik so auszurichten, dass eine Behinderung von Anwendern nicht eintreten kann, d.h. dass verdeckte Män-



gel ausgeschlossen sind, und flexibel und zu akzeptablen Kosten die Bedürfnisse von Anwendern abgedeckt werden können.

Eine Analyse der gegenwärtigen Situation der Softwareentwicklung impliziert, über deren Probleme zu diskutieren. Diese Probleme sind sehr groß. Im täglichen Leben können wir das feststellen, wenn wir schwer lesbare Rechnungen bekommen, wenn ein Programm plötzlich abstürzt, oder wir über "Tricks" versuchen müssen, von einem Programm das zu bekommen, was wir wollen. Wenn Software ausfällt, kann dies zu großen Störungen führen.

Ende September 2004 fiel das Flugbuchungssystem bei der Deutschen Lufthansa aus. Flüge – auch von Partnergesellschaften – mussten gestrichen werden, hatten große Verspätung und Passagiere mussten umbuchen. Obwohl ein solches Ereignis glücklicherweise selten eintritt, ist der Schaden im Fall der Fälle groß.

Natürlich gibt es eine Menge von Programmen, mit denen der Anwender voll oder weitgehend zufrieden ist. Hohe Kundenzufriedenheit impliziert aber momentan hohe Entwicklungskosten und lange Entwicklungszeit. Für gute Qualität muss ein hoher Preis bezahlt werden. Anwender und Entwickler meinen zu wissen, dass sie bei niedrigem Preis ihre Ansprüche an Qualität erheblich reduzieren müssen, ein Anspruch auf Mängelbeseitigung ist in der Regel bei frei verkäuflicher Software nicht durchsetzbar. Möglicherweise wird der Mangel mit der nächsten Version behoben, aber dafür muss wieder bezahlt werden. Dies sind Prozeduren, die in anderen Bereichen nicht üblich sind.

*These 62  
Aufwand sen-  
ken, Qualität  
erhöhen*

Um ausreichende Qualität sicherzustellen, werden große Entwicklungsressourcen benötigt. Diese sind teuer und auch in der benötigten Menge tatsächlich nicht vorhanden. Seit einiger Zeit versucht man mit Outsourcing, Nearshoring oder Offshoring die hohen Personalkosten zu senken, ohne die eigentlichen Probleme – den hohe Bedarf an Entwicklern und ihre (relativ hohe) Fehlerrate – befriedigend lösen zu können.

Zur Skizzierung der aktuellen Situation und zur Erläuterung unseres prinzipiellen Lösungsansatzes werden wir auch Beispiele aus anderen Bereichen oder dem Alltag heranziehen, um dadurch besser die Probleme und ihre potenziellen Lösungen charakterisieren zu können. Uns ist bewusst, dass ein Vergleich möglicherweise nicht vollständig alle Aspekte berücksichtigen kann. Wir wählen auch das Mittel der "Karikatur", um – vielleicht überspitzt – die Situation und Lösungsansätze treffender darzustellen zu können.

## 1.2.1 Optimale Arbeitsteilung

Aus unserer Sicht werden zwar Rechner (Computer) vielfältig zur Ausführung von Anwendungsprogrammen benutzt, aber nur unzureichend zur Entwicklung von Programmen und zur Lösung der hierbei auftretenden Probleme hinsichtlich Ressourcen, Kosten und Zeit. Die Möglichkeiten von Computern werden u.E. nicht voll ausgeschöpft. Zur Optimierung reicht es nicht, die aus dem manuellen Entwicklungsprozess bekannten einzelnen Schritte durch Einsatz von Computern zu optimieren, sondern der gesamte Prozess muss neu betrachtet werden. Um optimieren zu können, benötigt man Kriterien über die Effizienz des Entwicklungsprozesses, und Metriken, durch die solche Kriterien messbar und vergleichbar werden. Daran mangelt es aber.

Der noch relativ junge Bereich "Softwareentwicklung" hatte anscheinend nicht genügend Zeit, einen langsamen und kontinuierlichen Reifeprozess zu durchlaufen. Nach einer kurzen "Anlaufphase" werden hohe Anforderungen an Qualität und Funktionalität gestellt. Andere Bereiche konnten auf jahrhundertlange Erfahrung zurückgreifen, um den Übergang von handwerklichen Abläufen in industrielle Prozesse innerhalb der letzten 150 Jahre zu bewältigen. Ein "Drill" wie bei der Ausbildung von Lehrlingen im Handwerk und bei Ingenieuren in der Ausbildung, strikte Anleitung zu Einfachheit, Zweckmäßigkeit und Robustheit ist in der Ausbildung von Softwareentwicklern weitgehend unbekannt.

Die Softwareentwicklung wird als kreative Tätigkeit verstanden, und Entwickler beanspruchen dafür genügend Freiraum. Solche Freiräume führen bei der Umsetzung von der Idee in ein ausführbares Programm zu Problemen, weil dann regelkonformes Vorgehen notwendig ist. Unser Ansatz zur Lösung dieses Konfliktes ist:

- der Entwickler *beschränkt* sich auf die kreativen Tätigkeiten,
- die Tätigkeiten, die nicht kreativ sind, sondern nach Regeln ausgeführt werden müssen, werden Automaten, den Computern, überlassen.

Damit gewinnt der Entwickler Freiräume für seine Kreativität, er wird von starren Tätigkeiten entlastet. Rechner dagegen können nur nach einem starren Plan ihre Arbeit ausführen, sind daher wesentlich zuverlässiger bei der Einhaltung der Regeln und schneller. Entwickler und Rechner ergänzen sich in dieser Weise ideal.

Bei der bisherigen Vorgehensweise ergeben sich Konflikte zwischen Kreativität und Vorschriften, weil die Entwickler ihre Aufga-

*These 2  
Der Stand der Technik versucht die Probleme nur durch punktuelle Automation zu lösen.*

*These 25  
Der kreative Entwickler soll nicht gezwungen werden, starre Regeln einzuhalten.*

*These 26  
Eine klare Trennung der Aufgaben zwischen Entwickler und Rechner ist erforderlich*

be hauptsächlich darin sehen, dort kreativ zu sein, wo die Einhaltung von Regeln angebracht wäre, ein inhärenter Konflikt, der sich historisch entwickelt hat. Aus unserer Sicht konzentrieren sich die bisherigen Entwicklungsmethoden und -werkzeuge darauf, den Ingenieuren zu helfen, mit diesem Konflikt leben zu können, die Auswirkungen zu mildern. Bei dieser Zielsetzung können sie ihn aber nicht grundsätzlich beseitigen, obwohl immense Ressourcen an Personal für Entwicklung und Qualitätssicherung eingesetzt werden.

*These 1  
Mit dem Stand  
der Technik  
können die prin-  
zipiellen Proble-  
me nicht gelöst  
werden.*

Der von uns einleitend geforderte Paradigmenwechsel besteht in einer strikten Aufgabenteilung entsprechend der Fähigkeiten, und einem gezielten Einsatz von Rechnern zur Umsetzung der kreativen Vorgaben unter Nutzung der großen Rechnerressourcen, die die Produktion großer und komplexer Softwaresysteme durch skalierbare und korrekte Produktionsabläufe ermöglichen. Wir fordern damit die Abkehr von einem Entwicklungsansatz, der die Lösung eines Konfliktes versucht, der prinzipiell nicht lösbar ist.

*These 27  
Die Arbeitstei-  
lung zwischen  
Mensch und  
Rechner muss  
gut organisiert  
werden.*

Diese Arbeitsteilung zwischen Mensch und Rechner erfordert aber eine gut abgestimmte Organisation, eine klare Abgrenzung zwischen kreativen und starren Tätigkeiten, eine exakte Definition, was ein Rechner ausführen soll, und klare Anweisungen. Aus den bisherigen Diskussionen wissen wir, dass der Übergang nicht einfach umzusetzen ist, auch weil alle bisherigen Anstrengungen sich nicht auf eine zufriedenstellende Lösung dieses Konfliktes konzentrierten, einschließlich Ausbildung. Wir werden den Leser daher langsam an diese Aufgabe heranzuführen.

Wir haben erlebt, dass Entwickler glaubten, die Idee sei so einfach, dass sie sie ohne spezielles Training verwirklichen könnten, dann scheiterten, und daraus schlossen, dass der Ansatz nicht realisierbar sei. Wenn jemand ein Flugzeug ohne Pilotenausbildung fliegt und dann abstürzt, wird (heute) niemand behaupten, dass Fliegen nicht möglich ist.

Wenn Alexander der Große sich die Idee zur Lösung des gordischen Knotens (s.a. Kap. 1.2.6) nur von einem anderen abgeschaut hätte, ohne selbst die notwendigen intellektuellen Fähigkeiten zu besitzen, hätte er zwar die Stricke trennen, aber nicht die damit verbundene Voraussage erfüllen könne, Asien zu erobern.

Der oben erwähnte Drill muss kein Widerspruch zur Kreativität sein. Mit reiner Kreativität kann man Bilder malen oder Musikstücke komponieren. Will man aber eine Idee in ein brauchbares Produkt umsetzen, so erfordert das Planung, und Planung impliziert geordnetes Vorgehen. Bei dieser Planung können Rechner aber die Entwickler unterstützen, und die Menge der anzuwendenden Regeln minimieren bzw. die Einhaltung der Regeln kontrollieren.



Neben der Kreativität wird also noch ein Minimum an geordneter Vorgehensweise verlangt. Damit die Tätigkeit effizient ausgeübt werden kann, ist auch hierfür Training oder eine Ausbildung erforderlich, so wie zum Fliegen.

Nach einer Beschreibung der aktuellen Situation, Identifikation der wichtigsten Anforderungen und einem (Rück-)Blick auf die bisher eingesetzten Entwicklungstechniken, bereiten wir dann den Leser allmählich auf die Methodik vor. Dies geschieht über eine qualitative und quantitative, *messbare* Bewertung der Situation, eine prinzipielle Beschreibung der Vorgehensweise und der Möglichkeiten, und schließlich durch Präsentation konkreter Beispiele aus verschiedenen Anwendungsbereichen.

Wir begannen mit Produktionsprozessen für komplexe technische Systeme, wie "embedded systems", Echtzeitsysteme, und haben dann die Erfahrungen abstrahiert und auf weitere Anwendungsbereiche übertragen: grafische Benutzeroberflächen, Datenbankanwendungen, logistische Systeme und Re-Engineering. Obwohl wir im Bereich "kritischer Systeme", und dort speziell im Bereich "Raumfahrtssysteme" anfangen, konnten wir die Methodik erfolgreich in anderen Anwendungsgebieten einsetzen, deren (funktionale) Anforderungen sich von den ursprünglichen stark unterscheiden. Die Erfahrung zeigt: je häufiger die Methodik angewendet wird, desto mehr potenzielle Anwendungsgebiete sieht man.

### 1.2.2

## Arbeitsplätze im Wandel

Die veränderte Rollenverteilung zwischen Softwareentwicklern und Rechnern wird zu einer strukturellen Änderung auf dem Arbeitsmarkt führen. Sowohl die Art der Tätigkeit als auch die Anzahl der benötigten Entwickler werden davon betroffen sein.

Die zukünftige Tätigkeit erfordert eine andere und höhere Qualifizierung, an die sich die Ausbildung anpassen muss. Für eine Übergangsfrist kann daher der Bedarf nur durch "Training-on-the-job" und begleitende Weiterbildungsmaßnahmen gedeckt werden.

Entwickler, die die nötige Qualifikation besitzen, werden ihren Arbeitsplatz sichern, weil sie eine höhere Produktivität erreichen sowie Produkte bei besserer Qualität mit mehr Funktionalität erzeugen können.

Firmen können ihre Kosten senken und ihre Wettbewerbssituation allgemein verbessern und gewinnen mehr Flexibilität, auf die Bedürfnisse des Marktes zu reagieren.



Dieser Strukturwandel wird zu einem Abbau an herkömmlichen Arbeitsplätzen führen, in der gleichen Weise, wie er in anderen Bereichen bereits bei fortschreitender Automation beobachtet werden konnte. Er bietet aber auch Chancen, wie aus diesen Bereichen ebenfalls bekannt. Denn die verbleibenden Arbeitsplätze werden stabilisiert, und neue können entstehen, weil neue Bedürfnisse entstehen, die erst durch die geänderten Entwicklungsstrukturen gedeckt werden können.

Erfolgt dieser Strukturwandel nicht, sind alle Arbeitsplätze gefährdet, während neue nicht entstehen. Diese Situation können wir bereits jetzt (2004) beobachten. Zur Kostensenkung werden Arbeitsplätze massiv abgebaut und in sog. "Niedriglohnländer" verlagert. Da alle Firmen auf diese Weise ihre Kosten senken werden, wird ein kontinuierlicher Druck entstehen, weitere Arbeitsplätze zu verschieben.

Natürlich können auch die "Niedriglohnländer" ihre Entwicklungsstrategie optimieren. Da bei dem von uns beschriebenen Ansatz die Produktionskosten praktisch unabhängig vom Lohn sind (Personalkosten fallen nur für Spezifikation und Abnahme an), wird sich der Wettbewerb von den Kosten auf die Produkteigenschaften verlagern. Somit entfällt der Druck zur Verlagerung von Arbeitsplätzen wegen zu hoher Lohnkosten. Kurz- bis mittelfristig wird der den größten Vorteil haben, der früh die neue Technologie einsetzt.

Detailliert werden wir auf die gesellschaftspolitischen Aspekte in Kap. 8 eingehen.

### **1.2.3**

#### **Synergie muss organisiert werden**

Software entzieht sich unseren Sinnen, das Entstehen eines Softwareproduktes ist schwer zu überwachen, sein Güte schwer zu beurteilen. Eine schiefe Fahrzeugkarosserie findet keine Käufer, jeder Entwickler und jeder Käufer sieht den Mangel sofort. Daher sind Sofortmaßnahmen unumgänglich, um die Produktion zu verbessern. Bei einem Softwareprodukt dauert dies länger. Da die Mängel nicht für jedermann sofort sichtbar sind, fehlt offensichtlich die unmittelbare Motivation zu ihrer sofortigen Behebung.

Die Produktion von Quellcode wurde bzw. wird als Hauptziel oder sogar als alleiniges Ziel der Softwareentwicklung angesehen. Wichtig ist anscheinend nur, dass ein Programm ausgeführt werden kann, ob dies zuverlässig geschieht und es alle Anforderungen des Anwenders erfüllt, scheint zweitrangig zu sein. Test, Verifikation

und Validierung der Eigenschaften wurden lange bzw. werden immer noch vernachlässigt.

Während in der digitalen Technik jeder Eingang oder Ausgang einer elektronischen Bauelementes nur zwei (operationelle) Zustände hat, hat der Parameter einer Funktion praktisch ein Kontinuum von Zuständen. Die Anzahl der Testfälle explodiert daher sehr schnell, der Nachweis der Funktionsfähigkeit ist somit sehr schwer und zeitraubend.

Umso mehr werden daher geeignete Strategien benötigt, die bei der Lösung dieser immensen Probleme helfen. Wir werden zeigen, dass Qualität und niedrige Kosten kein Widerspruch sein müssen. Im Gegenteil, wir identifizieren eine Synergie zwischen Produktion von Code und dem Nachweis seiner Qualität, d.h. wir leiten aus dem Produktionsprozess für den Code Information ab, die den effizienten Nachweis seiner (geforderten) Eigenschaften in einer für den Entwickler bzw. Qualitätssicherer verständlichen Form erlaubt.

Diese Korrelation ist nicht zwingend, wie die Vergangenheit zeigt. Wenn der Produktionsprozess für den Code nicht geeignet organisiert ist, dann kann auch die für die Synergie notwendige Information nicht abgeleitet werden. Diese Synergie war auch zuerst für uns nicht offensichtlich, sondern ihre Identifikation stand am Ende einer Reihe von Verbesserungsmaßnahmen, einer Analyse der ausgeführten Maßnahmen und Ergebnisse, und einer daraus resultierenden Abstraktion der Vorgehensweise. Die Erfahrung zeigt auch, dass eine solche Optimierung kaum theoretisch vorhergesagt werden kann, sondern nur durch Ausführung von kleinen Schritten, im Wechsel von Theorie und Praxis, erreicht werden kann. Nachdem wir aber diese Erkenntnis gewonnen hatten, konnten wir abstrahieren und sie gezielt auch bei anderen Entwicklungen von Produktionsprozessen einsetzen.

Das Finden einer guten Strategie ist sicher nicht einfach. Aber eine übergeordnete Lösungsstrategie hilft hierbei weiter: man muss auf Abstand zum Problem gehen, es hilft nicht, immer tiefer in das Problem einzudringen. Scheint das Problem immer komplexer zu werden, oder ist keine Lösung in Sicht, dann ist der Zeitpunkt gekommen, das Problem neu zu überdenken.

Wir haben den Eindruck – und diese Meinung wird von anderen uns bekannten Fachleuten geteilt: wenn Firmen bzw. deren Mitarbeiter Probleme haben, haben sie wegen der vorhandenen Probleme keine Zeit, diese Probleme zu lösen. Wichtig ist zu erkennen, dass man in einem solchen Teufelskreis gefangen ist, aus ihm herauszutreten und sich für eine mögliche Lösung zu öffnen.

Mitte der 80er Jahre haben wir mit der Vereinheitlichung von Systemstrukturen begonnen, zunächst mit der Zielsetzung, die Wieder-

*These 45  
Synergie zwischen Code-  
generierung und  
Qualitätssicherung  
ist möglich.*

*These 28  
Synergie muss  
organisiert werden.*

verwendbarkeit von Software zu erhöhen. Anfang der 90er Jahre kamen Aspekte zur Steigerung der Qualität und Zuverlässigkeit sowie Reduktion des Entwicklungsrisikos hinzu.

Ende der 90er Jahre erkannten wir, dass sich beide Aspekte unter dem Dach der Automation zusammenführen ließen, und begannen systematisch den Grad der Automation in unserer Softwareentwicklung zu erhöhen. Anfang des neuen Millenniums konnten wir vollautomatische Softwareproduktionsprozesse<sup>3</sup> realisieren und damit beginnen, weitere Bereiche wie grafische Benutzeroberflächen, Datenbankanwendungen oder "programmierbare Steuerungen" zu erschließen.

Die Portierung des prinzipiellen Ansatzes auf verschiedene Anwendungsgebiete ist möglich, wenn auch – je nach Ähnlichkeitsgrad – Anpassungen notwendig sind bzw. neue Software erstellt werden muss. Die Generierung eines Echtzeitsystems erfordert andere "Automaten" als die eines Datenbanksystems oder einer grafischen Benutzeroberfläche. Trotz dieser teilweisen Neuimplementierung bleiben die Vorteile erhalten, da viele Anwendungen des jeweiligen Bereiches abgedeckt werden können.

Die Vorgehensweise, Produktionsprozesse für Spezialgebiete einzurichten, ist nicht neu. Der Fertigungsprozess für ein Auto unterscheidet sich von dem für einen Fernsehapparat. Aber gerade durch diese Spezialisierung werden Kosten gespart und die Qualität erhöht.

Auch zeigt sich, dass sich Qualität und Effizienz überhaupt erst durch Spezialisierung messen lassen. Man versuche einmal, die Qualitätsprüfungsmaßnahmen für einen Fernseher auf ein Auto zu übertragen, oder die Produktivität eines PKW-Werks und eines Fernseher-Herstellers allein durch Vergleich der Zahl produzierter Einheiten pro Tag in Relation zu setzen.

Genauso kann die Qualität eines Datenbankentwurfs oder einer grafischen Oberfläche nicht mit denselben Metriken festgestellt werden wie die eines Echtzeitsystems. Optimierung kann also nur durch Spezialisierung in Fakten und Zahlen sichtbar werden.

---

<sup>3</sup> Unter einem "vollautomatischen Produktionsprozess" verstehen wir einen Softwareentwicklungsansatz, bei dem ein Ingenieur nur noch am Anfang die Anforderungen definiert und am Ende den vollständigen Code abholen kann einschließlich des Nachweises der Korrektheit, ohne dass dazwischen die Mitwirkung des Ingenieurs erforderlich ist.

## 1.2.4

### Offen sein für Problemlösungen

Unser Eindruck ist, dass Probleme, die im Bereich der Softwareentwicklung noch gelöst werden müssen oder gar als unlösbar angesehen werden, wie ausreichende Zuverlässigkeit, auch in anderen Bereichen auftreten und dort bereits zufriedenstellend gelöst wurden. Provokativ ausgedrückt sehen wir eher Ansätze zur Pflege der Probleme als zu ihrer Lösung. Dies mag damit zusammenhängen, dass die Softwareentwicklung und die Informatik relativ junge Domänen sind – nicht älter als ca. 70 Jahre, wenn man die ersten Schritte beispielsweise von Zuse hinzurechnet, und ihre Wurzeln mehr im mathematisch-wissenschaftlichen als im handwerklich-technischen Bereich liegen.

Für Laien ist es praktisch unmöglich, Einblick in die Praktiken der Softwareentwicklung zu bekommen. Durch den Einsatz von Programmiersprachen und komplexen Methoden und Werkzeugen können Außenstehende bzw. Anwender nur schwer verstehen, worin die Probleme liegen, ob sie besser gelöst werden könnten, und dann auf einer optimierten Lösung bestehen.

Programmiersprachen und -methoden wirken quasi als Zugangsbeschränkung zu Informationsquellen für Laien wie Latein oder Griechisch bei den Mönchen im Mittelalter: Dadurch wird jegliche Möglichkeit der Einflussnahme und Kritik von außen unterbunden. Dies führt zu einer Isolation und zu einer Verselbständigung der Vorgehensweise, der Konzentration auf Details, Verlust der Fähigkeit zur globalen Optimierung. Der hilfreiche Druck von außen, der Veränderungen anstoßen könnte, kann nicht entstehen.

Besteht eine Neigung, Probleme Laien oder Fachfremden, auch Informatikern oder Ingenieuren eines anderen Fachgebietes, nicht erklären zu können oder zu wollen, so erhöht sich die Wahrscheinlichkeit stark, dass eine Problemlösung komplexer als nötig wird.

Möglicherweise besteht ein Interesse, kontinuierlich unter gewohnten Bedingungen arbeiten zu können. Wird für eine Problemlösung sehr viel Zeit benötigt, oder ist eine Entwicklung sehr aufwändig, dann kann man sich lange damit beschäftigen, hat eine Aufgabe und Arbeit. Wird das Problem dann nach langer Zeit endlich gelöst, so spricht dies scheinbar auch für Kompetenz. Wenn Fremden Einblicke verwehrt werden, wird nicht sichtbar, dass tatsächlich Inkompetenz vorliegt, dass effizienter vorgegangen werden könnte.

*These 86  
Problemlösung  
setzt den Willen  
zur Diskussion  
und Offenheit  
voraus.*

*These 88  
Erfolgreiche  
Umsetzung von  
Veränderungen  
erfordert  
psychologisches  
Geschick.*

Will man einen Entwicklungsprozess optimieren, so muss man auch die Psychologie der Mitarbeiter berücksichtigen, wenn man Erfolg haben will. Aufgabe des Managements ist es, für einen Dialog mit Entwicklern die Voraussetzungen zu schaffen, über diesen Dialog Schwachpunkte zu identifizieren und dann die notwendigen strategischen Maßnahmen einzuleiten.

### **1.2.5 Helfen, nicht beschränken**

Softwareentwicklung ist prinzipiell immer eine Dienstleistung, auch wenn das Ergebnis später selbst ein Produkt oder in einem Produkt enthalten ist. Immer wird Software von "Anwendern" benutzt, direkt von Menschen oder von anderen technischen Systemen.

Ziel der Entwicklung sollte immer sein, den Bedarf eines Anwenders ("Kunden") zu decken. Dazu ist die Software ein Hilfsmittel, aber sie darf nicht Selbstzweck sein. Nicht sie steht im Mittelpunkt, sondern der Bedarf des Anwenders. Aus unserer Sicht wird aber zu oft versucht, den Bedarf des Anwenders in den Käfig der Softwareentwicklung zu zwängen, obwohl dies nicht nötig wäre, wenn die Entwicklungsansätze flexibel genug wären. Oft wird (vordergründig) argumentiert, es sei schwierig, auf die realen Bedürfnisse wegen der komplexen Zusammenhänge und hohen Kosten einzugehen, während zielgerichtete Arbeit zur Beseitigung dieser Ursachen unterbleibt.

So bekommt der Anwender nur zu oft, was als realisierbar angesehen wird, aber nicht, was er tatsächlich braucht. Jeder kennt aus dem täglichen Leben die Probleme, die – aus unserer Sicht – durch unsachgemäße oder zu komplexe Entwicklung verursacht werden, und zu mangelnder Flexibilität bei der Deckung der Kundenbedürfnisse führen.

Nutzt beispielsweise ein Betrieb ein Softwaresystem für die Verwaltung seiner betrieblichen Daten, so wird die Abhängigkeit von einem solchen System sehr groß. Unter den gegenwärtigen Bedingungen ist es dann kaum möglich, das System zu wechseln oder die interne Organisation des Betriebes an aktuelle Bedürfnisse anzupassen. Kürzlich musste ein größerer Betrieb seine gesamte Belegschaft (mehrere tausend Mitarbeiter) für mehr als einen Monat in Urlaub schicken, weil die Organisationssoftware umgestellt wurde.

Während Unternehmensberater den Firmen raten, ihre Organisationsstruktur kontinuierlich zu optimieren, beispielsweise auf prozessorientierte Organisationsformen umzustellen, um damit flexibler reagieren zu können, klagen Firmen, dass eine solche Änderung

nicht oder nur unter hohen Kosten und Beeinträchtigung des Betriebes möglich ist. Wenn ein Betrieb gerade unter hohen Kosten ein Softwaresystem für die Erfassung betrieblicher Daten basierend auf einer Matrixorganisation eingeführt hat, kann er nur unter großen Schwierigkeiten auf eine prozessorientierte Form umstellen. Neben der Bindung von Personal wäre auch der betriebliche Ablauf beeinträchtigt.

Die Einführung von Software für die Erfassung und Verwaltung von Betriebsdaten hat somit positive und negative Seiten. Dem besseren Überblick auf betriebliche Abläufe und dem schnellen Zugriff auf Daten steht (möglicherweise) eine eingefrorene Organisationsstruktur und daraus resultierend geringere Flexibilität gegenüber.

### 1.2.6 Probleme richtig verstehen

Einst löste Alexander der Große den "Gordischen Knoten" – unkonventionell – mit einem Schlag seines Schwertes. An den kunstvoll geknoteten Stricken, die den Streitwagen des Königs Gordios von Phrygien untrennbar mit dem Zugjoch verbinden sollten, waren zuvor die Gelehrten gescheitert, weil sie versuchten, ihn ohne Beschädigung zu entfernen. Wer den Knoten von dem Zeus geweihten Wagen lösen könne, würde Asien erobern, hatte das Orakel vorhergesagt. Alexander sollte später mit seinem erfolgreichen Asienfeldzug in die Geschichte eingehen, weil er auch für die Kriegsführung unkonventionelle Methoden einsetzte.

Dies zeigt uns, dass und wie ein Problem komplexer, und damit sogar unlösbar werden kann, wenn man nicht fähig ist, es unvoreingenommen zu betrachten. In diesem Fall haben die Gelehrten, die das Problem zu lösen suchten, nur einen Unterraum des gesamten Lösungsraumes betrachtet und daher die Lösung nicht gefunden. Die mögliche Lösung lag außerhalb ihrer Vorstellungskraft.

Während sich die Experten (Gelehrten) darauf konzentrierten, eine Trennung ohne Beschädigung der Stricke durchzuführen, fand Alexander der Große eine pragmatische und einfach realisierbare Lösung: er trennte die verknoteten Stricke mit einem Schwert und fand damit eine unerwartete, aber zulässige Lösung. Nirgends war gefordert, dass die Stricke bei der Lösung des Knotens unbeschädigt bleiben müssen.

Die vom Orakel hergestellte Verknüpfung zwischen Lösung der Aufgabe und der Herrschaft über Asien trat dann auch ein. Dies dürfte weniger auf göttliche Fügung als eher auf die zielgerichtete, pragmatische Vorgehensweise von Alexander dem Großen zurück-

*These 92  
Das richtige  
Problemver-  
ständnis führt  
zum Erfolg.*

zuführen sein. Er hat das Problem gelöst, indem er sich auf die eigentliche Aufgabenstellung konzentrierte. Während andere in ihrer üblichen "wissenschaftlichen" Denkweise gefangen waren ("den Knoten zu entwirren"), löste er sich davon. Er erkannte, dass eine Lösung des Knotens nicht unbedingt eine Umkehrung des früheren Vorganges der Verknötung erfordert.

*These 68  
Unkonventionel-  
le Lösungsan-  
sätze sichern  
den Erfolg.*

Wer das Potenzial besitzt, scheinbar komplexe Probleme so stark auf ihren eigentlichen Sachverhalt zu reduzieren, dass sie lösbar werden, besitzt auch das Potenzial, sich im Alltag oder im geschäftlichen bzw. politischen Umfeld durchzusetzen. Seine Fähigkeit, durch die Alexander den Gordischen Knoten lösen konnte, prädestinierte ihn auch, seine weiteren Siege zu erringen, weil er anscheinend gewohnt war, unkonventionelle Mittel zur Verwirklichung seiner Ziele einzusetzen. Wer erkennt, dass vorgegebene Wege verlassen werden müssen, um ein Ziel zu erreichen, ist seinen Konkurrenten immer mindestens einen Schritt voraus.

*These 69  
Einfachheit, nicht  
Komplexität  
impliziert den  
Erfolg.*

Das Schlüsselwort dazu heißt "Simplifizieren": Simplifizieren, um Erfolg zu haben, statt Komplexität pflegen, um Erfolg zu suchen – ihn aber nicht zu bekommen. Alexander der Große hat einen einfachen und unkonventionellen Lösungsansatz gewählt, und war erfolgreich.

## 1.2.7 Simplifizieren durch Organisieren

*These 29  
Einfachheit kann  
organisiert wer-  
den.*

Organisation hilft, scheinbar verwirrende Sachverhalte so zu vereinfachen, dass sie lösbar werden. Bevor wir an die Lösung eines Problems gehen, sollten wir die Komplexität der Aufgabe auf ein Minimum reduzieren, quasi auf eine "minimale Normalform" bringen.

Alexander der Große hat dies getan, indem er die für die Problemlösung nicht geforderte, von den Gelehrten irrtümlich hinzugefügte Nebenbedingung auf zerstörungsfreie Trennung der Stricke fallen ließ. Die Gelehrten waren Gefangene ihrer Ausbildung und ihrer Denkweise, und somit unfähig, das Problem zu lösen. Möglicherweise hatten sie auch zu viel Respekt vor dem Götterwagen, und kamen daher nicht auf die einfache Lösung.

Vereinfachen durch Organisieren bedeutet, nach einer Analyse Denkblockaden zu beseitigen und die Randbedingungen so zu gestalten, dass das Problem lösbar wird.

Um zu zeigen, wie man durch Organisation ein Problem vereinfachen kann, nehmen wir als Beispiel die Verifikation von Software, für die heute sehr viel Zeit und Aufwand benötigt wird. Durch den großen Bedarf an Ressourcen für die Codierung wird die Verifikati-



on vernachlässigt, unterbleibt ganz oder teilweise. Dagegen kann bei geeigneter Organisation eine Verifikation (in bestimmten Fällen) komplett entfallen. Um den Lösungsansatz verständlich beschreiben zu können, beginnen wir mit einem Verifikationsproblem aus unserem Alltag.

Wenn wir an einer Wand ein Bild aufhängen wollen, müssen wir – sinnvollerweise – verifizieren, dass sich dort, wo wir einen Nagel einschlagen oder ein Loch bohren wollen, keine elektrische Leitung befindet.

Wenn die Elektroinstallateure unorganisiert vorgegangen wären, müssten wir damit rechnen, dass an jeder Stelle der Wand eine Leitung unter dem Putz verlaufen könnte. Dann brauchten wir ein Werkzeug, das uns anzeigt, ob an der beabsichtigten Stelle tatsächlich eine Leitung verläuft oder wir müssten den Putz entfernen. Zur Verifikation unserer Hypothese "keine Leitung an der ausgewählten Stelle" wäre somit erheblicher Aufwand notwendig – relativ zur eigentlichen Aufgabe: wir müssten uns ein Werkzeug besorgen und damit die Wand untersuchen.

In der überwiegenden Zahl der Fälle brauchen wir aber ein solches Werkzeug überhaupt nicht. Die Handwerker haben nämlich gemäß "best practice" gewisse Regeln eingehalten, aus denen wir schließen können, dass im zentralen Teil der Wand keine Leitungen verlaufen<sup>4</sup>. Wir können daher auf eine Verifikation vollständig verzichten.

Übertragen auf die Softwareentwicklung heißt dies: bei guter Organisation können komplexe Aufgaben entfallen. Wenn bestimmte Regeln eingehalten werden, entstehen bestimmte Probleme überhaupt nicht. Voraussetzung ist, dass man sich von vorgegebenen Denkweisen löst, die einengen und eine Lösung erschweren oder verhindern.

Ein Entwickler sollte Regeln definieren, die der Rechner ausführt, aber keine Regeln auf der Ebene der Softwareimplementierung für sich oder Kollegen. Davon nicht betroffen sind Regeln, die der Festlegung der Aufgabe dienen, aber die Kreativität nicht beeinflussen. Solche Regeln – projiziert auf das obige Beispiel – könnten sein: für das Aufhängen des Bildes muss bestimmt werden, welches Bild und welcher Nagel oder welche Schraube und welcher Dübel benötigt werden.

Die Einführung solcher Regeln darf aber nicht die Anwendung beeinträchtigen. Statt der Regel "Leitung parallel zum Rand in ei-

<i>These 30 Gute Organisation verringert Komplexität, Aufwand und Entwicklungs- zeit.</i>
---

---

<sup>4</sup> Auf die Betrachtung von Sonderfällen wie einer Zuleitung für eine Wandlampe, die im Innenbereich der Wand liegen könnte, wollen wir nicht eingehen, da dafür ähnliche Schlussfolgerungen möglich sind.

nem bestimmten Abstand" hätte man auch die Regel einführen können "Stromanschlüsse gibt es nur in einem Raum, wo keine Löcher in die Wand gebohrt werden müssen/dürfen". Auch dieser Ansatz löst das Problem, jedoch nur vordergründig aus Sicht der Sicherheitsingenieure. Der Bewohner hat dagegen erhebliche Nachteile. Er muss sich mit frei verlegbaren Kabeln den Strom in die Wohnräume holen – in denen zum Aufhängen von dekorativen Gegenständen oder der Einrichtung Löcher in die Wand gebohrt werden müssen und die täglich beleuchtet werden müssen – und dafür erheblichen und wiederkehrenden Aufwand betreiben.

Aus Anwendersicht ist auch das Gesundheitsrisiko nicht kleiner geworden. Dem – relativ seltenen – Risiko eines Stromschlages beim Aufhängen eines Bildes – ein Bild hängt man nicht jeden Tag auf – steht nun das – relativ häufige – Risiko gegenüber, stündlich über die Kabel zu stolpern und sich dabei Knochen oder sogar den Hals zu brechen. Bei dieser Lösung würden somit die für Sicherheit der Elektroinstallation zuständigen Ingenieure zwar die potenzielle Gefährdung durch Stromschlag verhindern, ein Bewohner hätte aber insgesamt ein viel größeres gesundheitliches Risiko.

Glücklicherweise gibt es erfahrene (und vernünftige) Elektroingenieure, die die erste und nicht die zweite Lösung gewählt haben. Leider kann man im Bereich der Softwareentwicklung nicht immer davon ausgehen, dass der zweite Lösungstyp vermieden wird, wie wir aus der Analyse von Softwareimplementierungen und Qualitätssicherungsmaßnahmen wissen. Wir werden später in Kapitel 3 einige Beispiele schildern.

Ein alternative Lösung wäre auch, die Wand nicht zu verputzen, denn dann wäre der Verlauf der Leitungen immer zu erkennen. Das wäre nicht im Sinne des Bewohners, aber gut für seine Sicherheit. Analog dazu wird in der Softwareentwicklung bei hohen Sicherheitsanforderungen "Sichtbarkeit" verlangt.

Da es unmöglich zu sein scheint, Sicherheit mit Einkapselung der Funktionalität ("Verputzen der Wand") zu vereinbaren, erhält die Sicherheit und damit "Sichtbarkeit" den Vorrang. Die Sichtbarkeit hat aber nicht nur Vorteile. Übertragen wir hierzu die in der Software angewandten Methoden auf unser praktisches Beispiel, so impliziert das, dass wir nicht selbst entscheiden können, ob wir risikolos einen Nagel einschlagen können, sondern wir müssen dazu unabhängige Sicherheitsfachleute hinzuziehen, die dann nach Begutachtung der unverputzten Wand "grünes Licht" geben. Hieraus wird deutlich, welche Folgen schlechte Organisation haben kann: nicht nur Einschränkung der Brauchbarkeit, sondern auch hohe Personalkosten und zeitliche Verzögerungen.

Häufig wird eine Entscheidung "pro Sicherheit" und "contra Komfort" getroffen, obwohl auch – siehe oben – eine Lösung möglich wäre, die keinen Konflikt zwischen beiden Zielen entstehen lässt – durch geeignete Organisation. Wie wir gesehen haben, erfordert eine solche Lösung den unbedingten Willen zur Simplifizierung.

Wir wollen dieses Beispiel auch noch in einer anderen Richtung verwenden, um zu erklären, dass ein bewährter Ansatz nicht unbedingt in allen Fällen zum Erfolg führt.

Betrachten wir hierzu einen Nassraum und die Entscheidung, aus Sicherheitsgründen dort keine Elektroinstallation zu verlegen. Dies ist durchaus eine praktikable Lösung für diesen speziellen Fall. Überträgt man nun diese erfolgreiche Lösung auf alle Räume eines Gebäudes, so ist die Gesamtlösung nicht akzeptabel, wie oben beschrieben. Folglich sollte man immer in jedem Einzelfall prüfen, ob eine unter bestimmten Umständen erfolgreiche Lösung auch immer bei der Lösung des nächsten, scheinbar ähnlichen Problems hilft.

Aus diesen Betrachtungen folgt: gute Organisation ist notwendig, um ein Problem auch für den Anwender zufriedenstellend zu lösen, und nicht nur für den Entwickler.

Kehren wir nun zum Problem der Verifikation in der Softwareentwicklung zurück.

Bei dem überwiegend eingesetzten Phasenmodell (Spezifikation, Entwurf, Codierung, Test, Integration, Abnahme, s.a. Kap. 4) muss verifiziert werden, dass die Ergebnisse einer folgenden Phase mit den Vorgaben einer früheren Phase übereinstimmen. Das erfordert sehr viel Zeit und Aufwand, weil diese Tätigkeiten "manuell"<sup>5</sup> ausgeführt werden, und damit eine Abweichung von den Vorgaben nicht ausgeschlossen werden kann.

Setzt man dagegen Software- bzw. Systemgeneratoren ein, die beispielsweise eine Spezifikation regelkonform direkt in Code überführen (s.a. Kap. 6 und 7), braucht man nicht zu verifizieren, dass der Entwurf der Spezifikation entspricht, und der Code dem Entwurf. Die Übereinstimmung zwischen Code und Spezifikation ist inhärent durch das angewendete Verfahren gegeben, vorausgesetzt

*These 31  
Bei der Organisation der Lösung muss der Nutzen für den Anwender im Vordergrund stehen.*

---

<sup>5</sup> In diesem Buch verwenden wir den Begriff "manuell" für die durch den Einsatz von Personen (Softwareingenieuren) geprägte Art der Entwicklung, auch wenn diese hauptsächlich auf geistiger Leistung beruht. Unter "(voll-)automatischer" Entwicklung verstehen wir die Erzeugung von Software durch einen Produktionsprozess, der ohne manuellen Eingriffe aus einer (manuell erstellten) Spezifikation korrekte Software erzeugt und den Nachweis der Korrektheit selbst erbringt.

seine Korrektheit wurde vorher bewiesen. Wegen der Reproduzierbarkeit der Abläufe muss "nur"<sup>6</sup> einmal verifiziert werden, dass die Generatoren korrekt arbeiten. Dann kann man für jede weitere Anwendung von der Korrektheit des generierten Codes ausgehen.

In unserem Beispiel der Elektroinstallation war dies die Vorschrift, dass Leitungen parallel zu den Begrenzungen der Wand und nur in einem bestimmten Abstand vom Rand verlaufen dürfen. Bei der Erzeugung von Code sind es die Konstruktionsregeln, die sicherstellen, dass eine Spezifikation korrekt in Code umgesetzt wird, so dass eine Verifikation nicht notwendig ist.

Die zielgerichtete Organisation des Produktionsprozesses führt zu einer starken Vereinfachung und zur Verringerung von Aufwand und Zeit.

Wir wollen mit einem weiteren Beispiel schließen, das die Testorganisation betrifft. Wie wir in Kap. 7 zeigen werden, können aus einer vollständigen Spezifikation für ein Echtzeitsystem automatisch Testfälle für das Systemverhalten abgeleitet werden. Zum Nachweis seiner Eigenschaften wird das System mit den erwarteten Eingabedaten stimuliert, seine Reaktion beobachtet, die Ergebnisse aufgezeichnet und ausgewertet – alles automatisch.

Üblich ist zur Zeit, die Testfälle zu definieren, für jeden Testfall das System in den jeweiligen Anfangszustand zu bringen, den Test auszuführen und die Ergebnisse zu protokollieren – alles manuell.

Nachdem wir die Möglichkeiten der Testautomation aufgezeigt hatten, wurden wir verwundert gefragt, wie es denn möglich wäre, den immensen manuellen Testaufwand einzusparen, das sei doch höchst fraglich.

Die Erklärung ist jedoch recht einfach. Die Wurzeln sind in der geänderten Spezifikation und der daraus resultierenden Organisation der Produktion zu finden. Bei der manuellen Vorgehensweise müssen die Tests einzeln identifiziert und ausgeführt werden, da kein Automat diese Aufgabe übernehmen kann. Die Ausführung von Einzeltests impliziert jedoch, dass das System manuell in den jeweiligen Betriebszustand gebracht werden muss, was sehr aufwändig ist. Jeder Einzeltest repräsentiert einen Teil des gesamten operatio-

---

<sup>6</sup> "nur" weist auf den relativ geringen, *einmaligen* Aufwand für diese Aktivität hin, während bei herkömmlicher Verfahrensweise der Aufwand ständig anfällt. Gering ist dieser Aufwand aber auch nur dann, wenn wieder geschickt organisiert wird. Um zu garantieren, dass für beliebige Anwendungen des gewählten Bereichs das Ergebnis immer korrekt ist ("einmal korrekt, immer korrekt"), ist eine einmalige, geordnete Vorgehensweise erforderlich.

nellen Szenarios des Systems. Die Summe aller Tests deckt dann alle Betriebszustände ab.

Beim automatischen Testen wird das System wie im normalen Betrieb stimuliert. Somit wird – wenn alle möglichen Testfälle ausgeführt worden sind – auch das gesamte operationelle Szenario durchlaufen, mit dem Unterschied, dass die jeweiligen Voraussetzungen für den Test vom System automatisch eingestellt werden. Ein Test wird für die jeweilige Situation ausgewählt, daher muss das System nicht mehr in diesen Zustand gebracht werden.

Der wesentliche Unterschied zwischen beiden Vorgehensweisen besteht also darin, dass im ersten Fall ein Test willkürlich, d.h. unabhängig vom aktuellen Betriebszustand, ausgewählt wird, mit der Folge, dass die Voraussetzungen "künstlich" zu schaffen sind. Im zweiten, automatischen Fall werden nur die Tests ausgewählt, die in dem jeweiligen Zustand möglich sind, eine Vorbereitung des Systems kann daher entfallen. Trotzdem werden alle Betriebszustände durchlaufen.

Da bei der Testautomation das Testziel eine vollständige Testabdeckung ist, sind – in der Regel – beide Vorgehensweisen äquivalent zueinander, mit dem Unterschied, dass durch die Automation weniger manueller Testaufwand anfällt. Wenn keine vollständige Abdeckung bei Testautomation erreicht werden kann, bedeutet dies, dass auch bei normalem Betrieb bestimmter Code nicht ausgeführt wird, was auf einen Fehler in der Spezifikation hindeutet. Während bei manueller stückweiser Ausführung der Tests ein solcher Mangel unbemerkt bleibt, kann er dagegen bei Testautomation festgestellt werden.

Aber es gibt noch einen anderen Grund, weshalb die Testautomation effektiver ist. Durch die Möglichkeit der automatischen Instrumentierung wird die Testauswertung unterstützt. Hierdurch kann die Testabdeckung nachgewiesen werden, die Ergebnisse können für jeden Test aufgezeichnet und ausgewertet werden. Erst durch diese Unterstützung des automatischen Produktionsprozesses können die Tests im laufenden Betrieb kontinuierlich durchgeführt und der Nachweis der Korrektheit erbracht werden.

### **1.2.8**

## **Mehr erreichen durch strategische Entscheidungen**

Die Möglichkeiten, die gute Organisation bietet, um Komplexität und Aufwand zu reduzieren, führt beispielsweise direkt zu der Frage, ob es wirklich notwendig ist, dass ein Entwickler tage-, wochen-

oder monatelang vor seiner Apparatur sitzen muss, um schrittweise die in ihr enthaltene Software zu testen, und er diese Prozedur teilweise oder vollständig bei der nächsten Änderung wiederholen muss.

Nach unseren bisherigen Ausführungen ist dies eine ineffiziente Ausnutzung seiner Fähigkeiten. Was trägt mehr zum Erfolg und zur Effizienz der Entwicklung bei: seine Ausbildung bzw. Erfahrung im Testen oder seine Fähigkeit, den Testaufwand zu reduzieren? Umgekehrt muss gefragt werden, wie wertvoll ein Entwickler für seinen Betrieb ist, der seine Aufgabe allein darin sieht, gute Qualität durch hohe Kosten zu erreichen.

Hierbei spielt es keine Rolle, ob dies der Stand der Technik ist, ob alle Entwickler so vorgehen. Entscheidend ist, in welcher Weise ein Entwickler dabei hilft, die Wettbewerbsfähigkeit seines Betriebes durch geeignete Innovation zu stärken, und damit auch den Erhalt seines Arbeitsplatzes zu sichern.

Die wesentlichen Punkte zum Wert strategischer Entscheidungen arbeiten wir zunächst wieder an einem Alltagsproblem heraus.

Betrachten wir die Leistungsfähigkeit eines Speditors. Er benötigt einen Führerschein, einen Lkw und er muss die Route vom Abhol- zum Zielort planen. Wo liegt nun das größte Potenzial, um Aufwand und Zeit einzusparen? In der Fahrausbildung, in der Wahl des Fahrzeugtyps oder in der Routenplanung?

Unsere Meinung ist: durch gezielte Planung des Weges kann er am meisten einsparen. Gute Fahrausbildung und ein schneller Lkw nutzen ihm nichts, wenn er lange im Stau stehen muss oder er sich verfährt. Routenplanung lernt er aber nicht in der Ausbildung, obwohl sie wesentlicher Teil des ausgeübten Berufes ist.

Will er erfolgreich sein, muss er aber fit auf allen Gebieten sein, die seine Berufsausübung betreffen. Wenn er nicht den Wert strategischer Maßnahmen – die Routenplanung – erkennt, wird er verlieren. Das impliziert auch, dass er sich mit Dingen beschäftigen muss, die nicht Bestandteil seiner Ausbildung waren, die aber relevant für seine Tätigkeit sind bzw. im Laufe der Zeit relevant werden.

Nur ein Mitarbeiter, der die Effizienz seiner Arbeit ständig analysiert und verbessert, wird für seinen Betrieb mittel- bis langfristig ein wertvoller Mitarbeiter sein. Kümmert er sich nicht um die Effizienz seiner Arbeit oder versucht, seinen Arbeitsplatz durch hohe Komplexität der Arbeitsvorgänge und daraus resultierenden hohen Aufwand zu sichern, wird er ihn verlieren. Außerdem wird er leicht austauschbar, denn hohe Komplexität können viele erzeugen, während die Beschränkung auf minimale Komplexität bei vorgegebener Funktionalität ausreichende Qualifikation und Motivation voraussetzt.

*These 5  
Die ständige  
Verbesserung  
der Effizienz  
sichert das Über-  
leben im Wett-  
bewerb.*

Übertragen wir nun das beschriebene Szenario auf die Welt der Software: der Fahrausbildung entspricht die Ausbildung in Programmiersprachen und -methoden, die Fahrzeugtypen entsprechen den Softwarewerkzeugen, und der Routenplanung entspricht das Prozessmodell der Softwareentwicklung.

So wie der Spediteur handeln muss, wenn er immer lange Zeit im Stau hängen bleibt, so sollte der Softwareentwickler erkennen, dass sich etwas ändern muss, wenn seine Effizienz immer wieder durch gewisse Tätigkeiten leidet, wie beispielsweise durch das zeitraubende Testen.

Um effizienter zu werden, wird es ihm nichts nutzen, eine Programmiersprache durch eine andere zu ersetzen, oder eine manuelle Testmethode durch eine andere. Solange er das grundsätzliche Problem nicht durch eine geeignete strategische Entscheidung löst, wird er keinen Erfolg haben. Umgekehrt werden die erfolgreich sein, die ein Problem global angehen, aus der Welt der Details heraustreten und die übergeordneten Zusammenhänge sehen, und auf dieser Ebene eine Lösung suchen.

Dem Spediteur nutzt im Stau ein starker oder schneller Lkw nichts, dem Softwareentwickler ein besseres Testwerkzeug nichts, solange er immer noch selbst intensiv in den Testprozess eingebunden ist ("manuelles Testen"). Der Spediteur muss den Stau umfahren, der Softwareentwickler das manuelle Testen vermeiden.

Er kann es vermeiden, wenn er in den Vorphasen geeignete Maßnahmen trifft, beispielsweise durch ein (formaleres) Vorgehen, so dass entweder Tests entfallen oder automatisch generiert werden können, das ist die notwendige strategische Entscheidung, die er treffen muss.

*These 70  
Nur durch geeignete strategische Entscheidungen können wesentliche Verbesserungen erreicht werden.*

### 1.2.9

## Interdisziplinäre Kooperation – eine effektive Strategie

Die Zusammenarbeit zwischen Experten der Softwareentwicklung und Anwendern von Software bietet große Chancen, die Effizienz beider Partner zu erhöhen. Wettbewerbsvorteile können aber nur dann entstehen, wenn der vom Anwender benötigte Softwareproduktionsprozess schnell und zu geringen Kosten bereitgestellt und flexibel an zukünftige Bedürfnisse angepasst werden kann.

Ob sich solche Kooperationen auch tatsächlich entwickeln können, hängt von der Marktsituation ab. Ermöglicht der Markt solche Synergien oder verhindert er sie? Wir werden zuerst die Chancen beschreiben und dann mögliche Hemmnisse diskutieren.

### 1.2.9.1

#### **Synergie durch Kooperation**

Für die Entwicklung von Software gibt es bisher nur die Ansätze 1 und 2, der dritte Weg öffnet sich bei Automation.

1. Ein Hersteller bringt aufgrund seiner Kenntnisse über das Anwendungsgebiet ein Produkt auf den Markt,
2. Ein Anwender entwickelt im eigenen Betrieb die benötigte Software, oder beauftragt einen externen Entwickler.
3. Ein Anwender konfiguriert einen Produktionsprozess, um ein spezifisches Produkt bei niedrigen Kosten und kurzer Lieferzeit zu erhalten, ohne selbst entwickeln zu müssen.

Im ersten Fall ist die finanzielle Investition für einen Anwender gering, und das damit verbundene Risiko auch. Das Produkt ist praktisch sofort verfügbar. Aber möglicherweise werden nicht alle Anforderungen des Anwenders abgedeckt. Seine Konkurrenten verfügen über die gleichen Fähigkeiten, wenn sie dieses Produkt kaufen. Dagegen trägt der Produktentwickler ein hohes Risiko, er muss in Vorleistung gehen, und kann nicht sicher sein, ob er einen ausreichenden Erlös erzielt.

Im zweiten Fall ist – heute – ein großes finanzielles Engagement durch den Anwender notwendig, bei entsprechend hohem Risiko. Die Wahrscheinlichkeit ist größer, dass das Ergebnis seinen Wünschen entspricht, und er erhält möglicherweise einen Vorsprung gegenüber seinen Mitbewerbern, aber er muss warten, bis die spezifische Entwicklung abgeschlossen ist. Das Risiko für den beauftragten Entwickler dagegen ist gering.

Bei effizienterer Entwicklung wird die dritte Alternative machbar: Dieser Weg vereinigt die Vorteile der beiden früheren Möglichkeiten, und vermeidet deren Nachteile, für beide Partner.

Der Entwickler bringt hierbei seine Fähigkeit ein, schnell und preiswert ein kundenspezifisches Produkt bei hoher Qualität herstellen zu können. Der Anwender muss über seine Anforderungen seine Expertise einbringen. Dadurch entsteht eine Synergie für beide Partner.

Diese Synergie resultiert aus der Ausschöpfung von Fähigkeiten beider Partner, die erst durch die Kooperation möglich wird. Während der Entwickler die Fähigkeit besitzt, bestimmte Probleme optimal zu lösen, kennt er nicht die spezifischen Probleme des Anwenders. Dieser kennt zwar die Probleme, kann sie aber nicht optimal lösen.

Zur Zeit scheint aber – leider – nach unseren Beobachtungen ein gegenteiliger Trend zu entstehen. Die Anwender minimieren ihr Risiko, indem sie auf Standardprodukte und -methoden setzen, und hoffen, dass auf diese Weise ihre Probleme gelöst werden können.



Die Entwickler dagegen minimieren ihr Risiko, und bringen Produkte bzw. Methoden und Werkzeuge mit unspezifischen Eigenschaften auf den Markt, um möglichst viele Anwender ansprechen zu können. Eine Schere, die immer weiter aufgeht.

Aus unserer Sicht leidet darunter die Innovation insgesamt, die Fortschritte der Anwender von Software werden begrenzt, sie können nicht das anbieten, was eigentlich möglich wäre. Die Wettbewerbssituation des einzelnen Anbieters ist insgesamt nicht stark gefährdet, da es prinzipiell allen Mitbewerbern gleichermaßen ergeht. Aber es entsteht eine labile Marktsituation, aus der der erste, der ein effizienteres Verfahren einsetzt, als Sieger hervorgeht, während die Überlebenschancen der Mitbewerber nicht vorhersagbar bzw. gering sind.

In einer solchen labilen Situation entsteht wie bei einem Vulkan allmählich ein hoher Druck, der sich dann plötzlich über einen Ausbruch entlädt, und dabei die Umweltbedingungen ("Wettbewerbsbedingungen") stark und unkontrolliert verändert.

Ein Verfahren, das erheblich mehr Effizienz in die Softwareentwicklung bringt, wird den aufgezeigten dritten Weg ermöglichen, und den beteiligten, kooperierenden Partnern Vorteile bringen. Die früher beschriebene Aufgabenteilung zwischen "Mensch und Maschine" besitzt u.E. dieses Potenzial.

#### **1.2.9.2**

#### ***Evolutionsshemmnisse***

Wir wollen nun die Erfolgsaussichten solcher Kooperationen betrachten. Nur wenn sie gut sind, werden sie sich entwickeln und durchsetzen können. Dabei spielt die Struktur des Marktes eine entscheidende Rolle.

Zur weiteren Betrachtung unterscheiden wir zwischen zwei verschiedenen Typen:

- einen Markt, bei dem freier Wettbewerb möglich ist, und
- einen Markt, bei dem der Wettbewerb eingeschränkt ist.

Ein freier Wettbewerb begünstigt die Innovation, weil zur effizienten Deckung des Bedarfs neue Produktionsverfahren entwickelt und verbessert werden müssen. Hier stehen die Interessen der Abnehmer im Vordergrund. Bei eingeschränktem Wettbewerb stehen dagegen die Interessen von Gruppen im Vordergrund, und Innovation kann blockiert werden, wenn sie nicht Vorteile für die jeweilige Gruppe bringt.

Wenn die Interessen der Abnehmer und ihr Verhalten am Markt nicht koordiniert werden, ergibt sich der Druck auf die Anbieter aus der Majorität der Bedürfnisse, und die Evolution wird daher nicht gehemmt. Werden dagegen die Interessen von Gruppen und ihr

Verhalten gezielt gesteuert, wird die Evolution beschränkt oder überhaupt nicht ermöglicht. Eine solche Beschränkung kann verschiedene Gründe haben.

Um die verschiedenen Hemmnisse verstehen zu können, müssen wir die Beziehungen der Partner am Markt näher definieren. Wir hatten bisher nur eine "2er Beziehung" betrachtet zwischen

Entwickler<sup>7</sup> ← Werkzeug- bzw. Methodenlieferanten (WML)

und müssen noch eine "3er Beziehung" hinzunehmen: indem wir die Rolle des Abnehmers des Entwicklers, des "Endkunden" betrachten:

Endkunde ← Entwickler ← Werkzeug- bzw. Methodenlieferant

Wie bereits im letzten Abschnitt beschrieben, können auch die Abnehmer (Entwickler, Endkunde) ihre Interessen koordinieren, das kann bewusst oder unbewusst geschehen.

So kann der Endkunde ein Interesse an einer Harmonisierung haben, und durch Standards oder andere Vorschriften die Entwickler an gewisse Vorgehensweisen binden. Ist er mächtig genug, so wird er viele Entwickler steuern, und damit indirekt auch den WML.

Eine ähnliche Situation entsteht, wenn ein Entwickler den Markt dominiert, oder es sich um einen geschlossenen Markt handelt, zu dem nur eine begrenzte Zahl von Firmen Zugang hat. Wenn eine Entwicklungsfirma nicht durch Wettbewerb gezwungen ist, auf Effizienz zu achten, werden firmenpolitische Entscheidungen im Vordergrund stehen. Kann eine solche Firma ihr Personal nicht auslasten, oder müssen sich Investitionen noch amortisieren, so wird sie den Einsatz einer neuen Technologie verzögern.

Ähnliches gilt für einen WML. Besitzt ein WML eine marktherrschende Stellung, wird er den Markt nur mit Produkten versorgen, die zu seiner Strategie passen. U.a. wird er primär an einer Gewinnmaximierung seiner Investitionen interessiert sein, eine (zu frühe) Einführung neuer Produkte könnte zu Ertragseinbußen führen.

Bei solchen Marktverhältnissen kann der Entwickler zwischen beiden Fronten eingeklemmt werden. Ein mächtiger Endkunde kann über seine dominierende Position am Markt versuchen, die Preise zu drücken. Kann der Entwickler dann nicht seinen Produktionsprozess optimieren, weil der Endkunde ihn auch noch mit Vorschriften einengt, oder der WML ihm dazu keine Unterstützung bietet oder bieten kann, so kommt er in starke Bedrängnis.

---

<sup>7</sup> Wir sehen hier den "Entwickler" nicht nur als Person, sondern eher als Firma.

In dieser Situation nutzt es ihm nicht, dass langfristig gesehen auch Endkunde und WML davon negativ betroffen werden können. Wenn keine leistungsfähigen Entwickler mehr zur Verfügung stehen, können die Endkunden auf ihrem Markt die Wettbewerbsfähigkeit verlieren. Verschwinden viele Entwickler vom Markt, dann sinkt auch der Umsatz eines WML.

Außerdem können sich die Machtverhältnisse durch den sog. "Lopez-Effekt"<sup>8</sup> umkehren: durch den durch Preisdruck eingeleiteten Schrumpfungsprozess verbleiben nur noch wenige Anbieter am Markt, die dann aufgrund ihres Quasi-Monopols dominieren, der Endkunde kann dann nicht mehr wie früher seine Position durchsetzen.

### **1.2.9.3**

#### **Auswege**

Besonders groß ist die Gefahr, dass ein Entwickler zwischen den beiden Machtpolen eingeklemmt wird, wenn er es versäumt, mit eigenen Strategien in seinem Betrieb Entwicklungsfähigkeiten aufzubauen, die es ihm erlauben, aus dieser Notsituation zu entkommen oder ihn davor bewahren.

Der aktuelle Status der Softwareentwicklung mit dem großen Bedarf an Entwicklungsressourcen und hohen Kosten begünstigt den Trend, auf einen WML zurückzugreifen, statt eigene Entwicklungsprozesse zu benutzen, die optimal auf die eigenen Bedürfnisse abgestimmt werden können. Somit kann ein Entwickler in starke Abhängigkeit von einem WML geraten.

Einen Ausweg bieten effiziente Entwicklungsprozesse, deren Realisierung für einen Entwicklungsbetrieb keine hohen Kosten verursachen, aber mehr Flexibilität bringen.

### **1.2.10**

#### **Mehr Zuverlässigkeit und Effizienz durch Automation**

Eine wesentliche Frage bzgl. des Beispiels der Elektroinstallation wurde bisher noch nicht beantwortet: „Hat sich der Handwerker wirklich an die Regeln gehalten?“ bzw. „Ist der Generator wirklich fehlerfrei?“ Sollte man nicht doch eine Verifikation durchführen, um sicher zu sein?

---

<sup>8</sup> José Ignacio Lopez drückte in den 90er Jahren erheblich die Preise der Automobilzulieferer, wodurch es zu einem Massensterben von Zulieferern kam.

Natürlich bleibt immer ein gewisses Restrisiko, dass die Vorschriften nicht eingehalten werden. Die Frage ist daher, welche Fehlerwahrscheinlichkeit wir tolerieren können.

Wenn wir maximal in unserem Leben 1,000 Bilder aufhängen (in einem Zeitraum von 50 Jahren durchschnittlich etwa 2 Bilder pro Monat), wie viele schlechte Handwerker darf es geben, damit wir mit einer Sicherheit von 99.999999% keinen Stromschlag erhalten? Wir wollen diese Frage nicht näher quantitativ untersuchen, sondern auf folgende Aspekte hinweisen:

- Eine absolute Sicherheit gibt es nicht, man muss mit einer gewissen Fehlerwahrscheinlichkeit leben, und aus dem akzeptablen Limit Anforderungen an die Verlässlichkeit ableiten.
- Die Verlässlichkeit ist umso höher, je größer die Reproduzierbarkeit des Vorganges ist.

Da Handwerker Individuen sind, kann man nicht von 100% Reproduzierbarkeit ausgehen, d.h. das Ergebnis hängt von Ausbildung, Tagesform und persönlicher Zuverlässigkeit ab. Wird ein Automat eingesetzt, wie beispielsweise bei der Produktion einer Wand eines Fertighauses, dann ist die Reproduzierbarkeit sehr hoch, praktisch 100%, abgesehen von einer Störung des Automaten. Trotz dieses (geringen) Restrisikos ist die Zuverlässigkeit bei einem automatischen Prozess um Größenordnungen höher als bei "manueller" Ausführung.

Übertragen auf die Softwareentwicklung folgt, dass die automatische Erzeugung von Code neben hoher Effizienz die höchste Zuverlässigkeit bietet. Ist der Produktionsprozess genau bekannt – ohne solchen detaillierten Kenntnisse kann nicht automatisiert werden, sind auch die Qualitäts- und Prüfkriterien bekannt. Bei der Fertigung einer Wand kann man an den entsprechenden Stellen Sensoren anbringen, die überprüfen, dass die Leitungen vorschriftsgemäß verlegt wurden. Im Fall eines Softwareproduktionsprozesses kann man durch einen Stimulator den Testfall automatisch erzeugen ("Verlegen der Kabel"), und ebenso mit einen "Observer" das Antwortverhalten aufzeichnen, überwachen und ggf. eine Fehlermeldung auslösen („wurden die Kabel richtig verlegt?“).

Die Automation erschließt somit vorher unbekannte Möglichkeiten der Prozessoptimierung. Daher sollte nicht gefragt werden, „Kann ich das vielleicht automatisieren?“, sondern „Warum ist das noch nicht automatisiert?“

Bei der Einführung der Automation genügt es nicht, Automaten einzusetzen, man muss ihren Einsatz planen und effizient gestalten. Hierzu wollen wir ein einfaches Beispiel zur Testautomation bringen.

Betrachten wir die Aufgabe, eine grafische Benutzeroberfläche zu testen, und hier wieder speziell eine Menüauswahl wie beispielsweise bei der englischen Version 1.7 des Internet-Browser "Mozilla™" die Einstellung des Formats der Zeichenkodierung:

View → Character Encoding → Auto-Detect → Off

Als Testziel definieren wir das Setzen des Schalters "Off" über das Menü. Wir nehmen hier (ohne Beschränkung der Allgemeinheit) an, dass der Begriff "Off" innerhalb des Menüs eindeutig ist, und der Test nur auf die Auswahl des Menüpunktes zielt, nicht auf den Nachweis, dass alle Untermenüs vorhanden sind und auch erscheinen.

Um automatisch (z.B. per Testskript) "Auto-Detect" auszuschalten, geht man üblicherweise folgendermaßen vor (wir skizzieren den Vorgang nur):

```
mozilla.menu.View.click  
mozilla.menu.Character Encoding.click  
mozilla.menu.Auto-Detect.click  
mozilla.menu.Off.click
```

Der automatische Test folgt also genau dem von früher bekannten manuellen Testablauf und erfordert, die Eingabe von vier Befehlen. Das oben definierte Testziel verlangt aber nicht eine 1:1 Abbildung der manuellen Vorgehensweise. Die Fähigkeit eines Testwerkzeuges, selbst den Pfad zu "Off" herauszufinden und das Testziel mit nur einem Befehl zu erreichen, wird also nicht genutzt.

Neben der Einsparung von drei Anweisungen führt die Vereinfachung auf eine Anweisung noch zu einem weiteren Vorteil: die bessere Wartbarkeit des Testskriptes bzw. Unabhängigkeit (in gewissen Grenzen) von Änderungen durch Wartung des Browsers.

Natürlich sind die Einsparungen bei diesem Beispiel nicht besonders groß, aber wir haben ein einfaches und verständliches Beispiel gewählt, um das Prinzip erläutern zu können.

### 1.2.11

## Ohne richtige Dimensionierung geht nichts

Wenn ein Handwerker die Elektroinstallation "implementiert", überlegt er sich, was die mögliche Last sein kann. Er plant nicht nur die "Leitungsarchitektur", sondern auch die Systemperformance, also den Querschnitt der Leitung entsprechend der erwarteten Belastung. Das verhindert spätere Beschädigungen oder Nutzungseinschränkungen.

In der Softwareentwicklung ist es dagegen üblich, sich erst nur auf die statischen Teile, wie Architektur, Topologie, Funktionalität zu konzentrieren, während durchaus das Risiko wegen zu geringer Performance (zu hohe CPU-Last, zu wenig Speicher, zu lange Antwortzeiten) inzwischen bekannt ist. Ob richtig dimensioniert wurde, erfährt man beim ersten "Einschalten". Brennen die Leitungen nicht durch, oder werden sie nur mäßig warm, hat man Glück gehabt, das Projekt ist gerettet. Durch diese risikoorientierte Vorgehensweise scheitern etwa 25% der Softwareprojekte: sie "brennen" beim ersten Einschalten ab.

Der Elektriker hat es sicher auch einfacher. Er hat den Überblick über den Verbrauch. Erstens weiß er – durch die Kirchhoffschen Gesetze und die daraus für die Handwerker abgeleiteten Regeln, dass die Gesamtleistung sich linear aus der der einzelnen Verbraucher berechnen lässt. Zweitens kennt er die Anzahl der Verbraucher (bzw. Steckdosen) und ihre ungefähre Leistung.

Softwareentwickler haben es leider schwerer. Der "Verbrauch" eines komplexen Softwaresystems ist nicht so leicht überschaubar. Umso wichtiger ist es aber, die Überschaubarkeit durch gute Organisation zu erhöhen, und gleichzeitig Regeln einzuführen, um die Eigenschaften des geplanten Systems besser bestimmen zu können.

Bereits bei der eigenen Software ist es schwierig, die Performance vorhersagen zu können. Durch Nutzung von Fremdsoftware wie Betriebssystem, grafischen Oberflächen, Datenbanken usw. wird dieses Problem aber noch viel größer. Die Möglichkeit, schnell nach Beginn der Entwicklung eine repräsentative Version ausführen zu können, hilft, dieses Risiko zu verringern.

Ein automatischer Produktionsprozess kann auch diese Anforderung abdecken. Bei geeigneter Organisation kann ein Generator schon die erste einfache Idee in ein repräsentatives, ausführbares Programm unter Berücksichtigung der Schnittstellen zu anderen Programmen überführen.

Damit besteht bereits von Beginn der Entwicklung an die Möglichkeit, die "Last" zu prüfen und rechtzeitig für Korrekturen zu sorgen.

### **1.2.12**

#### **Komplexität – weniger ist mehr**

Die Beherrschung hoher Komplexität gilt üblicherweise als Nachweis von Kompetenz, woraus dann wieder ein Anspruch auf eine Führungsposition oder angemessen hohe Bezahlung abgeleitet wird. Aus dieser Sicht ist die Entwicklung von Systemen bzw. Produkten

hoher Komplexität für einen Entwickler erstrebenswert. Auch in der Ausbildung wird auf Beherrschung hoher Komplexität geachtet, damit die späteren Ingenieure die Anforderungen ihres Berufes möglichst gut erfüllen können.

Hohe Komplexität kann aber auch von Nachteil sein. Fehlerrate, Entwicklungskosten und –zeit eines komplexen Produktes sind meistens auch höher. Daher muss zwischen diesen zueinander in Konflikt stehenden Zielen in der Praxis ein Optimum gefunden werden: minimale Komplexität bei maximaler Leistung, gute Funktionalität und Zuverlässigkeit bei niedrigem Preis, geringen Entwicklungskosten und –zeit.

Aus der Praxis wissen wir, dass es viel schwieriger ist, die gleiche Funktionalität mit niedriger Komplexität zu erreichen, als mit hoher. Minimale Komplexität für eine bestimmte Aufgabe zu erzielen, das zeichnet den Fachmann aus und weist Kompetenz nach.

Meistens findet man zuerst eine komplizierte Lösung, von der aus man die einfachere Lösung ableiten muss – vorausgesetzt, man hat dieses Ziel. Dazu ist es notwendig, sich ein Komplexitätslimit zu setzen, das nicht überschritten werden darf. Wenn doch, muss eine einfachere Lösung gefunden werden. Die Erfahrung zeigt, dass die einfachere Lösung immer viel besser ist. Komplexe Lösungen verhalten sich wie ein Kartenhaus, das schon beim Anhauchen zusammenfällt. Sie entsprechen einem labilen Gleichgewicht, während einfache Lösungen sich stabil und robust verhalten, auch hinsichtlich Wartung.

Kürzlich hatten wir ein sehr komplexes Problem zu lösen, das uns durch das benutzte Betriebssystem aufgezwungen worden war. Wir mussten einen Mouse-Click emulieren, der Anwendung vom Programm her vortäuschen, dass auf eine bestimmte Position in einem Fenster einer grafischen Benutzeroberfläche ein Mouse-Click ausgeführt wurde. In der Version A des Betriebssystems mussten wir dazu eine Mindestverzögerung zwischen "mouse-down" und "mouse-up" einbauen, so wie es der Realität entspricht.

Diese Version ging als Untermenge in der folgenden Version B des Betriebssystems auf. Dort stellten wir fest, dass die Einspeisung von "mouse-down" und "mouse-up" bei Verwendung der zu A neu hinzugekommenen, zweiten grafischen Oberfläche, "atomar" sein musste, also ohne Verzögerung. Nach "mouse-down" ging das System in einen Zustand, der den Empfang unseres erzeugten "mouse-up" ausschloss. Wir passten daraufhin die Software einheitlich an und mussten an mehreren Stellen ändern.

Bei weiteren Tests stellten wir dann fest, dass die Untermenge der früheren Version A doch die Verzögerung benötigte. Es gab also zwei unterschiedliche Implementierungen der gleichen Operation.

*These 10  
Ein Ziel mit  
niedriger  
Komplexität zu  
erreichen weist  
mehr  
Kompetenz  
nach.*

Die frühere Version A war integriert worden, und für die neue Version wurde eine andere Funktionalität zusätzlich implementiert.

Eine mögliche Lösung wäre gewesen, vor dem emulierten Mouse-Click abzufragen, von welchem Typ das zu bedienende Objekt ist, und dann atomar oder nicht-atomar die Aktionen einzuspeisen. Das hätte aber weitere Änderungen erfordert. An dieser Stelle zogen wir die "Notbremse", das Komplexitätslimit war überschritten worden. Wir begannen, den gesamten Lösungsweg zu überprüfen und nach Möglichkeiten der Vereinfachung zu suchen.

*These 20  
Weniger ist mehr*

Die zugehörige Dokumentation wurde konsultiert, und nach ca. 2 Stunden hatten wir eine einfache Lösung gefunden. Wir konnten daher vereinheitlichen, für beide Objekttypen nicht-atomar einspeisen, und damit eine gemeinsame und einfache Lösung realisieren.

*These 71  
Das Setzen  
eines Komplexitätslimits er-  
zwingt einfache  
Lösungen.*

Bei Umstellung auf die Version B hatten wir die Standardversion einer Option bei einem Funktionsaufruf gewählt, und es gab zu diesem Zeitpunkt keinen Grund, sich anders zu entscheiden. Aber genau diese Entscheidung verursachte den Konflikt. Das Problem war, dass wir beim Auftreten des Konfliktes keinen Zusammenhang mehr sehen konnten, die frühere Entscheidung lag schon mehr als ein Jahr zurück. Nur durch die Weigerung, eine Lösung hoher Komplexität zu realisieren, hatten wir Gelegenheit, diese Abhängigkeit doch zu entdecken und das Problem zufriedenstellend lösen zu können.

In dem Moment, als wir die Notbremse zogen, bestand für uns zwar das Risiko, dass die Suche nach einer einfacheren und besseren Lösung zusammen mit deren Implementierung mehr Zeit benötigen würde, als eine mögliche komplexe Lösung. Die Erfahrung zeigt aber, dass dieses Risiko erstens recht gering ist, und zweitens sich eine grundsätzliche Neubetrachtung der Situation aufgrund der dann möglichen Vereinfachung in nahezu allen Fällen lohnt. Diese Vereinfachung entsteht in unserem Beispiel, weil nur noch ein Zweig anstatt zweier Zweige existiert, und sich somit Wart- und Testbarkeit erheblich verbessern.

Die Beibehaltung der komplexen Lösung hätte dagegen das Risiko erhöht. Denn es war nicht absehbar, wie viele komplexe Hilfskonstrukte in Zukunft notwendig gewesen wären, um die dieses "Kartenhaus" zu stützen. Außerdem hat die Suche nach einer besseren Lösung das Verständnis des Problems und damit sowohl die Stabilität als auch das Vertrauen in die Gesamtlösung deutlich erhöht.

Für die Festlegung des Komplexitätslimits gibt es leider keine Regel. Jeder muss es gemäß seiner Erfahrung definieren. Prinzipiell kann man sich am eigenen Verständnislimit orientieren. Im vorgestellten Fall merkten wir, dass wir die entstehende Lösung des für das Gesamtsystem kritischen Problems nicht mehr ohne Hilfsmittel



hätten überschauen können, und sahen unsere Komplexitätsgrenze damit erreicht.

Immer sollte das Ziel aber sein: je niedriger, desto besser. Nur auf diese Weise können frühzeitig Fehlentwicklungen entdeckt und verhindert werden.

Hätten sich die Gelehrten bei der Lösung des Gordischen Knotens ein niedrigeres Limit gesetzt, so hätten sie wohl früher bemerkt, dass sie das Problem und die Anforderungen an eine Lösung nicht ausreichend verstanden haben und nach einem neuen Lösungsansatz suchen müssen.

### 1.2.13

## Mensch-Maschine-Schnittstelle

Ein Mensch kann mit einem Rechner in zweifacher Weise Kontakt aufnehmen:

- als Entwickler, und
- als Anwender des entwickelten Programmes.

Da der Entwickler auch wieder Programme benutzt, ist er gleichzeitig auch Anwender. Unterschiede bestehen nur in der Art der Anwendung. Die Dialoge des Entwicklers können denen eines Anwenders ähneln, hauptsächlich in einer frühen Phase wie während des Entwurfes, d.h. er definiert über eine grafische Benutzeroberfläche eines Werkzeuges das zu entwickelnde System. Er kann aber auch die Schnittstelle zu einer Programmiersprache nutzen, beispielsweise während der Codierungsphase.

Eine "Mensch-Maschine-Schnittstelle" (MMI, "Mensch-Maschine-Interaktion", "Man-Machine-Interface") muss so gestaltet sein, dass ein Anwender seine Ziele einfach und schnell erreichen kann. Aus der Praxis wissen wir, dass dies nicht immer optimal gelingt.

Der Entwurf eines Dialogs mit dem Anwender ist meistens kompliziert, besonders wenn der Dialog komfortabel sein soll. Entsprechend aufwändig ist die Implementierung. Um Kosten und Zeit zu sparen, müssen dann Kompromisse geschlossen werden, und die Schnittstelle bietet dann möglicherweise nicht den erwarteten und notwendigen Komfort.

Für die effiziente Implementierung einer Schnittstelle gelten die an anderen Stellen bereits getroffenen Aussagen. Wir werden uns daher hier auf die Gestaltung einer MMI konzentrieren.

*These 37  
Die Komplexität  
einer MMI kann  
mit einem Rechner reduziert  
werden.*

*These 54  
Ein Prozess  
muss dem Anwender helfen,  
die Informationsmengen zu bewältigen*

Die effiziente Gestaltung einer MMI erfordert u.a. Einfachheit und Verständlichkeit des Dialoges einschließlich Fehlerprävention. Durch den Dialog soll ein Anwender ent- und nicht belastet werden. Ein Rechner sollte die einzugebende Informationsmenge minimieren und auf Korrektheit und Vollständigkeit überprüfen. Allgemeiner ausgedrückt, kann ein Rechner die Komplexität einer Mensch-Maschine-Schnittstelle reduzieren.

In dieser Form lassen sich die immensen Ressourcen eines Rechners sehr sinnvoll nutzen, neben dem bereits beschriebenen Einsatz im Softwareproduktionsprozess. Aber eine solche Nutzung muss ebenfalls organisiert werden.

Wenn beispielsweise sich überlappende Information abgefragt wird, und der Rechner dann Widersprüche erkennt, so ist das sicher ein Fortschritt, indem der Rechner einen Fehler frühzeitig erkennt, aber nicht gut genug. Der Anwender muss überlegen, worin der Fehler besteht, und muss neu eingeben.

Sinnvoller wäre es, wenn nur nicht-redundante Information abgefragt würde, und der Rechner diese Information in die geeignete Form bringen würde. An einem einfachen Beispiel für die Definition einer Funktion in den Programmiersprachen Ada und C erläutern wir diese Forderung. Kenntnisse in diesen beiden Sprachen werden nicht vorausgesetzt. Um das Problem verstehen zu können, muss man nur wissen, dass

- eine Funktion definiert wird durch ihren Namen, einen Rückgabewert und eine Parameterliste,

In Ada wird diese Definition "function declaration" genannt, in C "prototype". Diese Deklaration wird benutzt, um die Funktion im gesamten Code bekannt zu machen.

- einen Ausführungsteil benötigt bestehend aus der Wiederholung der Definition und Anweisungen.

In Ada wird dieser Teil "function body" genannt, in C ist es die "function".

Hier ein Beispiel einer Ada-Funktion:

Deklaration:

```
function myFunction(para1 : type1; para2 : type2) return type3;
```

Body:

```
function myFunction(para1 : type1; para2 : type2) return type3 is  
<Anweisungen>
```

Einsichtig ist, dass beim "Body" die Namen der Parameter angegeben werden müssen, weil sie in den Anweisungen im Funktionsrumpf benötigt werden. Dagegen würde bei der "Declaration" die Angaben der Typen ausreichen, um feststellen zu können, dass ein Aufruf mit den richtigen Parametertypen erfolgt. Die Angabe von Parameternamen in der Deklaration stellt nicht nur redundante In-

formation dar (die Namen werden beim Body ebenfalls definiert), sondern die Information wird an dieser Stelle überhaupt nicht benötigt.

Einziger Nutzen könnte die Dokumentation des Zwecks der Parameter durch ihre Namen sein, da in vielen Fällen nur die Funktionsdeklaration im Quellcode zur Verfügung steht, nicht aber der Funktionsrumpf, etwa weil eine Bibliothek nur in Binärform verfügbar ist. Selbst dann sollte es allerdings dem Entwickler überlassen bleiben, ob er die Option der Dokumentation wahrnimmt.

Ada fordert aber nicht nur, dass die Parameternamen bei der Deklaration angegeben werden müssen, sondern prüft auch deren Übereinstimmung mit den Namen, die beim Body angegeben werden. Gibt man versehentlich bei der Deklaration falsche Namen an, führt das zu einem Fehler, wie in folgendem Fall, obwohl die Information für die Übersetzung überhaupt nicht benötigt wird:

Deklaration:

```
function myFunction(par1 : type1; par2 : type2) return type3;
```

Body:

```
function myFunction(par1 : type1; para2 : type2) return type3 is  
<Anweisungen>
```

In C dagegen wäre ein solcher Konflikt ausgeschlossen, denn folgender Code ist zulässig:

Prototype / Deklaration

```
type3 myFunction(type1, type2);
```

Function / Body

```
type3 myFunction(type1 para1, type2 para2) {  
<Anweisungen>  
}
```

Bei Ada wird also nicht nur nicht erforderliche Information verlangt, sondern unnötigerweise die redundante Information auch noch auf Konsistenz überprüft.

Die MMI wird also unnötig komplex. Dieses Beispiel ist relativ trivial. Wir haben es gewählt, weil es verständlich ist. In der Praxis treten aber viel erheblichere Probleme auf, wenn die Daten- bzw. Codemengen groß werden, und Teile durch Wiederholung von Information voneinander abhängen.

Bei der Tabellenkalkulation beschreibt man Abhängigkeiten durch Regeln bzw. Rechenausdrücke. Wenn eine Zahl in einem Formblatt verändert wird, so werden alle davon abhängigen Daten automatisch neu berechnet. In Programmen gibt es ebenfalls viele solcher Abhängigkeiten im Code, deren einfache Verwaltung wird aber durch Programmiersprachen nicht unterstützt.

*These 38  
Bei guter Organisation muss der Anwender seine Welt nicht verlassen.*

Das Fehlen solcher Mechanismen erhöht den Aufwand und die Fehlerrate. Die MMI wird unnötig komplex. Durch geeignete, übergeordnete Organisation kann man aber diesen Nachteil von Programmiersprachen beheben, indem man wie bei der Tabellenkalkulation Regeln ("Konstruktionsregeln") einführt, und durch die Hilfe des Rechners Komplexität, Aufwand und Fehlerrate verringert.

Ziel muss daher sein, die MMI so einfach zu gestalten, dass sie selbst bei komplexen Problemen einfach und für den Anwender übersichtlich bleibt, den Anwender bei der Problemlösung (beispielsweise bei der Definition eines Programmes) führt und Fehler frühzeitig erkennt. Zur Umsetzung der kreativen Ideen des Anwenders in ein komplexes System kann ein Rechner eingesetzt werden. Geeignete organisatorische Maßnahmen können gewährleisten, dass diese Transformation auch wieder möglichst einfach und vielseitig verwendbar wird.

Die für den Anwender beste Lösung ist, ihn nicht zu einer anderen Notation zu zwingen, sondern seine eigene Notation zu übernehmen, und die Transformation auf die Standardnotation des Produktionsprozesses vom Rechner durchführen zu lassen.

Wir wissen, dass dies eine Herausforderung ist. Immer wieder hören wir in Diskussionen, dass es schwierig bzw. unmöglich sei, die MMI so zu gestalten, dass ein Programm die Transformation ausführen kann. Die Folge ist, dass sie von Experten manuell durchgeführt werden müsste, mit allen Nachteilen hinsichtlich Aufwand, Zeit und Qualität. Wir werden später zeigen, dass eine automatische Transformation möglich ist. Ihre Realisierung erfordert aber grundsätzliches Umdenken.

### 1.2.14

#### **Ohne Zusatzaufwand unendlich viele Fälle abdecken**

*These 64  
Bei geeigneter Organisation können mit endlichem Aufwand unendlich viele Anwendungsfälle bzw. Operationen abgedeckt werden.*

Manuell erzeugter Code muss immer auf Korrektheit getestet werden, was hohen Aufwand erfordert. Code, der nach bewährten Konstruktionsregeln erzeugt wird, muss nicht getestet werden ("einmal korrekt, immer korrekt"). Besonders effizient wird dieser Ansatz dann, wenn durch einmaligen endlichen Aufwand alle zukünftigen, unendlich vielen Fällen abgedeckt werden können. Unter "Zusatzaufwand" verstehen wir hierbei den Aufwand, der anfällt, wenn Erweiterungen für ein bereits bekanntes Objekt implementiert werden. Der frühere Aufwand zur Erzeugung des Objektes wird dann nicht zu der Erweiterung hinzu gerechnet.

Als Beispiel wählen wir benutzerdefinierte Datentypen, auf denen wir Operationen ausführen wollen. Solche Operationen können sein: Initialisierung, Anzeige, Konvertierung, Überprüfen der Werte, beliebige Metriken usw. Die Implementierung erfolgt über Funktionen, und für jeden Typ und jede dieser Operationen muss eine Funktion implementiert werden.

Bei geeigneter Organisation ist es jedoch möglich, alle Funktionen für beliebige Typen automatisch zu generieren, ohne manuellen Zusatzaufwand, d.h. Aufwand, zusätzlich zu der (unumgänglichen) Definition des Datentyps. Einmalig ist dazu folgender Aufwand notwendig:

- Definition der Operation über eine Regel (in einer geeigneten Notation)
- Implementierung der Operation für jeden Basistyp der benutzten Programmiersprache über eine Funktion. In C sind dies beispielsweise die Typen char, short, int, long, float, double, also eine sehr geringe Anzahl.

Mit diesen einfachen Maßnahmen kann man alle künftigen Fälle abdecken, ohne – außer der Definition des Typs selbst, manuell neuen Code generieren zu müssen.

Wird ein Typ hinzugefügt, so ist sofort die gesamte Funktionalität auch für diesen Typ verfügbar, ohne dass weiterer manueller Aufwand entsteht, abgesehen von der Definition des neuen Typs. Entsprechendes gilt auch, wenn ein Typ entfernt wird.

<i>These 48 Einmal korrekt, immer korrekt</i>
---

## 1.2.15

### Qualitätssicherung

Unter dem Begriff "Qualitätssicherung" versteht man eine Überprüfung auf Konformität mit vorgegebenen Regeln oder Vorschriften bei der Entwicklung, Produktion oder Ausübung einer Dienstleistung. Ein "Qualitätssicherer" sichert also nicht die Qualität des Ergebnisses, sondern bestätigt, dass entsprechend dem Stand der Technik die nötige Sorgfalt angewendet wird. Im Sinne der Norm ISO 9000 (s. ISO9000) muss der "Hersteller" in der Lage sein, die Eigenschaften seines Produktes zu definieren, zu messen und zu bewerten, und den Produktionsvorgang so zu beeinflussen, dass das Produkt die gewünschten Eigenschaften aufweist – innerhalb gewisser zulässiger Fehlergrenzen. Somit ist der Produzent verantwortlich, dass geeignete Verfahren eingesetzt werden, die zu der gewünschten Qualität führen. Nur die Verfügbarkeit solcher Methoden wird von Qualitätssicherern bewertet.

In der Softwareentwicklung werden Vorschriften ("Softwarestandards") angewendet, die sicherstellen sollen, dass bewährte Verfah-

ren zur Implementierung der Software eingesetzt werden, und deren Einhaltung überprüft. Die Überprüfung erfolgt meistens anhand von Dokumentation und Quellcode durch "Einsichtnahme" in die Unterlagen, und Extrapolation der gewonnenen Information auf das spätere Verhalten. Wegen der großen Anzahl von Testfällen können an der Gesamtzahl gemessen nur relativ wenig Fälle – auch bei der üblichen (partiellen) "Testautomation" ausgeführt und ausgewertet werden.

*These 108  
Bei geeigneter  
Organisation  
kann ein Rechner nicht nur  
Software produzieren, sondern  
auch die Qualität  
selbst überwachen.*

Ist der Produktionsprozess bekannt und wird von einem Rechner ausgeführt, dann können vom Rechner aus dem Produktionsprozess auch die Kontrollaufgaben abgeleitet, realisiert und ausgewertet werden, so dass der Entwickler in verständlicher Form die Qualitätsinformation bekommen kann, ohne dafür selbst aktiv zu werden.

Wenn in den Anforderungen ein "Timeout" von 2 Sekunden definiert ist, dann kann der Generator entsprechende Kontrollmechanismen einbauen. Er kennt die Start- und Stopbedingungen, kann eine Überwachung auslösen und eine Überschreitung im Bericht vermerken, oder falls kein Zeitfehler auftritt, dies ebenfalls dem Anwender anzeigen, der dann nur den automatisch erstellten Bericht durchgehen muss.

Ein Generator, der den Code erzeugt, kann auch Maßnahmen treffen, um die sog. "Coverage" zu messen, z.B. wie oft und unter welchen Bedingungen ein Statement ausgeführt wurde.

*These 109  
Die Synergie  
zwischen Erzeugung und Überwachung reduziert die Fehler.*

Die Erzeugung von Code und Überwachungsmechanismen durch einen Produktionsprozess ist kein Widerspruch zu der sonst notwendigen Trennung von Ausführung und Kontrolle. Im Gegenteil, dieser Ansatz bietet die Chance, mehr Fehler zu erkennen als wenn nur ein Teil realisiert würde, oder beide nicht über eine Schnittstelle arbeiten würden. Die erforderliche Abstimmung zwischen den unabhängigen Teilen des Produktionsprozesses deckt Fehler in einem der beiden Teile auf, ähnlich zum bekannten "n-version programming".

Falls verschiedene Generatoren zur Verfügung stehen, kann auch ohne die sonst hohen Kosten "n-version programming" selbst realisiert werden, indem die Generatoren ausgetauscht werden. Falls dabei Schnittstellen angepasst werden müssen, kann das auch automatisch geschehen.

*These 110  
Je größer die  
Bandbreite eines  
Produktionsprozesses, desto  
größer seine  
Zuverlässigkeit.*

Der breitbandige Einsatz von solchen Produktionsprozessen impliziert eine höhere Zuverlässigkeit. Je mehr Anwendungen generiert werden, und je unterschiedlicher die Anwendungen sind, desto größer ist die Wahrscheinlichkeit, dass latente Fehler entdeckt werden, die sonst trotz aller Qualitätssicherungsmaßnahmen nicht gefunden werden. Oft bleiben logische Fehler unentdeckt, weil sie für die Anwendungsfälle korrekte Ergebnisse liefern. Die Software, die von einem wiederverwendbaren Produktionsprozess generiert wird, ist

mit großer Wahrscheinlichkeit zuverlässiger als ein manuell hergestelltes Unikat.

Trotz der erwähnten Vorteile eines Softwareproduktionsprozesses auf die Qualitätssicherung, sind aber noch einige Probleme zu lösen, wenn an die Implementierung eines solchen Prozesses hohe Sicherheitsanforderungen gestellt werden, wie beispielsweise in der Luft- und Raumfahrt.

Geht man von "verified-by-use" aus, also von der Tatsache, dass ein Produkt sich in der Praxis als zuverlässig erwiesen hat, dann bestehen diese Probleme nicht. Für die meisten Anwendungen wird dies zutreffen. Compiler sind ein typisches Beispiel hierfür. Sie arbeiten zuverlässig, ein spezielles Zertifikat wird aber in der Regel nicht verlangt. Wenn sie bei kritischen Anwendungen eingesetzt werden sollen, wird dagegen eine spezielle Prüfung durchgeführt.

Zum Nachweis der Qualität wird zertifiziert oder qualifiziert. Bei der Zertifizierung wird die Konformität mit Qualitätssicherungsstandards durch eine autorisierte Organisation bestätigt. Als "Qualifizierung" bezeichnet man den Prozess der Zertifizierung für ein bestimmtes Anwendungsgebiet und bestimmte Einsatzbedingungen (Camus02).

Für C++ gibt es eine Standard-Testsuite für den Test von Compilern. Für Ada-Compiler ist eine Zertifizierung möglich. Sie bezieht sich aber nur auf die Konformität zum Ada-Standard, nicht auf die Korrektheit eines Ada-Compilers.

Der Einsatz von Codegeneratoren (keine Softwareproduktionsprozesse) in den vergangenen Jahren und ihre Zertifizierung bzw. Qualifizierung hat gezeigt, dass dabei hohe Kosten anfallen können, und zwar für jede Änderung an den Generatoren (Cass04). Diese Kosten können sehr hoch sein, so dass aus finanzieller Sicht die reine automatische Codegenerierung uninteressant werden kann. Dieses Thema wird zur Zeit (2004) intensiv diskutiert. Zur teuren Zertifizierung und Qualifizierung gibt es gegenwärtig noch keine akzeptierte alternative Lösung. Ein Ausweg wäre, die Anforderungen im jeweiligen Anwendungsbereich zu überprüfen und – wenn möglich – zu verringern.

Aus strategischer Sicht und im Hinblick auf automatische Produktionsprozesse ist die weitere Zielsetzung aber klar vorgegeben: der Nachweis der Korrektheit muss für jeden Anwendungsfall und jede aktuelle Version der Prozessimplementierung ebenfalls wie die Qualitätssicherungsmaßnahmen des Produktionsprozesses durch einen generischen Ansatz erbracht werden können. Jedes Softwarepaket, das vom Produktionsprozess erzeugt wurde, soll von einem dazu passenden, automatisch erzeugten Qualitätssicherungsprozess überprüft werden können. Die ersten Arbeiten zu einem solchen

*These 111  
Der Nachweis  
der Korrektheit  
für den Produkti-  
onsprozess kann  
durch einen  
generischen  
Ansatz erbracht  
werden.*

Ansatz wurden von uns bereits im Rahmen des EU-Projektes "AS-SERT" begonnen.

## 1.2.16 Schnittstellenanpassung

Durch Benutzung von Fremdsoftware oder Wiederverwendung vorhandener Software stellt sich häufig das Problem der Integration dieser Software in eine vorgegebene Umgebung. Dafür müssen meistens Schnittstellen angepasst werden. Bei Fremdsoftware kann das Programm selbst nicht angepasst werden, und auch bei eigener Software ist häufig die beste Lösung, spezielle Software zwischen beide Teile einzufügen, die die Schnittstellen anpasst.

Ein solcher Transformator kann eine Zwischenfunktion sein, die für das rufende Programm die gewünschte Schnittstelle realisiert, die Daten anpasst und an die Zielfunktion weiterreicht. Er kann auch als eigenständiges Programm realisiert werden, um Daten von Datei zu Datei oder über Kanäle des Betriebssystems zu konvertieren und auszutauschen. Ebenso sind Mischformen möglich.

Für jedes der miteinander zu verbindenden Programme ist die Schnittstelle definiert, und – meistens – liegt diese Spezifikation bereits in maschinenlesbarer Form vor. Ebenso ist bekannt, von welchem Medium nach welchem Medium der Datenstrom zu empfangen und zu senden ist. Damit liegen – prinzipiell – alle Voraussetzungen vor, um das benötigte Bindeglied automatisch zu generieren.

Wenn die beiden Schnittstellen logisch zueinander kompatibel sind – im einfachsten Fall müssen nur Datenstrukturen umgeordnet werden, kann der Transformator ein breites Gebiet abdecken. Sind noch spezielle Operationen erforderlich, dann wird auch das Anwendungsgebiet des Transformators enger.

Der zugehörige Produktionsprozess muss als Parameter erhalten:

- die Definition der Datentypen und Ort der Speicherung,
- die Struktur der jeweiligen Schnittstelle in Form von Elementnamen und Datentyp,
- evtl. noch eine auszuführende Operation (s.a. Kap. 1.2.14), beispielsweise die Konvertierung von Binärdaten, und
- die Art der Schnittstelle wie Funktionsname, Datenkanal.

Eine einfache Anwendung ist beispielsweise, den Inhalt einer Datenstruktur über ein Netzwerk zu übertragen. Dazu muss beim Sender der Inhalt der Datenstruktur in einen Datenpuffer geschrieben werden, was nicht so aufwändig ist. Beim Empfänger müssen die ein-

*These 39  
Schnittstellen  
können automa-  
tisch angepasst  
werden.*



zelnen Elemente bytewise aus dem Datenpuffer geholt und den Elementen der Datenstruktur zugewiesen, und möglicherweise auch noch binär konvertiert werden. Bei größeren Datenstrukturen und vielen Schnittstellen, ist die binäre Konvertierung aufwändig und auch fehleranfällig, durch einen allgemeinen Transformator aber leicht zu erledigen, einschließlich der Dokumentation (s.a. Kap. 7.5.1).

### **1.2.17**

#### **Dokumentation**

Um einen Produktionsprozess oder eine Schnittstellenanpassung zu realisieren, muss die Fähigkeit vorhanden sein, Information auszuwerten, beispielsweise aus einer Datei, aus Quellcode oder aus einem Dokument. Diese Fähigkeit wollen wir als "Extraktion von Information" bezeichnen.

Wenn diese Fähigkeit verfügbar ist, kann sie genutzt werden, um die extrahierte Information in Dokumentation über die Informationsquelle zu transformieren. Diese Transformation kann auch eine Weiterverarbeitung der Information einschließen, wie beispielsweise die Erzeugung grafischer Darstellungen, Erstellen von Querverweisen usw.

Die Extraktion und Transformation ist dann Teil des Produktionsprozesses, und wie die anderen Schritte auch, auf ihn abgestimmt. Dies muss nicht implizieren, dass alle Teile nur für diesen Produktionsprozess implementiert werden müssen, sondern kann die Verwendung von Teilen existierender Produktionsprozesse einschließen.

## 1.3

### 111 Thesen zur effizienten Softwareentwicklung

Wir schließen dieses Kapitel mit der Aufzählung unserer Thesen ab, zu denen wir in den späteren Kapiteln Stellung nehmen werden.

#### 1.3.1

##### Stand der Technik

- These 1** Mit dem Stand der Technik können die prinzipiellen Probleme der Softwareentwicklung, hohe Fehlerrate und niedrige Produktivität, nicht gelöst werden.
- These 2** Der Stand der Technik sucht die Probleme der Softwareentwicklung hauptsächlich durch manuelle Ansätze zu lösen und nur punktuell durch Automation.

#### 1.3.2

##### Wettbewerb

- These 3** Wer als erster eine neue Technologie einsetzt, hat einen erheblichen Wettbewerbsvorteil.
- These 4** Wer automatisierte Softwareentwicklungsprozesse einsetzt, hat erhebliche Wettbewerbsvorteile.
- These 5** Die ständige Verbesserung der Effizienz sichert das Überleben im Wettbewerb.

#### 1.3.3

##### Softwareentwicklung

- These 6** Softwareentwicklung ist kein Selbstzweck, sondern eine Dienstleistung für den Benutzer.
- These 7** Effizienz und Qualität schließen sich nicht aus.
- These 8** Ziel von Test, Verifikation und Validierung ist es, Fehler nachzuweisen, und nicht, die Abwesenheit von Fehlern zu bestätigen, d.h. das Nichtauftreten von Fehlern wird erst einmal auf unzureichende Test- und Verifikationsmethoden zurückgeführt und nicht als Beweis für Fehlerfreiheit gewertet.

Nur durch einen wartbaren Entwicklungsprozess können gleichzeitig hohe Produktivität und Qualität erreicht werden. **These 9**

Es ist viel schwieriger, das gleiche Ziel mit niedriger Komplexität zu erreichen als mit hoher. Ein Ziel mit niedriger Komplexität zu erreichen, weist mehr Kompetenz nach **These 10**

### 1.3.4 Organisation

Das anzuwenden, was alle anwenden, führt nicht unbedingt zum Erfolg. **These 11**

Rationalisierungspotenzial kann nur durch detaillierte Betrachtung der Produktionsschritte erschlossen werden. **These 12**

Spärliche Fehlermeldungen implizieren erhöhten Aufwand beim Testen und zeigen damit erheblichen Mehraufwand an bzw. identifizieren ein Rationalisierungspotenzial. **These 13**

Die Softwareentwicklung muss so organisiert werden, dass sich Arbeitsabläufe wiederholen, um sie für die Automation vorzubereiten. **These 14**

Wenn Probleme in Teilprobleme aufgelöst werden, findet man die Lösung schneller. **These 15**

Alle Entwicklungsabläufe bzw. Entwicklungsphasen müssen auf Optimierungsmöglichkeiten analysiert werden, ebenso die Übergänge zwischen den Phasen. **These 16**

Wartung impliziert Fortsetzung der Entwicklung. **These 17**

Spezialisierung führt zu mehr Effizienz. **These 18**

Spezialisierung impliziert nicht die Beschränkung auf wenige Anwendungsfälle. Trotz Spezialisierung eines Produktionsprozesses kann eine unendliche Anzahl von Anwendungen abgedeckt werden. **These 19**

"Weniger ist mehr" – durch Beschränkung kann mehr erreicht werden. **These 20**

Nur kompetente strategische Entscheidungen zur Entwicklungsmethodik können die Situation verbessern, nicht aber der zufallsartige Einsatz von Methoden oder Zukauf von Produkten. **These 21**

Software selbst kann bei Vorgabe von Konstruktionsregeln und Anforderungen weitere für die Produktion von Software benötigte Programme, also wieder Software, erzeugen. **These 22**

- These 23** Eine geänderte Rollenverteilung zwischen Entwickler und Rechner kann die Effizienz wesentlich erhöhen.
- These 24** Die "Verstärkereigenschaften" eines Rechners können effizient für die Softwareentwicklung eingesetzt werden.
- These 25** Der kreative Entwickler soll nicht gezwungen werden, starre Regeln einzuhalten. Die Einhaltung von Regeln ist Aufgabe eines Rechners.
- These 26** Überforderung eines Entwicklers und daraus resultierende Konflikte und Fehler werden durch klare Trennung der Aufgaben entsprechend der Fähigkeiten von Entwickler und Rechner vermieden.
- These 27** Die Arbeitsteilung zwischen Mensch und Rechner muss gut organisiert werden.
- These 28** Synergie muss organisiert werden.
- These 29** Einfachheit kann organisiert werden.
- These 30** Gute Organisation verringert Komplexität, Aufwand und Entwicklungszeit.
- These 31** Bei der Organisation der Lösung muss der Nutzen für den Anwender im Vordergrund stehen.
- These 32** Produktionsprozesse können organisiert werden.
- These 33** Effizientes Vorgehen heißt: Einfache Fälle sofort lösen, andere bei Bedarf.
- These 34** Häufig anfallende Tätigkeiten zuerst automatisieren, andere bei Bedarf.
- These 35** Allgemeine Lösungen zuerst, spezielle bei Bedarf.

### 1.3.5 Schnittstellen

- These 36** Vollständige, abstrakte und klare Schnittstellen helfen Anwender und Entwickler. Eigene Schnittstellen schützen gegen Änderungen.
- These 37** Die Komplexität einer MMI kann mit einem Rechner reduziert werden.
- These 38** Bei guter Organisation muss der Anwender seine Welt, d.h. seine gewohnte Notation, nicht verlassen.
- These 39** Schnittstellen können automatisch angepasst werden.

### 1.3.6

#### Information

Abhängigkeiten von Programmteilen sollten klar dokumentiert sein. **These 40**

Inkompatibilitäten sollten automatisch erkannt werden.

Entwickler, die kurze und unverständliche Fehlermeldungen verwenden, führen ihre Tests manuell durch. **These 41**

Kurze Fehlermeldungen, die beispielsweise nur aus einer Fehlernummer bestehen, verursachen nicht nur dem Anwender Probleme, sondern behindern auch den Entwickler und erhöhen somit die Entwicklungskosten. **These 42**

Kurze Fehlermeldungen sind ein Zeichen für ineffiziente Organisation. **These 43**

### 1.3.7

#### Automation

Automation impliziert Spezialisierung. **These 44**

Durch Automation entsteht bei geeigneter Organisation eine Synergie zwischen Codegenerierung und Test, Verifikation und Validierung. **These 45**

Fehler in der Spezifikation dürfen sich nicht in den Produktionsprozess fortpflanzen. Der Produktionsprozess darf nur dann ausgeführt werden, wenn die Anforderungen fehlerfrei sind. **These 46**

Automation senkt das Entwicklungsrisiko. **These 47**

Bei richtig umgesetzter Automation gilt: einmal korrekt, immer korrekt. **These 48**

Automation senkt Kosten und Entwicklungszeit. **These 49**

Automation erhöht die Qualität. **These 50**

Ein Anwender muss voll auf einen automatischen Produktionsprozess vertrauen können. **These 51**

Ein automatischer Produktionsprozess muss einem Anwender die volle Kontrolle über die Erzeugung des Produktes ermöglichen. **These 52**

Ein automatischer Produktionsprozess muss dem Anwender die Ergebnisse in verständlicher Form präsentieren. **These 53**

- These 54** Automatische Produktionsprozesse müssen dem Anwender helfen, große Informationsmengen zu bewältigen.
- These 55** Ein automatischer Produktionsprozess muss die Informationsmenge reduzieren.
- These 56** Keine Eingabefehler → korrekter Code; bei Eingabefehlern → kein Code.
- These 57** Automation verringert die Komplexität für den Anwender, ohne die maximal erreichbare Komplexität zu beschränken.
- These 58** Automatische Codegenerierung hat nur ein Einsparungspotenzial von maximal ca. 20% des Gesamtaufwandes.
- These 59** Software kann Softwareproduktionsprozesse erzeugen.
- These 60** Softwareproduktionsprozesse decken Produktklassen ab.
- These 61** Konstruktionsregeln vervielfachen Information intelligent um Größenordnungen, ohne unnötigen Code zu erzeugen.

### 1.3.8 Strategie

- These 62** Das Ziel jeglicher Optimierung und Rationalisierung in der Softwareentwicklung muss die Minimierung des Aufwandes und die Erhöhung der Qualität sein.
- These 63** Ein besonders hohes Einsparungspotenzial kann dann erschlossen werden, wenn bereits vorhandene Information automatisch extrahiert und in Ergebnisse umgesetzt werden kann.
- These 64** Bei geeigneter Organisation können mit endlichem Aufwand unendlich viele Anwendungsfälle, viele Operationen mit einmaligem, endlichem Aufwand abgedeckt werden.
- These 65** Zur Erschließung des maximalen Rationalisierungspotenzials muss ein Produktionsprozess auf Konzepten aufgebaut werden, die minimalen Aufwand erfordern.
- These 66** Nicht die aktuelle, bereits vorhandene Organisationsform darf die Grundlage der Prozessoptimierung sein, sondern die Organisation, die den Aufwand tatsächlich minimiert (totales Minimum).
- These 67** Um Erfolg messen zu können, braucht man Metriken und muss kontinuierlich Kontrollen (Benchmarking) durchführen. Nur wiederholbare und messbare Arbeitsabläufe können optimiert werden.
- These 68** Unkonventionelle Lösungsansätze sichern den Erfolg.

Einfachheit, nicht Komplexität impliziert den Erfolg.	<b>These 69</b>
Nur durch geeignete strategische Entscheidungen können wesentliche Verbesserungen erreicht werden.	<b>These 70</b>
Das Setzen eines Komplexitätslimits erzwingt einfache Lösungen.	<b>These 71</b>
Je früher ein Fehler erkannt wird, desto mehr spart man.	<b>These 72</b>
Fehlerprävention in frühen Phasen, senkt die Kosten der späteren Phasen.	<b>These 73</b>
Frühzeitige Validierung des Entwurfs zahlt sich aus.	<b>These 74</b>
Einsparungen im Bereich 85% .. 95% sind durch Automation möglich.	<b>These 75</b>

### 1.3.9 Kostenschätzung

Kostenreduktion impliziert Reduktion der Funktionalität.	<b>These 76</b>
Kostenreduktion durch Suche nur nach zu großzügigen Schätzungen führt zu späteren Verlusten. Das Gleichgewicht von unterschätzten und überschätzten Problemen wird dadurch in Richtung höheres Risiko verschoben.	<b>These 77</b>
Fehlerhafte Kostenschätzungen können durch Inkonsistenzen zwischen Schätzungen verschiedener Ausprägung, aber gleicher Wurzel entdeckt werden, beispielsweise durch Vergleich von Aufwandschätzung und (umgerechneten) technischen Budgets wie solche über Speicherbedarf.	<b>These 78</b>

### 1.3.10 Projektmanagement

Die Anzahl der Fehler pro Entwicklungsphase steigt von der Spezifikation bis zur Codierung kontinuierlich an – bedingt durch die Menge der zu bearbeitenden Information.	<b>These 79</b>
Fehler müssen vor der Auslieferung erkannt werden.	<b>These 80</b>
Die meiste Zeit und der meiste Aufwand werden für die Fehlerlokalisierung benötigt, bis zu ca. 95%.	<b>These 81</b>

- These 82** Bei frühzeitiger Validierung von Spezifikation und Entwurf lassen sich hohe Einsparungen erzielen und das Risiko erheblich senken.
- These 83** Bei spät erkannten Fehlern müssen alle früheren Aktivitäten erneut ausgeführt werden.
- These 84** Durch automatische Übergänge entfallen sonst notwendige manuelle Prüfungen auf Konsistenz der Ergebnisse, entweder ist die Konsistenz durch die Automation inhärent gewährleistet, oder sie werden durch automatische Prüfungen ersetzt.
- These 85** Ein Mitarbeiter, der bereits ein Problem hatte, kann zukünftige Probleme besser vermeiden (wenn er lernfähig ist) als ein Mitarbeiter, der noch kein Problem hatte. Erfahrung, insbesondere negative, kann zu zukünftigen Verbesserungen führen.
- These 86** Problemlösung setzt den Willen zur Diskussion und Offenheit voraus.
- These 87** Durch Vorgaben des Auftraggebers können beim Auftragnehmer Risiken und Mehraufwand entstehen.

### 1.3.11 Management

- These 88** Erfolgreiche Umsetzung von Veränderungen erfordert psychologisches Geschick und Durchsetzungsvermögen.
- These 89** Das Management sollte im Dialog mit den Entwicklern den Herstellungsprozess optimieren.
- These 90** Das Management sollte nicht ungeprüft organisatorische Vorschläge der Entwickler übernehmen.
- These 91** Im Management muss die Kompetenz vorhanden sein, technische Lösungen und Maßnahmen hinsichtlich Rationalisierung zu beurteilen bzw. vorzuschlagen.
- These 92** Das richtige Problemverständnis führt zum Erfolg.

### 1.3.12 Qualitätssicherung

- These 93** Vollständiges Testen ist wegen der großen, meist unendlichen Anzahl von Testfällen praktisch unmöglich. Je geringer der Aufwand pro Testfall, desto mehr Testfälle können abgedeckt werden. Ausreichende Testabdeckung kann nur durch Automation erzielt werden.



Je mehr Fehler erkannt wurden, desto besser die Qualität.	<b>These 94</b>
Aus der Absicht, das Risiko durch Vereinfachung zu senken, entsteht möglicherweise ein viel größeres Risiko.	<b>These 95</b>
Fehlende Information über die Korrektheit einer Spezifikation erhöht das Risiko.	<b>These 96</b>
Wenn man die erlaubten Fälle kennt, kennt man auch die Fehlerfälle.	<b>These 97</b>
Für jede auszuführende Aktion muss auch der Fehlerfall betrachtet werden.	<b>These 98</b>
Portabilität erhöht die Qualität. Jede Plattform wirkt als Filter für bestimmte Fehler, sie filtert einige aus, andere können (prinzipiell) nicht erkannt werden. Durch Einsatz verschiedener ("gekreuzter") Filter können Fehler nicht durch das Gesamtfiler "durchfallen".	<b>These 99</b>
Je mehr Aufwand erforderlich ist, um einen Fehler zu finden, desto weniger Fehler sind in der Software noch enthalten.	<b>These 100</b>
Tritt während der Tests kein Fehler auf, impliziert das keine Fehlerfreiheit.	<b>These 101</b>
Die Anzahl der tatsächlich ausgeführten, verschiedenen unabhängigen Testfälle hängt nicht von der Ausführungs- bzw. Betriebszeit ab.	<b>These 102</b>
Wenn nur bestimmte Testszenarien benutzt werden, kann man prinzipiell nicht alle Fehler finden.	<b>These 103</b>
Die bessere Umsetzung von Standards durch mehr, insbesondere durch qualifizierteres Personal, liegt nahe, ist aber nicht zwingend.	<b>These 104</b>
Die Anwendung von Standards führt nicht zwingend zu fehlerfreier Software. Standards dienen der Beschränkung der vollständigen kreativen Freiheit auf bewährte Ansätze. Aber die menschliche Unzulänglichkeit, Regeln nicht immer folgen zu können, auch wenn der Wille vorhanden ist, führt trotzdem zu Fehlern, und zwar immer mit einer endlichen (persönlichen) Fehlerwahrscheinlichkeit, auch nach bestem Training.	<b>These 105</b>
Qualitätssicherungsmaßnahmen wie die Anwendung von Standards, deren Einhaltung nicht kontrolliert werden kann, führen nicht zwingend zu mehr Qualität.	<b>These 106</b>
Zertifizierung ist keine Garantie für Fehlerfreiheit.	<b>These 107</b>
Bei geeigneter Organisation kann ein Rechner nicht nur Software produzieren, sondern auch die Qualität selbst überwachen.	<b>These 108</b>

- These 109** Die Synergie zwischen Erzeugung und Überwachung reduziert die Fehler.
- These 110** Je größer die Bandbreite eines Produktionsprozesses, desto größer seine Zuverlässigkeit.
- These 111** Der Nachweis der Korrektheit für den Produktionsprozess kann durch einen generischen Ansatz erbracht werden.

111 Thesen zur erfolgreichen Softwareentwicklung

Argumente und Entscheidungshilfen für Manager.

Konzepte und Anleitungen für Praktiker

Gerlich, R.

2005, XXII, 522 S., Hardcover

ISBN: 978-3-540-20910-2