Richtlinien für unwartbaren Code

© Thomas Birnthaler (tb@ostc.de)
OSTC GmbH (www.ostc.de)

14.6.2006 — V1.16 [badcode.txt]

Inhaltsverzeichnis

1 Einführung

Folgende Richtlinien wurden gesammelt, um den **Personal- und Kostenbedarf** in der Software-Industrie so hoch wie möglich zu halten. Mit ihrer Hilfe wird die erstellte Software so **schwierig zu warten**, dass selbst für die einfachsten Änderungen Jahre benötigt werden. Weiterhin wird die Software dadurch so **teuer**, dass keiner mehr die Wichtigkeit der Projekte in Frage zu stellen wagt und die erstellte Software niemals weggeworfen wird, da sie zu wertvoll ist.

Durch das gewissenhafte Befolgen dieser Regeln werden die **Arbeitsplätze** aller Beteiligten auf Lebenszeit gesichert, da keiner außer ihnen selbst jemals die geringste Chance hat, die Software zu verstehen und zu warten. Als Nebeneffekt vermeidet man auf diese Weise auch, sich immer wieder in neue Projekte einarbeiten zu müssen, da man früher oder später mit der Wartung der Altprojekte genügend ausgelastet ist.

2 Richtlinien für Manager

Mit Hilfe folgender Techniken schaffen Sie die **optimalen Rahmenbedingungen** für maximalen Aufwand und minimalen Erfolg Ihrer Software-Projekte. Halten Sie sich an diese Richtlinien und kaum ein Entwickler kann auf der Ebene der **Konzeption und Codierung** mehr etwas dagegen ausrichten. Aller Wahrscheinlichkeit nach werden die Entwickler sich eine Weile wehren, irgendwann aber aufgeben und sich dann darauf beschränken, auf ihrer eigenen Ebene einen entsprechenden Beitrag zur Misere zu leisten. Denken Sie immer daran:

- Je größer und teurer ein Projekt ist, desto höher ist Ihr Ansehen als Manager.
- Bei Problemen mit einem Projekt können Sie immer auf die Entwickler verweisen, die seine Komplexität nicht im Griff haben.
- Weisen Sie immer darauf hin, dass ohne Sie alles noch viel schlimmer wäre, da Sie als einziger das Chaos noch überblicken.

2.1 Allgemeine Techniken

1. Verzichten Sie auf einen klar definierten **Projektstart** und ein klar definiertes **Projektende**. Dadurch lösen Sie gleichzeitig auf einen Schlag die beiden Themen Motivation und Finanzierung.
2. Machen Sie keine **Kosten**/**Nutzen-Analyse** für geplante Projekte (falls diese dennoch nötig sein sollte, lassen Sie das die Entwickler machen, die haben dafür Zeit).
3. Legen Sie zu jedem Projekt immer zuerst den **Abschlußtermin** fest (am besten „gestern“), dann das **Budget** und lassen Sie dann sofort eine **Aufwandsschätzung** machen (wenn möglich innerhalb eines Arbeitstages). Legen Sie die ersten 50% der **Vorgaben** danach im Laufe von 1-6 Monaten fest.

4. Legen Sie die zweiten 50% der Vorgaben erst im Rahmen des **Produktionseinsatzes** fest, auch wenn sich regelmäßig herausstellt, dass diese 50% etwa 90% des Realisierungsaufwandes erfordern.
5. Definieren Sie keine Nachfolgeprojekte, sondern **Dauerprojekte**.
6. Fragen Sie nicht zuviel und legen Sie nicht zuviel fest, das stört nur die **Projekteuphorie**.
7. Nehmen Sie nur zur Kenntnis, was Ihnen gefällt, ignorieren Sie insbesondere **Interessenskonflikte**.
8. **Besprechungen**:
   - ▷ Erstellen Sie **keine Agenda** oder verschicken Sie diese frühestens 1h vorher, damit
   - ▷ sich keiner **vorbereiten** kann und
   - ▷ offensichtlich wird, dass keiner regelmäßig in sein Mail-System schaut.
   - ▷ Beanspruchen Sie in 50% aller Fälle selbst das „**akademische Viertel**“,
   - ▷ bestehen Sie in den anderen 50 % der Fälle darauf, dass **pünktlich** begonnen wird,
   - ▷ laden Sie möglichst **viele Mitarbeiter** ein,
   - ▷ besprechen Sie möglichst **viele verschiedene Themen und Projekte** gleichzeitig und durcheinander.
   - ▷ Setzen Sie als **minimale Dauer 3-4h** an (noch besser einen ganzen Arbeitstag),
   - ▷ nehmen Sie während der Besprechung jedes **Telefongespräch** an (damit die Teilnehmer merken, wieviel wichtiger Sie sind als sie),
   - ▷ erstellen Sie **kein Protokoll** (oder ein vollkommen subjektives), damit Sie
   - ▷ bei der nächsten Besprechung die **gleichen Punkte** wieder von vorne aufrollen, aber
   - ▷ zu ganz **anderen Ergebnissen** kommen können.
9. Verlangen Sie trotz der fehlenden Vorgaben eine **exakte Aufwandsschätzung** (seien Sie großzügig und definieren Sie „exakt“ als $\pm 10\%$). Wundern Sie sich nicht, wenn das aktuelle Datum, die Lottozahlen oder der Kontostand eines Entwicklers als Ausgangsbasis für diese Aufwandsschätzung dienen.
10. Lassen Sie mindestens **3 Hierarchieebenen** die Aufwandsschätzungen nach eigenen Interessen und Gutdünken manipulieren.
11. Ändern Sie ständig die **Prioritäten** der Projekte (dies fördert die Flexibilität der Entwickler).
12. Setzen Sie jeden Entwickler gleichzeitig in mindestens **2 Projekten** ein (dies fördert die Fähigkeit zum „Task-Switching“ und zur Kommunikation).
13. Ziehen Sie Entwickler für ein anderes Projekt ab, sobald sich diese einigermaßen in ein Projekt eingearbeitet haben (dies fördert ihre Flexibilität).

14. Sparen Sie sich **Anwendertests** (z.B. indem die Anwender keine Zeit dafür erhalten), die Anwender wollen sowieso lieber ihr altes System behalten.

15. Sorgen Sie dafür, dass aufgrund der Terminsituation immer die Formel „**Anwendertest = Produktionseinsatz**“ gilt.

16. Planen Sie eine **zentral eingesetzte Software** für alle Firmenteile. Sorgen Sie dafür, dass keine Einigung über die **gemeinsamen Anforderungen** erfolgt oder dass diese **nicht schriftlich** beschrieben werden.

17. Lassen sie die Entwickler mit allen Anwendern gleichzeitig kommunizieren, statt ein **Projektkoordination auf Anwenderseite** durchzuführen (dies fördert die Kommunikation).

18. Lassen Sie den Personenkreis, der die **Anforderungen** festlegt, und den Personenkreis, der die **Budgets** kontrolliert, nur indirekt über die Software-Entwicklung miteinander kommunizieren (dies fördert die Kommunikation). Auf diese Weise können Sie vermeiden, dass Budgetbeschränkungen zu Einschränkungen bei der Funktionalität der Software führen.

19. Führen sie weder eine kurz-, noch eine mittel- oder gar eine langfristige **Planung** für die Wartung, Pflege und Weiterentwicklung der im Einsatz befindlichen Software durch.

20. Lassen Sie sich **vom Kunden treiben**; wenn er nach Hilfe ruft, sichern Sie ihm diese sofort (mündlich) zu. Nennen Sie ihm einen beliebigen Entwickler als Ansprechpartner für das Problem (ohne diesen oder seine Vorgesetzten davon zu informieren).

21. Definieren Sie nicht, was genau unter **Manntag**, **Mannmonat** oder **Mannjahr** zu verstehen ist. Gehen Sie immer davon aus, dass Entwickler jahrelang 10h am Tag an 7 Tagen in der Woche mit voller Kraft arbeiten können, da sie keine Familie besitzen und ihr Hobby zu ihrem Beruf gemacht haben.

22. Üben Sie möglichst viel **Druck** aus, wenn ein Projekt nicht rechtzeitig fertig zu werden scheint. Schreiben Sie sich den Erfolg auf Ihre Fahnen, wenn die Entwickler es unter Mobilisierung aller Ressourcen wider Erwarten doch noch rechtzeitig schaffen.

23. Verzichten Sie auf **Motivation** der Entwickler. Dass diese den ganzen Tag mit toller Hard- und Software umgehen dürfen, ist Motivation genug.

24. Verlangen Sie mehrsprachige **Oberflächen** hoher Komplexität mit Undo-Funktionaliät, kontextsensitiver Hilfe, erstklassigen Handbüchern in mehreren Sprachen, ... Gehen Sie dabei von dem inzwischen durch Microsoft-Word oder -Excel erreichten Level aus. Ignorieren Sie, dass diese Programme (ohne die Vorläufer anderer Software-Hersteller) inzwischen etwa die 10. Version erreicht haben, von 100en von Entwicklern entwickelt, von 100en von technischen Redakteuren dokumentiert und übersetzt, von 1000en von Testern alpha- und von 10000en von Anwendern beta-getestet werden (und immer noch beschämend viele Fehler enthalten).

25. Trauern Sie den guten alten Zeiten nach, in denen die Hardware-Kosten so hoch waren, dass die Software-Kosten dagegen vernachlässigbar waren. Rechnen Sie nicht nach, was die Software damals wirklich gekostet hat.

26. Versprechen Sie regelmäßig **Änderungen der Vorgehensweise**, um die Mitarbeiter bei Laune zu halten (aber vergessen Sie nie den Zusatz: „Aber dieses Mal noch nach dem alten Schema aufgrund der Terminsituation“).

27. Bringen Sie mindestens alle halbe Jahre das Thema „**Können wir unsere Software nicht in Indien entwickeln lassen**“ (alternativ **Polen**, **Ukraine**, **Russland**, **China**) zur Sprache. Dies motiviert Ihre Mitarbeiter zu Höchstleistungen.

2.2 Spezielle Techniken

1. Lassen sie **Aufwandsschätzungen** wie auf dem Bazar mehrfach wiederholen, bis auf Stundenebene in Einzelteile zerlegen und von selbsternannten „Experten“ (die um 1950 herum ein einziges Programm geschrieben und nie ein Programm gewartet haben) einschätzen, um die Aufwände und damit die Projektkosten zu senken (die Anforderungen können nicht gesenkt werden, da sie nicht bekannt sind).

2. **Hören Sie nicht hin**, wenn zu einem Aufwand von N Manntagen gesagt wird, dass dafür nur ein einziger Entwickler ohne Einarbeitungsaufwand verfügbar ist, der aber bereits durch 3 andere Projekte überlastet ist.

3. Behalten Sie eine einmal mündlich am Telefon für ein Projekt von einem Entwickler erhaltene „**Hausnummer**“ für den Aufwand auch nach Jahren noch wie in Stein gemeißelt im Gedächtnis, und ignorieren Sie, dass sich mittlerweile sämtliche Projekt-Voraussetzungen geändert haben.

4. Üben Sie mindestens einmal in der Woche **direkten Durchgriff** auf einzelne Entwickler aus. Beschweren Sie sich sofort, wenn dieser Durchgriff andere Projekte behindert oder die Kapazitäts-Planung nicht sofort aktualisiert wird.

5. Lassen Sie die Entwickler die **Testdaten** selbst auswählen und zusammenstellen (da die Anwender keine Zeit dafür haben, einen vollständigen Testdatensatz zusammenzustellen). Alternative: Lassen Sie gar keine Testdaten zusammenstellen.

6. Verzichten Sie auf eine **Abteilung zur Qualitäts-Sicherung**, zum Test und zur Abnahme von Programmen. Lassen Sie statt dessen die Anwender den Alpha-Test einer neuen Software machen. So sorgen Sie für möglichst viel „Reibung“ zwischen den Software-Entwicklern und den Anwendern (dies fördert die Kommunikation).

7. Lassen Sie für ein neues Problem immer erst nach einem „**Workaround**“ mit einem Aufwand von 1 bis 2 Manntagen suchen. Drohen Sie gegebenenfalls mit Kundenverlust, falls die Entwickler murren. Fragen Sie auf keinen Fall nach dem Aufwand, der nötig ist, um die Sache (anschließend) wirklich geradezubiegen.

8. Ignorieren Sie, dass jeder Workaround mindestens 2 ernsthafte Probleme innerhalb eines Monats und damit mindestens 2 weitere Workarounds zur Folge hat (sogenanntes „**exponentielles Wachstum**“). Wichtig ist nur, dass der gerade akute Termin gehalten wird.

9. Lassen Sie mindestens 3 verschiedene **Versionen** eines Programms gleichzeitig in der Produktion laufen (z.B. aufgrund fehlender Anwendertests, fehlender Abnahmen, Workarounds, Speziallösungen, fehlender Wartungsverträge, Sonderwünsche, . . .).
10. Setzen Sie alle am Projekt Beteiligten so lange unter **Kostendruck**, bis nur noch über die Kosten diskutiert wird. Dann kann die Definition der Anforderungen noch etwas weiter hinausgeschoben werden.
11. Setzen Sie die Entwickler für die Durchsetzung von **Abrechnungen** und das Eintreiben von Kosten ein (Manager und Controller wären dafür zwar ausgebildet, stecken aber inhaltlich zuwenig in der Materie drin).
12. Investieren Sie möglichst wenig in die **Ausbildung der Entwickler**, sie werden irgendwann sowieso die Firma verlassen. Vernachlässigen Sie, dass Wissen, Methoden und Werkzeuge in der Informations-Technik alle 2-5 Jahre veralten (Hardware teilweise noch schneller).
13. Investieren Sie noch weniger in die **Ausbildung der Anwender**, die mit der erstellten Software arbeiten müssen. Sämtliche modernen Oberflächen sind so beschaffen, dass sich die Software von selbst erklärt, wenn man nur lange genug in den Menüs und Dialogboxen „herumklickt“.
14. Setzen Sie die Anwender solange unter Druck, bis diese sich gar nicht mehr trauen, eine **Schulung** zu verlangen, damit sie ordentlich arbeiten können.

3 Richtlinien für Entwickler

Sie können davon ausgehen, dass die Manager mit Hilfe obiger Richtlinien ihren Teil zur Software-Misere beitragen, ohne dass Sie selbst irgend etwas dagegen unternehmen können, da auf Sie (trotz Ihrer spezifischen Ausbildung) sowieso keiner hört. Beschränken Sie sich aber nicht auf die passive, leidende Rolle, sondern wehren Sie sich und werden Sie ebenfalls aktiv tätig. Indem Sie folgende Richtlinien berücksichtigen, werden Sie mindestens genauso effektiv zur Misere beitragen und müssen die Lorbeeren dafür nicht den Managern alleine überlassen. Denken Sie immer daran:

- Je größer und komplizierter ein Programm ist, desto höher ist Ihr Ansehen als Entwickler.
- Bei Problemen können Sie immer auf die Manager verweisen, die das Projekt nicht im Griff haben.
- Weisen Sie immer darauf hin, dass ohne Sie alles noch viel schlimmer wäre, da Sie als einziger das Chaos noch überblicken.

3.1 Grundprinzip

Um einem anderen Entwickler das Leben so schwer wie möglich zu machen, muß man verstehen, wie er denkt: Er steht vor Ihrem **riesigen Programm**. Er hat keine Zeit, es voll-

ständig zu lesen, geschweige denn, es überhaupt zu verstehen. Er will schnell die Stellen finden, an denen er seine Änderungen machen muß, und er will mit diesen Änderungen keine unerwarteten **Seiteneffekte** auslösen.

Er sieht Ihren Code durch das Innere einer Toilettenpapierrolle an. Er sieht immer nur ein winziges Stück Ihres Programmes auf einmal. Sie müssen sicherstellen, dass er nie einen **vollständigen Überblick** über Ihr Programm erhalten kann. Sie müssen es ihm möglichst schwer machen, den Code zu finden, den er ändern will. Aber noch wichtiger, Sie müssen dafür sorgen, dass er auf keinen Fall irgend etwas ignorieren darf, wenn er sichergehen will, keine unerwarteten Seiteneffekte hervorzurufen.

3.2 Allgemeine Techniken

1. Fragen Sie nicht lange, sondern versuchen Sie die **Anforderungen** der Anwender zu erraten. (1) sind die Anwender sowieso nicht in der Lage, ihr Problem zu beschreiben, (2) verstehen Sie selbst nicht, was die Anwender wollen, und (3) verstehen die Anwender Ihre „Techno-Speak“ ebenfalls nicht.

2. Wenn es wirklich unbedingt sein muß, interviewen Sie zur Ermittlung der **Anforderungen** auf keinen Fall die wirklichen, zukünftigen Anwender der Software, sondern deren Manager (weil die Anwender keine Zeit haben). Auf diese Weise erhalten Sie ein kurzes und unverbindliches Lastenheft, das vor allem die Budgetplanung nicht überschreitet. Kümmern Sie sich nicht darum, ob damit wirklich die Anwenderwünsche abgedeckt sind.

3. Beginnen Sie sofort mit der **Codierung**, halten Sie sich nicht mit Pflichtenhefterstellung, Konzeption und Entwurf auf (bezahlt werden diese Tätigkeiten sowieso nicht). So sparen sie Zeit für das Debuggen der unvermeidlichen Fehler, die der Anwender findet

4. Halten Sie sich nicht mit der **Programm-Dokumentation** auf (bezahlt wird diese Tätigkeit sowieso nicht). Erst beim Erraten von „was war hier gemeint“ zeigen sich die wahren Meister der Programmierkunst

5. Verfassen Sie keine **Anwender-Handbücher**, sie werden sowieso nicht gelesen. Eine **Schulung** sollte grundsätzlich für jeden Anwender einzeln durch den Entwickler erfolgen und bei Problemen mit dem Programm sollte der Anwender am besten direkt der Entwickler anrufen (er wartet den ganzen Tag auf solche Anrufe).

6. Warum einfach, wenn es auch **kompliziert** geht? Verwenden Sie immer den kompliziertesten, umständlichsten, längsten und am wenigsten nachvollziehbaren Algorithmus, Literaturhinweise sind auf jeden Fall überflüssig.

7. Es ist absolut unprofessionell, **Code** zu schreiben, den andere Entwickler auf Anhieb verstehen können. Der Haufen Geld, den Sie verdienen, ist doch nur aufgrund der hohen Komplexität der von Ihnen erstellten Software überhaupt zu rechtfertigen.

8. Ihr **Code** sollte beeindruckend sein. Wenn Ihre Ausdrucksweise und Ihr Vokabular unverständlich sind, nehmen die Leute an, dass Sie sehr intelligent und Ihre Algorithmen sehr tiefgründig sind. Vermeiden Sie schlichte Analogien in den Erklärungen zu Ihren Algorithmen.

9. Verwenden Sie keine Mühe darauf, die **Namen** von Datenbanken (Tabellen, Spalten), Datenstrukturen (Klassen, Objekten), Funktionen (Methoden), Variablen (Feldern) und Konstanten gut und verständlich zu wählen, denn „Namen sind Schall und Rauch“. Achten Sie statt dessen darauf, dass sie möglichst kurz und einfach zu tippen sind (gute Namen sind z.B. `dummy`, `tmp`, `help`, `a`–`z`, `xxx`, `var1`–`varN`, ...).

10. Ignorieren Sie die **Konventionen** für die Groß/Kleinschreibung der Namen von Datenbanken (Tabellen, Spalten), Datenstrukturen (Klassen, Objekten), Funktionen (Methoden), Variablen (Feldern) und Konstanten (die Ersteller der Betriebssystem- und Compiler-Bibliotheken machen dies ja schließlich auch).

11. Vermeiden sie **konsistente Benennungsschemata**, sodass der Zusammenhang zwischen zusammengehörenden Datenbanken (Tabellen, Spalten), Datenstrukturen (Klassen, Objekten), Funktionen (Methoden), Variablen (Feldern) und Konstanten auf keinen Fall offensichtlich wird.

12. Legen Sie auf keinen Fall ein **Glossar** (Begriffslexikon) an, in dem Ihr spezielles Projektvokabular eindeutig definiert wird. Dies wäre eine unprofessionelle Verletzung des Prinzips des „**Information Hiding**“.

13. Führen Sie kein **Logbuch** über die von Ihnen gemachten (Denk)Fehler, es wird sowieso zu lang und damit zu unübersichtlich (am Ende könnten Ihnen oder Ihrem Chef vielleicht noch Zweifel an Ihrer Berufseignung kommen).

14. Verwenden Sie kein **Source-Code-Version-Control-System**, um verschiedene **Versionen** Ihres Codes aufbewahren und die Änderungen zwischen diesen Versionen wieder rekonstruieren zu können.

15. **Codieren** sie dieselbe Funktionalität möglichst oft, die Komplexität von Code wird schließlich immer noch in Anzahl Zeilen gemessen. Programmieren Sie insbesondere Bibliotheksfunktionen oder von Kollegen erstellte Funktionen grundsätzlich nach (mit an Sicherheit grenzender Wahrscheinlichkeit sind diese nicht effizient implementiert, das können Sie auf jeden Fall besser).

16. **Kopieren** Sie Code, den Sie leicht verändert erneut brauchen, um Ihre Effizienz zu steigern („**copy**/**paste**/**clone-Prinzip“**). Sie sind so viel schneller, als wenn Sie mühsam parametrisierte, mehrfach wiederverwendbare Funktionen erstellen.

17. Noch effektiver ist es, den Code ganzer Programme wegen einer Spezialversion für einen Kunden zu **kopieren** (der Wartungsaufwand wird auf diese Weise mit einem einzigen Schlag verdoppelt, mehr können Sie im Namen der Arbeitsplatzsicherung auch mit anderen Methoden kaum erreichen).

18. Code einzutippen ist derart mühevoll, dass Sie nicht auf das **Aussehen** und konsistentes + korrektes **Einrücken** achten sollten. Sie sind (1) keine Sekretärin und (2) kommt es auf den Inhalt an, nicht auf die Form. Überlassen Sie das Ihrem weniger qualifizierten Nachfolger, der beim Reformatieren Ihren Code gleich viel besser kennenlernen kann.

19. Verwenden Sie nie **Tools**, um Code automatisch gemäß einem Standard einzurücken. Kreieren Sie Ihren eigenen persönlichen **Einrückungsstil**, der sich von dem aller anderer Entwicklern unterscheidet. Kombinieren Sie verschiedene Einrückungsstile, am besten in jeder Zeile einen neuen (berufen Sie sich auf die schriftstellerische/künstlerische Freiheit, wenn hier Einschränkungen drohen).

20. **Fehlerbehandlung** und Zusicherungen (`assign`s) sind nur etwas für andere Entwickler (sogenannte „Stümper“), gehen Sie grundsätzlich davon aus, dass Ihr Code perfekt ist und korrekt läuft. Wenn Fehlerbehandlung überhaupt sein muß, reduzieren Sie die Komplexität soweit wie möglich und brechen Sie bei Problemen das Programm sofort und einheitlich mit dem Exit-Code 1 ab.

21. Verzichten Sie auf Fortlauf-, Warnungs-, Fehler-, Usage- und einschaltbare Debug-**Meldungen**. Aufgrund Ihres perfekten Codes sind diese nicht notwendig und kosten nur Performance.

22. Vermeiden Sie **daten/tabellengesteuerte Logiken**. Es könnte ja sein, dass auf diese Weise Ihre Anwender den Code verstehen, ihn eventuell sogar Korrektur lesen und im allerschlimmsten Fall sogar verändern können.

23. Ändern und „Verbessern“ Sie Ihren Code möglichst oft, insbesondere die **Schnittstellen** Ihrer Funktionen.

24. Dokumentieren Sie die **Änderungen** zwischen verschiedenen Programmversionen nicht (wozu Anwender oder Ihren Chef auf behobene Fehler hinweisen, wenn diese sie eventuell noch gar nicht entdeckt haben).

25. Je mehr triviale Änderungen Sie zwischen zwei **Versionen** durchführen, um so besser. Auf diese Art und Weise werden Ihre Anwender z.B. nicht mit der immer gleichen Oberfläche, der gleichen Reihenfolge der Eingaben oder dem gleichen Ausgabeformat gelangweilt. Teilen Sie die Änderungen nicht mit, das finden die Anwender viel lieber selbst heraus.

26. Beschreiben Sie nicht die **Primär-** und **Sekundärschlüssel** von Datenbanktabellen sowie die Zusammenhänge zwischen den Tabellen (Fremdschlüssel).

27. Kümmern Sie sich nicht um **Indices**, moderne Datenbanken sind selbstoptimierend und legen die benötigten Indices selbständig an.

28. Führen Sie keine **Performance-Messungen** Ihrer Applikation oder der Datenbank-Zugriffe durch (denn beim 1. Einsatz der Software in 2 Jahren ist die verbesserte Hardware auf jeden Fall schnell genug dafür geworden).

29. Halten Sie die gleichen **Daten an mehreren Stellen** und sorgen Sie dafür, dass keine automatische Synchronisation stattfindet. Sagen Sie, dass dies der Datensicherheit dient, da die Anwender so bei Bedienungsfehlern noch ihre Originaldaten wiederherstellen können. Besonders effektiv ist es, die Daten gleichzeitig auf hard- und softwaretechnisch inkompatiblen Systemen (z.B. HOST und UNIX) zu halten.

30. Schreiben Sie Ihre Programme so, dass diese möglichst umständlich zu **installieren** sind (z.B. durch eine Mischung aus INI-Dateien, Datenbank-Tabellen mit Default-Einträgen, Umgebungsvariablen, Registry-Einträgen, dynamisch hinzuzulinkenden Bibliotheken, Programm ruft Programm auf, ...).

3.3 Spezielle Techniken

1. Noch besser als überhaupt nicht zu kommentieren ist es, zu „vergessen" die **Kommentare** und den Code auf dem gleichen Stand zu halten.

2. **Kommentieren** Sie einfache Dinge wie z.B. `/* add 1 to i */` ausführlich, aber nie komplexe Dinge wie z.B. den Sinn und Zweck einer Funktion oder einer Bibliothek.

3. Verstehen Sie selbst Ihren Quellcode erst nach **mehrmaligem sorgfältigem Durchlesen** und nur durch viel erklärenden Kommentar außenherum, dann haben Sie das Optimum an Programm-Komplexität erreicht und sollten den Code nicht mehr ändern (aber durchaus den Kommentar entfernen).

4. Sorgen Sie dafür, dass jede **Funktion** ein bißchen mehr (oder etwas weniger) macht, als ihr Name vermuten läßt (z.B. sollte ein Funktion namens `isValid(x)` als Seiteneffekt den Wert `x` in einer Datenbank abspeichern).

5. Verwenden sie **Akronyme** (Abkürzungen) um den Code knapp zu halten. Wahre Entwickler definieren Akronyme nie, sie haben ein angeborenes Verständnis dafür.

6. Vermeiden Sie **Kapselung** um die Effizienz zu steigern. Die Verwender einer Funktion brauchen aufgrund der fehlenden Dokumentation sowieso soviele Hinweise wie möglich, um zu wissen, wie die Funktion intern realisiert ist.

7. Schreiben Sie Ihre Software **nicht änderungsfreundlich**, damit jede kleine Änderungen einen hohen Aufwand erzeugt (Arbeitsplatzsicherung). Wenn Sie z.B. ein Flugreservierungssystem codieren, sorgen Sie dafür, dass zur Aufnahme einer weiteren Fluglinie Änderungen an mindestens 25 Stellen notwendig sind. Dokumentieren Sie diese Stellen nicht.

8. Kommentieren Sie niemals irgendwelche **Variablen**. Wie eine Variable verwendet wird, ihre Grenzwerte, die für sie erlaubten Werte, ihre Anzahl an Dezimalstellen, ihre physikalische Einheit, ihr Anzeigeformat, ihre Eingaberegeln (vollständig auszufüllen, Muß/Kannwert), wann ihr Wert gültig ist, ... sollte nicht im Code beschrieben werden.

9. Versuchen Sie soviele Anweisungen wie möglich in **eine Zeile** zu packen (Tipp: Entfernen Sie Leerzeilen und alle Leerzeichen um Operatoren und Zuweisungen). Sie sparen dadurch zusätzlich mindestens 10% Plattenplatz ein.

10. **Cd hn Vkle st vl krzr** (das ist lesbar, wenn man die Vokale ergänzt). Verwenden Sie möglichst viele verschiedene Abkürzungen des gleichen Begriffs (z.B. `Pr`, `Pt`, `Prt`, `Prn`, `Prnt` für `Print`) und variieren Sie die Groß/Kleinschreibung möglichst oft (damit keiner auf die Idee kommt, Ihren Code mit **grep** durchsuchen zu wollen).

11. Setzen Sie nie {...} um **if/else/for/while-Blöcke**, außer dies ist syntaktisch unbedingt notwendig. Dies gilt insbesondere in tief verschachtelten Schleifen und Bedingungen.

12. Brechen Sie grundsätzlich die Regeln, keine `goto`s, keine frühen `return`s und keine markierten `break`s zu verwenden, außer Sie können dadurch die Verschachtelungstiefe von if/else-Blöcken um mehrere Ebenen erhöhen.

13. Verwenden Sie sehr lange **Namen**, die sich nur in einem einzigen Zeichen (in der Groß/Kleinschreibung) voneinander unterscheiden. Die Zeichen `1lIi`, `0ODo`, `8B`, `tf`, `uvUV` und `nm` bieten sich dafür geradezu an (z.B. `parselnt` und `parseInt` oder `DOCalc` und `D0Calc`).

14. Wo auch immer es die Zugriffsregeln gestatten, verwenden Sie bereits **vorhandene Variablen** für andere Zwecke erneut (um Speicherplatz zu sparen). Dies gilt insbesondere für temporäre Variablen.

15. Verwenden sie das kleine `l` für `long`-Konstanten, da z.B. `10l` leichter mit `101` verwechselt wird als `10L`.

16. Verwenden Sie nie `i` als **Schleifenvariable**, sondern ausschließlich für andere Zwecke; insbesondere sollte `i` keine `int`-Variable sein. Verwenden sie analog `n` als Schleifenvariable, nicht als Schleifengrenze.

17. Führen Sie nie **(modul)lokale** Variablen oder Funktionen ein, irgend jemand könnte irgendwann direkten Zugriff darauf brauchen.

18. Dokumentieren Sie niemals einen gefundenen **Fehler** im Code (**„gotcha“**), insbesondere wenn er sehr schwierig zu finden war. Sollten Sie einen Fehler im Code vermuten, schreiben Sie keinen Hinweis wie `/* ??? */` in den Code, sondern behalten ihn für sich.

19. Damit keine Langeweile beim Lesen Ihres Codes aufkommen kann: Verwenden Sie soviele verschiedene **Begriffe** wie möglich für den gleichen Zweck (z.B. `display`, `show`, `print`, `output`, `generate`, `create`, ...). Schlagen Sie in einem **Thesaurus** nach, falls Ihnen nicht genügend Varianten einfallen.

20. Verwenden Sie aber auf jeden Fall den gleichen Begriff, wenn Sie zwei ähnliche Funktionen haben, die sich in einer wesentlichen Sache unterscheiden (z.B. `print` für das Schreiben auf Datei, Ausgeben auf einem Drucker und Anzeigen auf einem Bildschirm).

21. Verwenden Sie möglichst viele **abstrakte Begriffe** bei der Benennung von Funktionen, z.B. `routineX48`, `PerformDataFunction`, `DoIt`, `HandleStuff`, `doArgsMethod`.

22. Verwenden Sie Substantive für Funktionsnamen (z.B. `Method`) und Verben oder Eigenschaftswörter für Variablen (z.B. `print`, `isnew`).

23. Übergeben Sie Parameter grundsätzlich „**by reference**“, auch wenn Sie nicht geändert werden. Dokumentieren Sie nie, ob eine Funktion Parameter ändert oder nicht. Benennen Sie Parameter verändernde Funktionen so, dass sie den Anschein erwecken, als greifen sie darauf nur lesend zu.

24. Dokumentieren Sie nie die **physikalische Einheit** von Variablen oder Ein-/Ausgabe-Parametern. Ebenso sollten Sie die Einheit von Umrechnungskonstanten — und wie diese berechnet wurden — nicht dokumentieren.

25. Es gibt zwei Möglichkeiten der **Einheitenbehandlung**: (1) jede Eingabe in metrische Einheiten umwandeln, mit metrischen Einheiten rechnen und bei der Ausgabe wieder zurückwandeln, (2) die verschiedenen Einheiten gleichzeitig behanden. Wählen Sie grundsätzlich die 2. Methode, das ist der Amerikanische Weg.

26. Verwenden Sie bei Zugriffen auf Arrays abwechselnd folgende Schreibweisen: `array[i]`, `*(array + i)`, `i[array]`, `*(i + array)`. Verwenden Sie analog bei Zugriffen über Zeiger abwechselnd folgende Schreibweisen: `(*p).xxx`, `p→xxx`, `(*(&p))→xxx`, `(*(*(&p))).xxx`.

27. Versuchen Sie **Konstanten** — wie z.B. die Länge **100** eines Arrays — an möglichst vielen Stellen fest in den Code zu schreiben. Verwenden Sie statt **100/2** oder **100-1** die Werte **50** oder **99**. Wenn Sie schon eine Konstante einführen, dann „vergessen“ Sie das ab und zu und verwenden an einigen Stellen im Code den Wert direkt. Wenn zwei verschiedene Konstanten zufällig den gleichen Wert haben, verwechseln Sie diese gelegentlich.

28. Definieren Sie für die Zahlen 1..1000 die Konstanten `one`, `two`, `... thousand` und verwenden Sie diese anstelle der Zahlen (ihr Code ist dann schon mal 1000 Zeilen länger).

29. Schachteln Sie Code so tief wie möglich ein, statt ihn in Ausdrücke mit temporären Variablen und Funktionen zu zerlegen. Gute Entwickler erreichen mit Leichtigkeit mehr als 10 ()-Ebenen in einer Zeile und mehr als 20 {}-Ebenen in einer Funktion.

30. Zerlegen Sie Ihr Programm nicht in **logische Einheiten** wie Module und Funktionen, sondern schreiben Sie es „am Stück“, damit es möglichst kurz und übersichtlich bleibt und Sie nicht im Code hin- und herspringen müssen, sondern es immer von vorne nach hinten lesen können.

31. Der **Präprozessor** bietet eine weitere — vom Code völlig unabhängige — Methode der Verschachtelung; nutzen Sie diese ausgiebig.

32. Sie erhalten eine Belohnung für jeden **Block**, dessen Anfang und Ende getrennt auf 2 Seiten ausgedruckt werden.

33. Verwandeln sie verschachtelte if/else grundsätzlich in **bedingte Anweisungen** (`?:`-Konstrukte).

34. Die **AboutBox** sollte nur Ihren Namen (und wenn möglich Ihr Bild, Ihren Lebenslauf und Ihre Hobbies) sowie den üblichen Text zur Ablehnung jeglicher Garantien und Gewährleistungsansprüche enthalten. Sie sollte nie den Programmnamen, eine Beschreibung des Programmzwecks, eine Major- und Minor-Versionsnummer sowie das Erstellungsdatum enthalten.

35. Übergeben Sie möglichst viele Parameter über das **Environment** (Umgebung), damit jeder Anwender darauf achten muß, genau das gleiche Environment einzustellen, insbesondere wenn das Programm automatisch als Batch-Job gestartet wird. Geben Sie

die verwendeten Environmentvariablen, ihre Defaultwerte und die für sie möglichen Werte nur auf konkrete Nachfrage preis. Das gleiche sollte für **Aufruf-Parameter**, **Optionen** und **INI-Dateien** gelten.

36. Wählen Sie **Variablennamen** so, dass sie nichts mit den Texten in Eingabemasken oder Ausdrucken zu tun haben (z.B. `postal_code` als Label in der Eingabemaske, `zip` für die zugehörige Variable und `PLZ` im Ausdruck).

37. Lassen Sie alle nicht mehr benutzten Variablen und Funktionen in Ihrem Code stehen, Sie könnten sie nochmal brauchen („**Don't re-invent the wheel**").

38. Erstellen Sie *n* verschiedene **Funktionen** statt eine generische Funktion mit Parametern zu verwenden (z.B. `LeftAlign()`, `AlignRight()` und `Center()` statt `Align(int direction)`).

39. Die **Kama Sutra Technik** bietet den unschätzbaren Vorteil, neben dem Wartungsentwickler auch den Dokumentierer und den Anwender zu verwirren. Erzeugen Sie viele verschiedene fast gleichnamige Funktionen, die sich nur in kleinen Details unterscheiden. Dies sorgt gleichzeitig für eine aufgeblähte und somit veraltete Dokumentation (es war wohl erst Oscar Wilde, der feststellte, dass sich die Positionen 47 und 115 im Kama Sutra nur dadurch unterscheiden, dass die Frau in Position 115 die Finger gekreuzt hat).

40. Machen Sie alle Funktionen und Variablen **global**, irgend jemand könnte Sie irgendwann verwenden wollen (Prinzip des „**Transparenten Interfaces**"). Sobald eine Funktion oder Variable global ist, kann Sie nicht mehr zurückgezogen werden, oder? Auf diese Art und Weise wird eine spätere Änderung an internen Datenstrukturen und Routinen praktisch unmöglich gemacht, außer man nimmt beliebig viele Seiteneffekte in Kauf.

41. Verwenden Sie soviele **Programmiertricks** wie möglich (erst hierin zeigt sich die wahre Beherrschung einer Programmiersprache).

42. Nutzen Sie jede **Besonderheit** Ihres Compilers oder Ihrer Betriebssystemversion aus (am besten im Namen der „Effizienz").

43. Verlassen Sie sich nicht auf den Compiler, sondern **optimieren** Sie den Code selbst (z.B. `i << 2` statt `i * 4`, `i + 3140` statt `i + 3 * 1000 + 140` oder Schleifen entrollen)

44. **Rücken** Sie nach Gutdünken **ein**, ein absoluter langjähriger Klassiker, über den man sich bei der Fehlersuche immer wieder wie ein Schneekönig freut, ist:

```
if (a)
   if (b) x = y
else w = z
```

45. Ersetzen Sie überall aus Effizienzgründen Floating-Point durch komplizierte Formeln mit **Integer-Arithmetik**.

46. Verwenden Sie eine 50:50-Mischung aus deutschen und englischen **Begriffen** für die Namen von Datenbanken (Tabellen, Spalten), Datenstrukturen (Klassen, Objekten), Funktionen (Methoden), Variablen (Feldern) und Konstanten. Mischen sie vor allem innerhalb eines Namens.

47. Überladen Sie System- bzw. API-Funktionen durch Ihre eigenen (z.B. `malloc`). Am besten mit Hilfe von **Macros**, dann lassen sie sich nicht mehr vernünftig debuggen.

4 Einige Gedanken und Fragen

1. Software-Entwicklung ist eine **kommunikative Tätigkeit** und erfordert soziale Kompetenzen sowie profundes Wissen in Gesprächsführung und Konfliktlösung.

   Werden die Mitarbeiter dafür ausgebildet?

2. Um **Probleme** bei der Definition und Erstellung von Software zu verstehen, sollte man folgende Fragen stellen:

   - ▷ „Wem wird die Software nützen und wem wird sie schaden?“
   - ▷ „Wer wird Macht verlieren und wer wird welche hinzugewinnen?“
   - ▷ „Liegen die Gewinn- und Verlustsituationen der beteiligten Parteien auf dem Tisch?“

   Kümmert sich irgend jemand um diese Gesichtpunkte und diskutiert sie mit den Betroffenen?

3. Wer sich festlegt hat verloren. Wer hingegen **keine Vorgaben** macht, kann jederzeit die Funktionalität der Software kritisieren und die Bezahlung verweigern.

   Ist das gute Zusammenarbeit?

4. Eine Spezifikation, die keine **vollständige Aufzählung** der Ein- und Ausgaben und des Zusammenhanges zwischen ihnen enthält, ist ein Flop; sie erfüllt die Anforderungen an eine Spezifikation nicht einmal ansatzweise.

   Hat dies irgend einer der Anwender verstanden und akzeptiert?

5. Wer die **erste Zahl** nennt, hat verloren (da sie nur noch kleiner werden kann).

   Fordert man dadurch nicht unrealistische Abschätzungen heraus?

6. Software ist **viel zu teuer**.

   Verglichen mit was für einem Ersatzprodukt eigentlich?

7. Ein anderes Wort für **Workaround** ist „Bastellösung“.

   Würden Sie mit Ihrer Familie im Auto über eine Brücke fahren, von der Sie wissen, dass sie nicht konstruiert, sondern gebastelt ist?

8. Den **fehlenden Konsens** bezüglich Funktionalität, Vorgehensweise, Verantwortung und Finanzierung zwischen Kunden, Geschäftsführung und Vorstand kann die Software-Entwicklung nicht lösen.

   Wer ist bereit, diesen Konsens herzustellen?

9. Mit ständig neuer Aufwandsschätzung, Kapazitätsplanung, Kostennachweis, Kostendiskussion, Kostenumverteilung, Kostenabrechung, Leistungserfassung, ... wird ein beträchtlicher Teil der Arbeitszeit der Entwickler für im Sinne der Software-Entwicklung **völlig unproduktive Tätigkeiten** gebunden.

   Wofür könnte wohl diese Zeit verwendet werden?

10. Das **Abwälzen** der obigen Punkte auf die Software-Entwickler führt zu Bürokratie, Absicherungstendenzen, Kommunikationsstörungen, Mißtrauen und Risikovermeidung.

    Sind wir nicht auf die vertrauensvolle Zusammenarbeit aller angewiesen?

11. **Druck und Drohungen** sind die allerletzte Zuflucht von Managern, die versagt haben.

    Was für Konsequenzen hat eigentlich die ständige Misere?

12. Wenn Entwickler ständig Kanäle und Dämme unter hohem Zeit- und Kostendruck basteln, dann **brennen Sie aus**, verlassen die Firma oder beginnen nach der Devise „Nach mir die Sintflut“ zu leben.

    Was für Kosten wird das wohl in der Zukunft verursachen?

5 Quellen

- How to write unmaintainable Code (`http://www.mindprod.com/unmain.html`).
- „Der Termin“ von Tom DeMarco (Hanser Verlag).
- Eigene Erfahrungen und Ideen.

— END —