Reengineering Web-basierter und anderer junger Systeme Erfahrungsbericht

Benedikt Mas y Parareda Dr. Markus Pizka

itestra GmbH
Ludwigstr. 35
D-86916 Kaufering
info@itestra.de

10. Oktober 2006

Zusammenfassung

Unter „Reengineering“ werden alle Maßnahmen subsumiert, die der nachträglichen Steigerung der Qualität existierender Software-Systeme dienen. In diesem Zusammenhang werden in aller Regel so genannte „Alt“-Systeme oder „Legacy“-Systeme betrachtet, deren Entwicklung oft mehr als 30 Jahre zurückliegt. Der Bedarf an Reengineering-Maßnahmen ist jedoch nicht auf diese Systeme beschränkt, sondern kann auch sehr „junge“ Systeme betreffen. Dies gilt besonders für webbasierte Systeme, die im Zuge der .COM Euphorie unter der Maxime der gleichzeitigen Minimierung von „time-to-market“ und Entwicklungskosten entstanden sind. Die so entstandenen Systeme leiden nicht nur unter Qualitätsmängeln, sondern sind auch aufgrund der zahlreichen und gleichzeitig eingesetzten Technologien (JSP, Servlets, PHP, JavaScript, XML, ...) besonders schwer zu entwickeln respektive nachträglich zu analysieren.

Anhand konkreter Beispiele aus der Praxis beschreibt der vorliegende Erfahrungsbericht die qualitativen Mängel eines webbasierten Software-Produkts, deren Konsequenzen sowie die Anforderungen an die nachträgliche Analyse und Restrukturierung des Systems. Die Erfahrungen aus diesem Beispiel werden anschließend um weitere Erfahrungen mit ebenfalls jungen Systemen ergänzt. Abschließend wird ein Modell zur systematischen Bewertung von Qualität in der Software Entwicklung vorgestellt.

1 Junge Legacy-Systeme?

Über den gesamten Lebenszyklus eines Software-Systems hinweg betrachtet entfällt nur ein geringer Teil der Gesamtkosten auf die initiale Entwicklung. Der weitaus größere Anteil, ca. $70 - 90\%$, muss für die Wartung des Software-Systems aufgewendet werden [5, 19, 27] (siehe Abbildung 1).

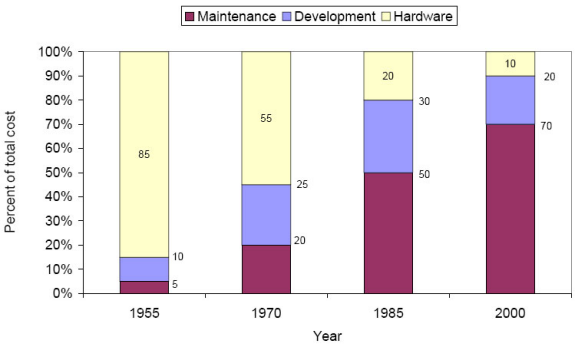

Abbildung 1: Wachsender Anteil Wartungskosten

Dieser hohe prozentuale Anteil an Wartungskosten an den Lebenszykluskosten führt in Unternehmen, die zahlreiche Software-Systeme einsetzen, dazu, dass ein Großteil des verfügbaren Budgets in der Pflege der vorhandenen Systeme gebunden ist. Hierdurch wird die Handlungsfähigkeit der betroffenen Unternehmen eingeschränkt, da kein Budget für neue Anwendungen zur Verfügung steht.

In der Praxis führt dies häufig dazu, dass die Wartung von Software generell als Problem be-

trachtet wird. Allerdings ist dieser Schluss nur bedingt berechtigt. So entsteht bei erfolgreichen Software-Systemen, die über viele Jahre von zahlreichen Benutzern genutzt und laufend an die sich ändernden Geschäftsprozesse angepasst werden, ganz offensichtlich ein Großteil der Kosten im Rahmen der Wartung und nicht bei der initialen Entwicklung. Das heißt, ein hoher Kostenanteil der Wartung am Gesamtlebenszyklus kann im Einzelfall sogar ein Hinweis auf ein besonders erfolgreiches System sein.

Ein reales Problem sind jedoch hohe und zunehmende Kosten pro Änderung, da sie nicht nur das zur Verfügung stehende Budget belasten, sondern die Wirtschaftlichkeit vorhandener Anwendungen gefährden.

1.1 Bedrohung Qualitätsverfall

In „Does Code Decay?“ beschreibt Stephen Eick [11] anhand der Untersuchung umfangreicher Software-Systeme das Phänomen des graduellen Qualitätsverfalls. Er zeigt, dass der Verlust an Qualität in Software-Systemen mit der Anzahl der durchgeführten Änderungen korreliert und dazu führt, dass Änderungen zunehmend teurer und fehlerträchtiger werden. Technisch manifestiert sich das Phänomen des Verfalls in Form von

- Strukturverlust und Architekturverletzungen,
- zahlreichen breiten Schnittstellen,
- Redundanz und Inkonsistenz,
- Behelfslösungen („Hacks“, Indirektionen, etc.),
- aufgeblähtem und „umständlichen“ Code.

Es werden drei Ursachen für Code-Verfall und dessen ständig zunehmende Beschleunigung identifiziert:

- Mangelhafte technische Grundlagen:
  - ungeeignete Architektur
  - Verletzung von Entwurfsprinzipien
  - fehlender Änderungsprozess
  - unzulängliche Werkzeuge
- Unterschiede zwischen Entwicklern
- organisatorische Schwierigkeiten
  - unangemessene Kommunikationsmittel
  - ungenaue Anforderungen
  - Zeitdruck

Systeme, die unter den genannten Symptomen des Qualitätsverfalls leiden, werden oft als problematisch angesehen und als *Legacy-Systeme* [3,18,22] bezeichnet, da sie eine Reihe der Merkmale aufweisen, die üblicherweise mit diesem Begriff verbunden werden.

1.2 Legacy Risiken

In gängigen Definitionen werden Legacy-Anwendungen als Systeme beschrieben,

> „... die sich oft durch unzureichende Dokumentation, veraltete Betriebs- und Entwicklungsumgebungen, zahlreiche Schnittstellen und hohe Komplexität auszeichnen.“ [1]

Üblicherweise sind damit Systeme gemeint, deren Entwicklung schon viele Jahre zurückliegt.

Weitere und eng verwandte problematische Eigenschaften, die Legacy-Systemen zugeschrieben werden, sind:

- Die Integration in eine moderne IT-Landschaft mittels aktueller Kommunikationsstandards ist schwierig und erfordert Speziallösungen.
- Die Software unterstützt den entsprechenden Geschäftsprozess nur noch mangelhaft.
- Die vorhandene Dokumentation ist aufgrund vernachlässigter Pflege
  - lückenhaft,
  - veraltet,
  - inkonsistent,
  - und unstrukturiert.
- Das Wissen über das System ist aufgrund von Mitarbeiter-Fluktuation verloren gegangen.

[1] Wikipedia, `http://de.wikipedia.org/`, Juni 2006

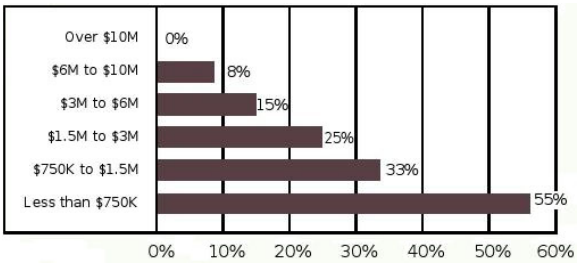

Abbildung 2: Erfolgswahrscheinlichkeit in Abhängigkeit von Projektgröße

- Werkzeuge für die verwendete Technologien entsprechen nicht mehr dem Stand der Technik oder sind nicht mehr verfügbar.
- Es gibt Schwierigkeiten, Mitarbeiter zu finden, die im Umgang mit den eingesetzten Technologien ausgebildet sind (z. B. Programmiersprachen geringer Verbreitung).

Um diesen Problemen zu begegnen und die laufenden Kosten zu senken, werden Legacy-Systeme Reengineering-Maßnahmen unterzogen oder durch eine Neuentwicklung abgelöst. Eine vollständige Neuentwicklung birgt jedoch häufig große Risiken.

1. Die Neuentwicklung einer gewachsenen Legacy-Anwendung bedeutet naturgemäß ein sehr großes Projekt. Für Großprojekte hat die Erfahrung allerdings gezeigt, dass sie eine verhältnismäßig geringe Erfolgswahrscheinlichkeit besitzen [15]; s. Abb. 2.
2. Werden vor der Ablösung die Ursachen, die zum Qualitätsverfall geführt haben, nicht sorgfältig analysiert und Gegenmaßnahmen eingeleitet, wird das neue System in wenigen Jahren unter den alten Problemen leiden.

Hinzu kommt, dass durch das bestehende, weitgehend funktionstüchtige System die Neuentwicklung erschwert wird:

- Die vollständigen Anforderungen, die das vorhandene System abdeckt, sind meist weder bekannt noch explizit dokumentiert.
- Die Ablösung eines Software-Systems erfordert ein aufwändiges Änderungsmanagement innerhalb der Organisation.

Alternativ zur riskanten Neuentwicklung ist es unter verschiedenen Voraussetzungen auch möglich, sukzessive Teile des Legacy-Systems nachträglich zu verbessern oder zu ersetzen. Wenngleich auf diese Weise die Risiken einer Ablösung vermieden werden, werden auch durch diese Strategie die eigentlichen Ursachen, die zu den genannten Problemen geführt haben, nicht dauerhaft beseitigt.

1.3 Keine Frage des Alters

Bei einer genauen Betrachtung der negativen Merkmale, die Legacy-Systemen zugeschrieben werden, wird deutlich, dass ein Großteil dieser Merkmale prinzipiell unabhängig vom Alter des Systems ist. „Strukturverlust“ umschreibt lediglich das tatsächliche Problem Strukturmangel und führt dieses auf die verstrichenen Zeit zurück. Ebenso besitzen die Defizite Architekturverletzungen, Redundanzen, „umständlicher Code“, u. v. m. keinen engeren Bezug zur Zeit und können schon während der Entwicklung entstehen. Die oben genannten Untersuchungen zum Code-Verfall zeigen zwar eine Korrelation zwischen Alter und der Anzahl von Qualitätsdefiziten, aber die Ursache der Mängel ist nach wie vor die Art und Weise, wie Änderungen durchgeführt werden.

Auch junge Systeme können schnell zu Legacy-Systemen werden, wenn sie ohne ausreichendes Qualitätsmanagement entwickelt wurden oder schon früh nach Auslieferung mangelhafte Wartungsarbeiten durchgeführt wurden.

Der vorliegende Erfahrungsbericht wird anhand von drei kommerziellen Software-Systeme kleiner bis mittlerer Größe belegen, dass Legacy im Wesentlichen keine Frage des Alters, sondern eine Frage des Qualitätsmanagements ist. Ganz offensichtlich erhöhen Qualitätsdefizite während der Entwicklung und bei frühen Änderungen die Kosten späterer Wartungsarbeiten. Verblüffend ist jedoch, mit welcher Konsequenz in der Praxis anerkannte Regeln und Praktiken missachtet, Qualitätsmängel eingeführt und von Beginn an Legacy entwickelt wird.

Übersicht

In Abschnitt 2 werden zunächst wichtige Begriffe wie *Software Wartung*, *Reengineering* und *Reverse Engineering* geklärt. Da nach unserer Erfahrung

die beschriebenen Qualitätsprobleme besonders häufig web-basierte Systeme des .COM Booms betreffen und außerdem Wartungsarbeiten an diesen Systemen besonders aufwändig sind, widmet sich Abschnitt 3 den Spezifika web-basierter Systeme. In Abschnitt 4 wird ein junges web-basiertes System vorgestellt, dessen Qualitätsmängel bereits bei Fertigstellung ein Reengineering erforderten. Der darauf folgende Abschnitt zeigt weitere Beispiele für Systeme, die bereits nach kurzer Zeit zu Problemfällen wurden. Abschnitt 6 stellt ein Qualitätsmodell vor, das zur Vermeidung der genannten Qualitätsdefizite sowohl während der Entwicklung als auch bei der Weiterentwicklung verwendet werden kann.

2 Terminologie

Im Umfeld der Software-Wartung und des Reengineerings werden einige Begriffe verwendet, für die es bislang keine präzisen und allgemeinhin akzeptierten Definitionen gibt. Um hieraus resultierende Missverständnisse zu vermeiden, wird in den folgenden Abschnitten das Verständnis einiger wichtiger Begriffe aus Sicht dieses Papiers geklärt.

2.1 Software Wartung

Der Begriff *Software Wartung* schließt in diesem Papier alle Arbeiten an Programmen inklusive deren Dokumentation ein, die der Pflege und Weiterentwicklung nach der ersten Auslieferung dienen [24]. Die Wartung beginnt dementsprechend nicht nach der Auslieferung sondern sollte bereits mit der Planung des Systems einsetzen.

Wartungsarbeiten nach der Auslieferung werden in *Wartungsfällen* durchgeführt, die analog zu Projekten mit vorgegebenen Ressourcen definierte Ziele verfolgen. Es lassen sich vier Arten von Wartungsfällen unterscheiden:

Perfektion - Umsetzung neuer Anforderungen

Adaption - Anpassung an technische Änderung; z. B. neuer Compiler oder neue Hardware

Korrektur - Beseitigung von Fehlern

Prävention - Vorbeugende Maßnahmen

2.2 Reengineering

Reengineering ist ein wesentlicher Teil der Software-Wartung und umfasst alle Arbeiten, die der nachträglichen Qualitätssteigerung dienen. Hierzu gehört die Perfektion in Bezug auf nichtfunktionale Anforderungen (z. B. Performance), Korrektur oder Prävention. Technische Adaption oder die Umsetzung neuer funktionaler Anforderungen sind hingegen keine Reengineering-Aktivitäten. Reengineering findet primär nach der Auslieferung, aber auch schon während der Entwicklung statt.

Varianten des Reengineerings sind [24, 27]:

Retargetting - Neuausrichtung/Migration auf eine neue Plattform.

Restrukturierung - Nachträgliche Modularisierung einer bestehenden Anwendung, Veränderung der bestehenden Modularisierung bzw. Reorganisation der Abhängigkeiten innerhalb der Anwendung.

Transformation - (Semi-)automatische Umformung eines Programms unter Erhalt der Semantik.

Reverse Engineering - Analyse eines bestehenden Systems, um Informationen über interne Abläufe und Verhaltensweisen zu gewinnen oder verloren gegangene Information wieder zu erlangen [2].

2.3 Reverse Engineering

Die nachträgliche Extraktion von Information aus einem bestehenden System ist eine Voraussetzung für nahezu alle Wartungsarbeiten. Untersuchungen belegen, dass ca. 50% des gesamten Wartungsaufwands allein hierfür anfallen [14].

Beispiele für Fragen, die durch Reverse Engineering beantwortet werden, sind „Warum tritt in Situation S ein Fehler auf?“, „Wie verhält sich die Performanz bei N Benutzern?“, „Welche Auswirkung hat die Änderung der Komponente K?“, „Wie ist Ablauf A implementiert?“ oder „Welche Motivation hat zur Designentscheidung D geführt?“.

In verschiedenen Fällen, z. B. bei Outsourcing, Wechsel des Entwickler-Teams oder nach Verlust bzw. Veralterug der Dokumentation, erfordert das

Reverse Engineering des völlig unbekannten Systems ein eigenes umfangreiches Projekt.

Methoden und Techniken

Neben der manuellen Analyse durch Lesen des bestehenden Quellcodes, Datenbank-Schemas und ggf. verfügbarer Zusatzdokumente, können Werkzeuge, u. a. durch Textsuche [26], Strukturvisualisierung[2], Dekompilierung oder Verhaltensprotokollierung, ein Reverse Engineering unterstützen.

Bezüglich der Vorgehensweise sind zwei Strategien zu unterscheiden. Bei der *top-down* Analyse wird eine Hypothese aufgestellt und anhand des zu untersuchenden Objekts belegt oder widerlegt.

Beispiel: Vermutung „Komponente K sortiert Daten alphabetisch aufsteigend“ und Prüfung durch Ausführung mit Testdaten.

Bei der *bottom-up* Strategie wird anhand der Untersuchung der Bestandteile des zu untersuchenden Objekts versucht, Rückschlüsse über das gesamte Objekt zu ziehen.

Falls z. B. in einer unbekannten Klasse `X` lediglich zwei Methoden `push` und `pop` gefunden werden, so könnte daraus geschlossen werden, dass die Klasse einen Keller implementiert.

In dem vorliegenden Erfahrungsbericht werden drei Systeme einem Reverse Engineering unterzogen. Aufgrund eines Mangels an Werkzeugen für die verwendeten Technologien (s. u.) musste die Analyse manuell durchgeführt werden, wobei beide Strategien zum Einsatz kamen.

3 Web-basierte Systeme

Web-basierte Anwendungen, die zu Zeiten des .COM Booms unter hohem Zeit- und Kostendruck entwickelt wurden, leiden häufig unter Qualitätsmängeln und müssen Reverse Engineering und anderen Reengineering Maßnahmen unterzogen werden.

Die Erfahrungen mit den beiden web-basierten Legacy-Systemen in diesem Bericht zeigen, dass bei Anwendungen dieser Art technische Gründe besondere Qualitätsprobleme verursachen und gleichzeitig die Analyse erschweren.

Sprachmix Um grafische Oberflächen mit dynamischem Inhalt zu gestalten, kommen bei Web-Anwendungen stets gleichzeitig mehrere unterschiedliche Technologien zum Einsatz. In der Praxis werden Fragmente von Beschreibungen der Oberfläche und der Programmlogik im Programmtext oft eng miteinander verwoben. Varianten dieses Sprachmix sind:

- Einbettung von HTML [28] Markup in String Konstanten und Variablen von Skriptsprachen und Ausgabe in `print()` Anweisungen
- Einbettung von Skriptsprachen in den HTML Markup

Diese Mischung behindert die Analyse des Quellcodes mit Werkzeugen, da diese in der Regel auf genau eine Technologie ausgerichtet sind. Darüber hinaus reduziert der Sprachmix sowohl die Wiederverwendbarkeit von Programmen als auch deren Verständlichkeit, da inhaltlich zusammengehörige Elemente der Oberfläche oder der Programmlogik fragmentiert und weit verstreut sind.

Durch „Template Engines“[3] kann die Fragmentierung reduziert und die Wiederverwendung grafischer Elemente verbessert werden, wobei der Sprachmix auch damit nicht vollständig beseitigt wird.

Mangelnde Reife der Technologien Die verwendeten Technologien waren in der Vergangenheit oft kurzlebig und unterlagen schnellem Wandel. Technologien wie CGI oder Applets galten bereits nach kurzer Zeit als veraltet. Sprachen wie PHP [13] besitzen, ungeachtet ihrer Popularität, im Vergleich zu klassischen Programmiersprachen einen geringen Reifegrad, der in diversen praktisch relevanten Defiziten zum Ausdruck kommt (s. u. a. [20]). Die Einhaltung akzeptierter Software-Engineering Methoden, z. B. „Separation of Concerns“ [23] oder die *Model-View-Controller* Architektur[4], ist unter diesen Voraussetzungen schwierig.

[2] z. B. `http://www.software-tomography.com/html/sotograph.htm` oder `http://www.borland.com/us/products/together/index.html`

[3] z. B. `http://jakarta.apache.org/velocity/`

[4] siehe z.B. `http://en.wikipedia.org/wiki/Model-view-controller`

Ungeeignete Basistechnologie Die Basis webbasierter Anwendung ist das Hypertext Transfer Protocol (http) [29], das nicht für interaktive Anwendungen konzipiert wurde und nur bedingt für diesen Zweck geeignet ist. Die Kommunikation des Browsers mit dem Server muss sich in der Regel auf das inperformante Anfragen und Versenden von HTML-Seiten beschränken. Die hieraus resultierenden Leistungseinbußen müssen durch Performance-Optimierungen zu Lasten der Wartbarkeit kompensiert werden.

Ein Grundvoraussetzung für langlebige webbasierte Anwendungen ist es, diese Probleme bereits während der initialen Entwicklung zu adressieren. Darüber hinaus sind die Qualitätskriterien, die auch für „klassische" Anwendungen gelten, auch für web-basierte Systeme zu berücksichtigen.

In dem folgenden Beispiel eines jungen webbasierten Legacy-Systems hat einerseits die gewählte Web-Technologie den Qualitätsverfall gefördert, dessen Wurzeln andererseits in einem generell unreifen Entwicklungsprozess und der Abwesenheit eines effektiven Qualitätsmanagements liegen.

4 Ein web-basiertes Legacy-System

Die Anwendung *WebBuilder*[5] wurde im Zeitraum 1999 bis 2004 durch ein deutsches Unternehmen entwickelt und als Kern-Produkt dieses Unternehmens in Europa vertrieben.

WebBuilder besteht aus einer Entwicklungs- und einer erweiterbaren Laufzeitumgebung. Mit Hilfe der Entwicklungsumgebung können aus vorgefertigten und neu eingebrachten Komponenten Web-Portale zusammengestellt und innerhalb der Laufzeitumgebung ausgeführt werden.

Das Ergebnis der Konfiguration eines Portals mit der Entwicklungsumgebung wird teilweise in einer relationalen Datenbank, teilweise als generierte PHP-Skripten [13] abgelegt, wobei die generierten PHP-Skripten fast ausschließlich Konfigurationsstrings enthalten, die von menschlichen Benutzern weder verstehbar noch sinnvoll änderbar sind. Da es sich bei PHP um eine interpretierte Sprache handelt und die Datenbankanteile ebenfalls jederzeit modifizierbar sind, können einzelne Portalkomponenten auch im laufenden Betrieb verändert oder ausgetauscht werden. Der Nutzer von WebBuilder erhält damit eine äußerst flexible Lösung für die rasche (Re-)Konfiguration und Einführung von Web-Portalen.

Sowohl Entwicklungs- als auch Laufzeitumgebung sind primär in einer Kombination der Programmiersprache PHP und der Seitenbeschreibungssprache HTML implementiert. Strukturell existiert keine Trennung von Entwicklungs- und Laufzeitumgebung, da die Entwicklungsumgebung selbst Komponenten der Laufzeitumgebung nutzt (siehe Abbildung 3).

4.1 Vorgefundener Zustand

Der Entwurf und die Implementierung des Systems erfolgten aus Kostengründen mit talentierten und hoch motivierten Entwicklern in Osteuropa, jedoch ohne genaue Vorgaben,

- welches Vorgehensmodell zu verwenden ist,
- welche Dokumentation zu erstellen ist und
- wie Qualitätssicherung durchzuführen ist.

Da diese grundlegenden Vorgaben gefehlt haben, erfolgte auch keine Kontrolle des Entwicklungsfortschritts und der Qualität der erzielten Ergebnisse.

Nach ca. vier Jahren kam es zu schwerwiegenden finanziellen und persönlichen Differenzen und schließlich zur Trennung des deutschen Unternehmens von den osteuropäischen Entwicklern. Die Entwicklung war zu diesem Zeitpunkt bereits weit fortgeschritten, so dass ein lauffähiges System bestand und an erste Kunden verkauft war.

Allerdings konnten die Mitarbeiter in Deutschland die Entwicklung nicht unmittelbar fortsetzen, da sie zuvor nicht an der Entwicklung beteiligt waren und aufgrund der fehlenden Vorgaben kaum Information über die innere Struktur des Systems besaßen. WebBuilder war plötzlich quasi eine Legacy-Anwendung, die nicht mehr gewartet und weiterentwickelt werden konnte. Um eine Weiterentwicklung wieder zu ermöglichen, wurde

[5] Aus Gründen der Vertraulichkeit wurden Namen, Daten und Codebeispiele anonymisiert.

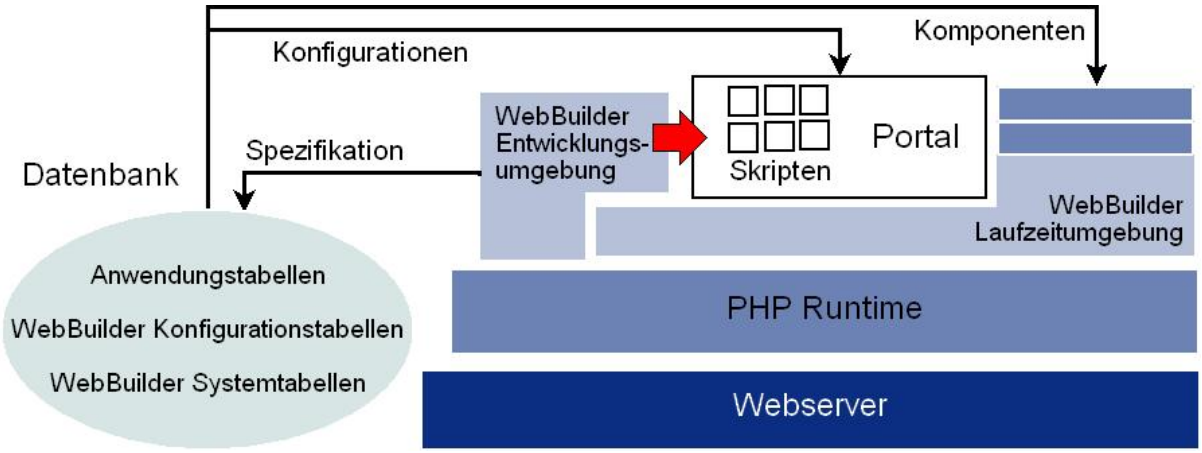

Abbildung 3: WebBuilder-Struktur und Portal Konfigurationen.

WebBuilder einem Reverse Engineering unterzogen.

4.2 Vorgehen

Um eine effiziente und effektive Analyse zu gewährleisten, wurde zunächst ein einfaches Configuration Management System eingeführt und die bestehende Dokumentation, soweit möglich, gesichtet, indiziert und strukturiert. Außerdem wurden Richtlinien für die Ergebnisdokumente des Reverse Engineering entworfen, um eine gleichbleibend hohe Qualität zu gewährleisten und sicherzustellen, dass die durch das Reverse Engineering gewonnene Information dauerhaft erhalten bleibt.

Um den Prozess des Verstehens des vorhandenen Systems zu erleichtern, wurde zunächst ein minimales System untersucht. Das heißt, dass aus der Menge aller vorhandenen PHP Skripten zunächst nur die Skripten untersucht und dokumentiert wurden, die für ein WebBuilder Portal zwingend erforderlich sind. In weiteren Schritten wurde dann die Betrachtung schrittweise auf das vollständige System erweitert.

Da das im Inneren unbekannte System selbst ausführbar war, wurde für die Untersuchung eine *Top-Down*-Strategie verfolgt:

1. Es wurde davon ausgegangen, dass nur der Teil des Programmes relevant ist, der bei der Interaktion eines Benutzers zur Ausführung kommt. Programmteile, die diese Forderung nicht erfüllen sind nicht erreichbar und damit überflüssig.

2. Anhand des beobachtbaren Verhaltens des Systems in Ausführung wurden Annahmen über Funktionsblöcke getroffen (als Block wird jeweils eine Menge an PHP Skripten bezeichnet, die gemeinsam Funktionalität bereitstellen).

3. Es wurde versucht, die Annahmen über die Existenz der Funktionsblöcke im Programmtext und anhand der Datenbankinhalte zu validieren oder zu falsifizieren.

   Bei den Versuchen der Identifikation von Funktionsblöcken wurden drei verschiedene Sichten getrennt:

   - Benutzerinteraktion
   - Interne Abläufe
   - Schnittstellen zwischen Komponenten

   Bei der Analyse von Benutzerinteraktionen konnte die Eigenschaft web-basierter Systeme genutzt werden, dass Benutzer generell nur auf zwei Arten mit einer Web-Anwendung interagieren können:

   (a) Die grafische Oberfläche enthält Steuerelemente, z. B. Buttons oder Links, die entweder an HTML GET oder POST Events Funktionen oder direkt an

HTML URL Adressen gebunden sind. In beiden Fällen wird unmittelbar die zu rufende Funktion adressiert.

(b) Der Benutzer kann Skripten direkt über die Adressleiste des Webbrowsers aufrufen. Beispielsweise war für die WebBuilder Laufzeitumgebung das Skript `index.php` als exklusiver Einstiegspunkt vorgesehen, über den mit entsprechender Parametrisierung die vollständige Funktionalität des Systems zu erreichen ist.

Als interner Ablauf werden die Funktionsausführungen bezeichnet, die durch eine Benutzerinteraktion oder einen Aufruf von außen ausgelöst werden.

Die Schnittstelle eines Blockes ergibt sich vereinfacht aus denjenigen Funktionen, die von außerhalb des Blockes aufgerufen werden können.

4. Für jeden Block wurde zunächst untersucht, wie ein Benutzer oder andere Programmteile mit diesem interagieren können und welche Programmteile als Reaktion auf Benutzereingaben oder Aufrufe von Außen ausgeführt werden.

5. Das Analyse-Verfahren wurde anschließend rekursiv und schrittweise auf die gefundenen Funktionsblöcke angewandt, um Unterblöcke und Details der Implementierung zu identifizieren.

4.3 Ergebnis

WebBuilder wurde durch eine Reihe unterschiedlicher Faktoren zu einem Legacy-System. Nicht zuletzt waren auch organisatorische Gründe und das Umfeld verantwortlich. Drei typische Symptome, die das Verstehen und die Änderbarkeit negativ beeinflussen, wurden bei der Analyse besonders deutlich:

- ungeeignet Wahl von Bezeichnernamen
- Strukturverlust insbesondere bei Inklusionen
- mangelhafte Dokumentation

Diese technischen Qualitätsmängel werden im Folgenden genauer erläutert.

4.3.1 Bezeichner

Eine weit verbreitete Empfehlung für Bezeichner in Quellcode, z. B. Variablen-, Funktions- oder Parameternamen, lautet, „sprechende“ Namen zu verwenden, aus denen möglichst präzise und eindeutig auf die Bedeutung des bezeichneten Elements geschlossen werden kann [1, 9] Da Programmverstehen für einen großen Teil der Aufwände in der Wartung verantwortlich ist und ca. 70% eines Programmes frei definierbare Namen sind, sind sinnvolle und verständliche Bezeichner eine wichtige Voraussetzung für Wartbarkeit und zur Vermeidung der Legacy-Problematik.

In WebBuilder wurden viele Bezeichner an den Produktnamen angelehnt und spiegeln nicht präzise die Bedeutung des bezeichneten Elements wider. Oft wurden auch Abkürzungengewählt, die kryptisch wirken und keine Beziehung zur Bedeutung der Variable erkennen lassen.

```
function imp_index_bound_parent(
  $index, $idds, $parent_index,
  $bdr, $register, $webBuilderData
)
```

In dem Programm-Fragment oben wird eine Funktion definiert, deren Namen vermuten lässt, dass es um einen Index geht, der an eine übergeordnete Datenstruktur gebunden ist. Über die Bedeutung der Parameter kann spekuliert werden:

- `$webBuilderData` könnte ein Datencontainer sein.
- Die Namen `$index` und `$parent_index` lassen vermuten, dass es um zwei voneinander abhängige Indizes geht.
- Aus den Bezeichnern `$idds` und `$bdr` lässt sich kein Zweck ableiten.
- Durch `$register` wird nahe gelegt, dass es um indizierte Register geht.

Offensichtlich ist aus den Bezeichnern nicht klar ersichtlich, was durch die Funktion implementiert wird, und welche Aufgabe welcher Parameter hat. Um das herauszufinden, muss der Funktionskörper und ggf. rekursiv alle verwendeten Funktionen und Variablen gelesen und verstanden werden, wodurch der Aufwand für das Verstehen des Programmes erheblich erhöht wird!

4.3.2 Inklusionsstruktur

Eine Voraussetzung für das Verstehen eines umfangreichen Systems ist die Kenntnis der statischen Struktur, die einen Überblick über prinzipielle Zusammenhänge und Sichtbarkeiten geben sollte. Im Falle von PHP ist von Interesse, welches PHP „Skript“ (eine PHP-Datei) Funktionen aus welchen anderen Skripten potentiell aufrufen kann (Sichtbarkeit) und ggf. auch tatsächlich entsprechende Aufrufe enthält.

Um in PHP aus einem Skript `Test` eine Funktion `f`, die in einem anderen Skript `SkriptName` definiert ist, aufrufen zu können, muss das Skript `SkriptName` mit Hilfe einer `include($SkriptName)` oder `include_once($SkriptName)`-Anweisung[6] in `Test` eingebunden werden. Der Parameter `$SkriptName` bestimmt den Namen des einzubindenden Skripts im Dateisystem. Dem einbindenden Skript `Test` werden alle Klassen, Funktionen oder Variablen, die in dem eingebundenen Skript öffentlich definiert sind, zur Verfügung gestellt. Durch Inklusionsanweisungen entsteht eine *Inklusionsstruktur*, die einen Großteil der statischen Struktur einer PHP Anwendung festlegt.

Da die Architektur von WebBuilder weder dokumentiert noch den Entwicklern bekannt war, wurde an der Technischen Universität München ein Systementwicklungsprojekt zur Analyse der Inklusionsstruktur von WebBuilder in Auftrag gegeben. Bereits diese Analyse erwies sich als schwierig, da der PHP-Inklusionsmechanismus auf verschiedenen Arten genutzt wurde.

Dynamische Inklusion Da es sich bei `include()`[7] in PHP um eine Funktion und nicht um eine spezielle Anweisung, wie z. B. die `import`-Direktive in Java, handelt können auch Variablen, die erst zur Laufzeit ausgewertet werden, als Include-Parameter verwendet werden. Dies ermöglicht quasi ein dynamisches Laden von PHP Dateien zur Laufzeit, das u. a. auch zur Erweiterung des Programms während seiner Ausführung genutzt werden kann.

Allerdings kann durch dieses Vorgehen die Inklusionsstruktur nicht anhand des Programmtextes erschlossen werden bzw. die statische Struktur existiert de facto kaum. Um die statische Struktur dennoch annäherungsweise verstehen zu können, muss bei einer Analyse der wahrscheinliche oder mögliche Wert der Variablen zur Ausführungszeit anhand des statischen Kontroll- und Datenflusses ermittelt werden. Dies verursacht erheblichen Aufwand.

In WebBuilder wurde diese Technik intensiv genutzt. In vielen Fällen hat sich die Variable jedoch als Konstante erwiesen. Vorteile ergeben sich hieraus lediglich durch die Wiederverwendung der „Variablen“ an mehreren Stellen.

Indirektes Laden Es wurde eine Funktion definiert, die selbst ausschließlich `include()` aufruft:

```
load_Source(INCLUDE_500098);
...
function load_source($param){
  include($param);
}
```

`load_source` ist überflüssig. Sie bietet keinen Mehrwert gegenüber einem direkten Aufruf von `include()`, erschwert jedoch die Programmanalyse, da ein weitere Funktion Inklusionsbeziehungen verursachen kann und berücksichtigt werden muss.

On-Demand Inklusion Es wurde ein Mechanismus geschaffen, der es ermöglicht, einzelne Funktionen erst dann zu laden, wenn sie tatsächlich ausgeführt werden. Dies sorgt für hohe Flexibilität, da a) im laufenden Betrieb Änderungen am Portal vorgenommen und Nutzern unmittelbar zur Verfügung gestellt und b) unbenötigte Skripten nicht in den Speicher geladen werden.

Diese Technik wurde mit Hilfe der Funktion `&loadFunctionPackage()` realisiert, mit deren Hilfe PHP-Skripten eingebunden werden, die ein Array mit dem Namen `desc_func` enthalten. `desc_func` definiert die Namen verfügbarer Funktionen, die der Nutzer des Skripts aufrufen kann.

Die nutzbaren Funktionen sind jedoch selbst nicht in dem selben Skript implementiert. Die Funktionskörper der in `desc_func` genannten Funktionen sind lediglich Stellvertreter und enthalten eine `include()`-Anweisung, mit der das

[6] `include_once($SkriptName)` stellt die genau einmalige Inklusion sicher liefert jedoch schwächere Performanz.

[7] Auf die separate Nennung der „once“ Variante wird im Folgenden verzichtet. Alle Aussagen gelten analog.

Skript mit der tatsächlichen Implementierung der gewünschten Funktion erst dann eingebunden wird, wenn die Funktion aufgerufen wird.

Pfadabhängige Inklusion Bei Aufruf der selbst geschriebenen Funktion `&import($package)` werden alle PHP-Skripten inkludiert, die in dem Verzeichnis das Dateisystems liegen, das sich aus `$package` relativ zum Pfad des einbindenden Skripts ergibt.

Weitere Inklusionsarten

- Die WebBuilder Funktion `includeScript()` nimmt JavaScript Dateien in die PHP Umgebung auf.
- Mit `&createPHPObject()` werden Skripten eingebunden und von den darin enthaltenen Klassen unmittelbar Instanzen erzeugt, die dem Aufrufer zur Verfügung gestellt werden.

Um trotz dieser Erschwernisse eine Aussage über die statische Struktur von WebBuilder treffen zu können, wurden zunächst alle Quelldateien von WebBuilder auf möglich Inklusionen untersucht. Die gefundenen Inklusionen wurden manuell aufwändig weiter analysiert und anschließend in eine Adjazenzmatrix übertragen. Aus dieser konnte mit Hilfe des Werkzeugs *GraphViz*[8] ein Strukturgraph erstellt werden, für den Abbildung 4 einen Ausschnitt zeigt. Es konnten vier Subsysteme identifiziert werden, die sich durch einen besonders engen Zusammenhang der beteiligten Quellen auszeichnen. Diese Subsysteme wurden als Basis für die Top-Down-Analyse verwendet.

Zudem wurden zahlreich überflüssige Skripten gefunden, die von keinem anderen Skript benutzt wurden. Dabei handelt es sich einerseits um einzelne Skripten und andererseits um Mengen von Skripten, die sich zwar gegenseitig referenzierten, jedoch von außen nicht aufgerufen wurden.

Insgesamt stellte sich die Inklusionsstruktur als komplex dar bzw. wurde es deutlich, dass kaum Struktur besteht; mit entsprechenden Konsequenzen für den Änderungsaufwand und die Fehleranfälligkeit. Ursache dieses Strukturverlusts war die Vorstellung, Komponenten zur Laufzeit verändern und anpassen zu können, was de facto tatsächlich weder intensiv genutzt wurde noch notwendig gewesen wäre. Zusätzlich hatte sich diese Vorstellungen über die Zeit zwar mehrfach verändert, WebBuilder wurde jedoch nicht konsequent angepasst. Auf diese Weise haben sich die verschiedenen Mechanismen angesammelt und vermischt, was enorme Mehraufwände für das Verstehen und die Pflege des Programmes verursacht hat.

[8] `http://www.graphviz.org`

4.3.3 Dokumentation

Die Dokumentation eines Systems ist nur dann von Nutzen, wenn sie grundlegende Anforderungen erfüllt. Unter anderem sind dies:

Korrekt Wichtige Sachverhalte sollten korrekt beschrieben werden.

Konsistent Die gesamte Dokumentation sollte in sich und gegenüber der Implementierung widerspruchsfrei sein.

Strukturiert Um ein leichtes Auffinden von Information zu ermöglichen, sollte ein Gesamtverzeichnis aller Dokumente existieren, aus dem hervorgeht, welche Information wo zu finden ist. Außerdem sollte jederzeit klar sein, wo zu einem Thema verwandte Information zu finden ist.

Redundanzfrei Um die Pflege der Dokumentation zu erleichtern, sollte jede Informationen nicht an verschiedenen Orten wiederholt werden.

Die Dokumentation von WebBuilder, erfüllte keine dieser Eigenschaften. Eine Entwicklerdokumentation mit der Beschreibung der internen Struktur oder Abläufe existierte nicht. Ebenso wurden kaum Quellcodekommentare genutzt. Die wenigen vorhandenen Quellcodekommentare sind meist in Russisch[9] und damit für die Mitarbeiter des deutschen Unternehmens nutzlos.

Es stand lediglich eine erwähnenswerte Anwenderdokumentation zur Verfügung, die jedoch unter sprachlichen Defiziten und mangelhafter Struktur litt. In manchen Teilen enthielt sie Information, die für einen Anwender unbedeutend ist und in eine Entwicklerdokumentation eingeordnet werden

[9] Der Name der verwendeten Sprache wurde für dieses Dokument geändert.

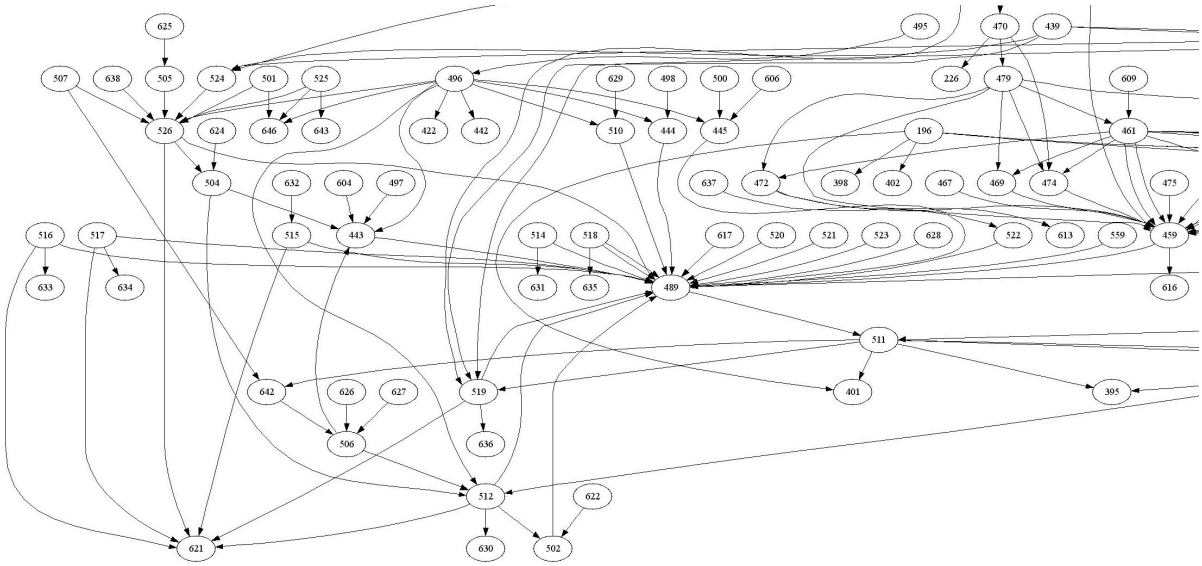

Abbildung 4: Inklusionsgraph (Ausschnitt)

müsste. Zusätzlich waren Sachverhalte mehrfach dokumentiert, andere fehlten.

Zu den vorhandenen Dokumente wurden keine Metainformationen wie z. B. Historie, Informationen über Autoren oder Kurzzusammenfassungen zur Verfügung gestellt. Weiterhin fehlte eine Übersicht der verfügbaren Dokumente und eine Übersicht, wo welche Information abgelegt ist. Für jedes einzelne Dokument war unklar, wie es im Verhältnis zu anderen Dokumenten einzuordnen ist, d. h. wo verwandte Information zu finden wäre.

Diese mangelhafte Dokumentation hat die vorhandenen Probleme verstärkt. Wären z. B. die Bezeichner durch Quellcodekommentare erklärt gewesen, hätte dies die Problematik der ungünstigen Benennungen abgeschwächt. Durch die niedrige Qualität der Dokumentation war zudem das Vertrauen in die vorhandene Information gering, so dass auch diese letztendlich nutzlos war. Die Dokumentation hatte weitere negative Konsequenzen. Durch die Quellcodekommentare in der Fremdsprache kam der Verdacht auf, dass die Entwickler absichtlich Informationen verbergen und die Weiterentwicklung gezielt behindern würden. In Kombination mit anderen Schwierigkeiten hatte dies zur Folge, dass das Team in Deutschland sämtlichen Arbeitsergebnissen der ursprünglichen Entwickler mit großem Misstrauen begegnete und absichtliche Fallen auch im Programm selbst vermuten musste. Bei der weiteren Programmanalyse musste deshalb stets zusätzlich der Frage nachgegangen werden, ob ein komplizierter Konstrukt tatsächlich Sinn hat oder lediglich der Verschleierung dient.

4.4 Bewertung

Trotz dieser vielen Hindernisse konnte das Reverse Engineering erfolgreich abgeschlossen und ein Verständnis für die Zusammenhänge und Abläufe im System gewonnen werden, so dass eine Weiterentwicklung möglich wurde.

Der Aufwand hierfür war jedoch enorm und hat die Existenz des betroffenen Unternehmens bedroht. Für alle Beteiligten war überraschend, dass eine Anwendung, bei der die Entwicklung gerade abgeschlossen war und gerade Marktreife erreicht wurde, bereits zu einem Legacy-System wurde.

Der Ursprung der aufgezeigten Mängel war das geringe technische Verständnis der Firmenleitung des deutschen Auftraggebers. Es wurden weder eine Vorgehensweise für die Entwicklung festgelegt noch Vorgaben zur Qualitätskontrolle gemacht. Dementsprechend wurde das System in einem kreativ-chaotischen bzw. anarchischen Umfeld mit

entsprechendem Ergebnis entwickelt. Durch eine kontinuierliche Kontrolle des Programms und seiner Dokumentation hätten viele Probleme vermieden werden können.

Eine weitere wichtige Erkenntnis dieses Fallbeispiels ist, dass bereits einzelne Qualitätsmängel das Programmverstehen nahezu unmöglich machen können. Die Benennung von Variablen ist sicherlich ein vergleichsweise unwichtiger Faktor, wenn es um die von Benutzern empfundene Qualität geht und wird in der Regel deshalb auch nicht geprüft. Für die Übernahme eines Software-Produkts oder ein Reengineering ist sie jedoch von zentraler Bedeutung.

5 Weitere junge Legacies

In zwei weiteren Beispielen wird deutlich, dass unsere Erfahrung mit WebBuilder kein Einzelfall sind und keineswegs davon abhängt, wo die Entwicklung stattfindet.

5.1 ETY

Das System ETY wurde bei einem deutschen Einzelhandelsunternehmen entwickelt und dient der Analyse von Kundendaten. ETY wird an verschiedenen Standorten in verschiedenen Zeitzonen und Sprachen verwendet. Die technische Basis von ETY ist Microsoft Visual Basic für den Client und Microsoft SQL-Server zur Haltung der Daten.

5.1.1 Vorgefundener Zustand

ETY litt bereits unmittelbar bei Produktivsetzung unter Akzeptanz- und Performanzproblemen sowie ungewöhnlich hohen Aufwänden bei der Beseitigung von Fehlern. U. a. war die Aufbereitungsdauer der Analysedaten mit mehr als zwei Stunden für die Nutzer inakzeptabel. Da sowohl der Betrieb als auch die Weiterentwicklung von ETY sehr teuer war, war ETY bereits bei seiner Einführung ein Legacy-System. Zur Verbesserung dieser Situation wurde ein Reengineering des Systems in Auftrag gegeben.

5.1.2 Analyse

In Rahmen dieses Reengineerings wurden die technischen Ursachen der beobachteten Symptome deutlich:

Datenstruktur In einzelnen Datenbanktabellen waren über 70 (!) Spalten zu finden, wobei die Spalten teilweise mit „Field1“, „Field2“, ... benannt waren.

Für Datenbanktabellen gilt das Prinzip, dass in einer Tabelle nur die Attribute gespeichert werden sollen, die zu dem Konzept der Tabelle gehören. In diesem Fall war dies nicht gegeben.

Zusätzlich gilt für die Bezeichner in Datenbanken das gleiche wie für Bezeichner in Programmen: werden „sprechende Namen“ verwendet, ist es bedeutend einfacher, das Datenmodell zu verstehen.

Programmtext Im Programmtext der Anwendung fehlten Kommentare und eine sinnvolle Formatierung des Programmtexts. Die fehlenden Formatierung machte das Lesen des Codes schwierig und erfordert hohe Konzentration.

Bezeichner Auch in diesem System wurde viele Bezeichner ungünstig gewählt. Die folgenden Code-Fragmente, die unmittelbar aus ETY entnommen wurden, sprechen für sich:

```
Function MSbloed()
  scripted_readData
End Function

Function ig_ig_1(ig, ig_1)
  If IsNull(ig) Or IsNull(ig_1) Then
    Exit Function
  End If
  ig_ig_1 = (ig - ig_1)
End Function

Function banane(x As String)
  If x = reps("rep").RNAME Then
    banane = True
  Else
    banane = False
  End If
End Function
```

Redundanz Insgesamt war der Quellcode zu 60% redundant, d. h. bei fast $\frac{2}{3}$ des Programms handelte es sich um überflüssige Kopien anderer Programmfragmente. Anstelle

der Parametrisierung von Funktionen wurden vorhandene Programmfragmente an verschiedene Stellen kopiert und minimal modifiziert.

Redundante Programme erhöhen den Aufwand für die Änderung, Erweiterung und den Test eines Programmes. Darüber hinaus verursacht Redundanz häufig Fehler nach Änderung, da die Existenz von Kopien des zu ändernden Programmfragmentes in der Regel nicht bekannt ist!

Mangelnde Kenntnis der Umgebung Die Entwickler von ETY kannten die verwendete Plattform nur partiell. Aus diesem Grund wurden Standard-Funktionen, welche die Plattform bereitstellen würde, neu erfunden und überflüssig implementiert. Beispielsweise wurde für das Runden von Gleitkommazahlen eine neue Funktion implementiert, die gegenüber der Standardfunktion `round()` keine Vorteile bietet.

Dieses Vorgehen ist unnötig, setzt die Effizienz des Systems herab und erhöht den Wartungsaufwand.

Steuerungs-„Hack“ Zur Zeitsteuerung von ETY wurde in einem MS-Excel Dokument eine Funktion implementiert, die in periodischen Abständen ETY startet (s.u.).

```
Sub Control()
  Application.OnTime TimeValue _
    ("15:23:00"), "import"
  ...
End Sub

Sub import()
  Dim fnameetx, fpfadetx As String
  Dim ergebnis
  fnameetx = "ETY.exe"
  fpfadetx = "C:\"
  ergebnis = Shell _
    (fpfadetx & fnameetx & _
    "␣/autoimport", 1)
End Sub
```

Die Verwendung eines Excel-Dokuments zur Steuerung eines periodischen Zeitablaufs ist unnötig, aufwändig und instabil. Periodische Abläufe können wesentlich einfacher durch das Betriebssystem gesteuert werden.

Ortsabhängigkeit Zahlen, Währungen und Zeitangaben werden in verschiedenen geographischen Regionen unterschiedlich dargestellt. So wird in Deutschland als Tausendertrennzeichen ein Punkt verwendet, im englischen Sprachraum ein Komma.

Das zu verwendende Format wurde in ETY dadurch ermittelt, dass ein MS-Excel Dokument erzeugt, eine Gleitkommazahl eingetragen und in eine Zeichenkette umgewandelt und anschließend geprüft wurde, welches Trennzeichen in der Zeichenkette wird. Dieses Zeichen wurde dann in einer Datenbanktabelle gespeichert und an allen betroffenen Stellen aus der Datenbank gelesen.

Auch dieses Vorgehen ist unnötig. Visual Basic besitzt bereits Funktionalitäten für den Umgang mit regional unterschiedlichen Formaten und zur korrekten ortsabhängigen Darstellung und Bearbeitung von Gleitkommazahlen und anderen Daten.

Betriebssystemspezifika Werden Texte aus Unix-Systemen übernommen, so muss das Zeilenende in das MS-Windows Format konvertiert werden. Hierfür wurde folgende Prozedur implementiert:

```
...
'Vorbereitende Deklaration
'von Variablen
Dim buffer As String
Dim char As String
Open filein For Input As #1
Open fileout For Output As #2
buffer = ""
'Loop bis Dateiende
Do While Not EOF(1)
  'Buchstabe einlesen
  MyChar = Input(1, #1)
  'Buchstabe != Zeilende
  While char <> Chr(10)
        'Buchstabe in Puffer
        buffer = buffer & char
    'nächsten Buchtaben lesen
    char = Input(1, #1)
        'Schleife
  Wend
  'Buchstabe ist Zeilenende
  'Puffer in neue Datei schreiben
  Print #2, buffer
  'neue Zeile anfangen
```

```
    char = ""
    buffer = ""
Loop
...
```

Der Mangel dieser Prozedur ist, dass jeder Buchstabe einzeln gelesen und verarbeitet. Dieses Vorgehen ist erheblich langsamer als blockweises Lesen und Verarbeiten. Ganz besonders, wenn die Eingabe-Datei, wie im Falle von ETY, auf einem Netzlaufwerk abgelegt ist.

Für einer effiziente Implementierung muss die Datei auf einen lokalen Speicher kopiert und blockweise verarbeitet werden. Allein durch diese Optimierung konnte die Verarbeitungsdauer bereits halbiert werden!

Die Beseitigung dieser Probleme gestaltete sich als aufwändig, konnte jedoch erfolgreich abgeschlossen werden. Die Gesamtdauer für die Aufbereitung der Analysedaten konnte ohne Änderung der Umgebung von über zwei Stunden auf knapp vier Minuten gesenkt werden. Der Umfang des Programmes ist bei gestiegener Funktionalität auf ca. $\frac{1}{3}$ des ursprünglichen Umfangs gesunken. ETY ist heute erfolgreich im Einsatz und wird kontinuierlich weiterentwickelt.

5.1.3 Ursachenforschung

„A fool with a tool is still a fool."

Tom de Marco

Ein erheblicher Teil der identifizierten Probleme rührt daher, dass die beteiligten Entwickler unzureichende Kenntnis der verwendeten Plattform besaßen. Der Einsatz einer geeigneten Plattform garantiert noch kein hochwertiges System. Die richtige Plattform muss auch richtig verwendet werden. Grundlagen der Informatik, wie z. B. Datenbank-Normalformen, waren den Entwicklern offensichtlich ebenfalls unbekannt, was zu den beschriebenen Mängeln führte. Zusätzlich wurden auch in diesem System Grundprinzipien, wie „sprechende" Namen für Bezeichner, missachtet.

Ähnlich wie bei WebBuilder führten wieder eine Reihe vermeidbarer Mängel zu einem System, das kaum nutzbar war und nicht mehr weiter entwickelt werden konnte. Auch ETY ist nicht gealtert, sondern war schon bereits mit Abschluss der Entwicklung eine Legacy-Anwendung mit allen damit verbundenen Problemen. Analog zu WebBuilder hätte durch Vorgaben und Qualitätskontrolle in der Entwicklung ein junges Legacy-System vermieden werden können.

5.2 QTS

Im Projekt QTS wurde für einen europäischen Versicherungskonzern eine mittel-große web-basierte Anwendung auf Basis von PHP realisiert.

Bereits die Entwicklung des Systems erwies sich als unerwartet aufwändig und die gestellten Anforderungen konnten nur schwer erfüllt werden. Bereits unmittelbar nach Auslieferung hatte das System mit mangelnder Akzeptanz, hohen Wartungskosten und einer hohen Fehlerrate zu kämpfen.

5.2.1 Analyse

Auch bei diesem System sind die Ursachen der Probleme in erster Linie in Qualitätsmängel zu suchen, die bereits früh im Entwicklungszyklus entstanden sind.

Struktur Die Anwendung wurde anhand technischer Kriterien strukturiert. Dies zeigt sich u. a. an der Organisation der verschiedenen PHP Dateien des Projekts in wenige Unterverzeichnissen mit technischen Namen, wie z. B. `scripts`.

In der Regel wird jedoch empfohlen, Systeme in fachliche Komponenten zu gliedern und diese dann nach technischen Kriterien zu strukturieren. Der Sinn der Modularisierung eines Software-Systems liegt darin, Einheiten zu schaffen, die weitestgehend unabhängig voneinander sind und möglichst unabhängig voneinander (weiter-)entwickelt werden können. Die Separierung nach fachlichen Kriterien ist hierfür bis zu einem gewissen technischen Niveau empfehlenswert.

Sprachmix QTS wurde in PHP geschrieben. Die Anteile der grafischen Benuztungsoberfläche wurden in HTML implementiert, wobei der HTML Markup direkt in die PHP Skripten eingebettet wurde, ohne zu versuchen,

Präsentation und Logik, wie in einer MVC-Architektur üblich, zu trennen.

Wie im Detail in Abschnitt 3 beschrieben wird, steigt hierdurch der Wartungsaufwand erheblich an.

Datenbankschema In der Datenbank wurden Daten redundant abgelegt, wodurch UPDATE, INSERT und DELETE Operationen verlangsamt und fehlerträchtig wurden. Dies hätte durch ein normalisiertes Datenbankmodell leicht vermieden werden können.

Datenbankzugriff Es wurde keine separate Persistenz- oder Datenbankzugriffsschicht gebildet. Stattdessen fanden sich SQL-Statements im Programmtext teilweise redundant und über alle PHP-Skripten verteilt.

Dies hat u. a. zur Konsequenz, dass bei Änderungen an der Datenbank prinzipiell jedes PHP-Skript darauf untersucht werden muss, ob es die geänderten Datenbankelemente verwendet und wie sich die Änderungen auf die in dem Skript implementierte Logik auswirken. Durch die Einführung einer Datenbankzugriffsschicht als Abstraktion wären zumindest einfache Änderungen auf eine Anpassung der Datenbankzugriffsschicht beschränkt gewesen.

Internationalisierung Meldungstexte wurden als Literale direkt im HTML Code abgelegt. Dass dies ein Problem darstellt, war den Entwicklern selbst bewusst:

> „//TOM: das derzeitige Texthandling ist Quatsch. . . “
>
> (Quellcodekommentar)

Eine Einbindung von Textliteralen direkt in den Code einer Anwendung erschwert die Pflege der Texte und macht eine Internationalisierung nahezu unmöglich. Zusätzlich kann so auch kein Werkzeug zur Bearbeitung der Texte eingesetzt werden.

Flexibilität QTS referenzierte Pfade im Dateisystem des Servers.

Daraus folgt, dass das Programm angepasst werden muss, falls sich die Verzeichnisstruktur auf dem Server ändert oder QTS auf einem anderen Server installiert werden soll.

Redundanz Viele PHP-Skripten implementierten eine einfache Autorisierung des Benutzers, die stets vor der eigentlichen Funktionalität des Skriptes durchgeführt wurde und sich kaum voneinander unterschieden.

Diese Implementierung ist äußerst aufwändig. Ändert sich der Autorisierungsmechanismus, ist es notwendig, sämtliche Skripten der Anwendung anzupassen. Besser wäre gewesen, eine zentrale Stelle zur Authentifizierung und Autorisierung zu schaffen, die dann von den einzelnen Skripten verwendet werden kann. Da dies nicht der Fall war, überrascht es nicht, dass in manchen Skripten die Autorisierung fehlerhaft durchgeführt oder gänzlich vergessen wurde.

Das Fallbeispiel QTS zeigt, wie Systeme durch Entwurfsentscheidungen, die früh im Entwicklungszyklus getroffen werden, rasch zu Legacy entarten können. Im Falle von QTS war dies vor allem die mangelnde Separierung von Schichten (GUI, Logik, Datenbank) sowie ein unzureichend normalisiertes Datenmodell.

6 Qualitätsmanagement

Wie die Fallbeispiele gezeigt haben, führt mangelnde Qualität bei der Entwicklung und Pflege von Software zu den typischen Legacy Symptomen. Das Alter einer Anwendung spielt dafür eine untergeordnete Rolle. Um Legacy-Systeme zu vermeiden ist daher ein umfassendes Qualitätsmanagement mit einer kontinuierlichen Qualitätskontrolle notwendig. Auf diese Weise können Probleme frühzeitig erkannt und vorbeugende Maßnahmen rechtzeitig ergriffen werden. Wie die Beispiele gezeigt haben kann bereits ein einzelnes Qualitätsmerkmal die Qualität eines Systems so negativ beeinflussen, dass es seine Zukunftsfähigkeit verliert.

6.1 Qualitätsmodell

Voraussetzung für ein wirksames Qualitätsmanagement ist ein *Qualitätsmodell*, das den abstrakten

Begriff Qualität auf fundierte und möglichst messbare bzw. zumindest objektiv bewertbare Kriterien abbildet. An der Technischen Universität München wurde ein solches Qualitätsmodell (TUM-QM) entwickelt [7,8], dass konkrete Vorschläge für wirksame Kriterien beinhaltet.

TUM-QM unterscheidet zwischen *Aktivitäten*, die an dem Software-System verrichtet werden und *Fakten* über die Situation, in der sich das Software-System befindet, und setzt diese in Beziehung. Zur Situationsbeschreibung gehört neben dem Programm selbst, die Dokumentation, aber auch Aspekte der Organisation, wie der Ausbildungsstand der Mitarbeiter und die Infrastruktur zur Pflege des Systems. Orthogonal hierzu sind Aktivitäten, wie Analyse, Implementierung und Testen von Interesse, da bei diesen Aktivitäten der Aufwand entsteht, der durch erhöhte Qualität — d. .h durch eine günstigere Situation – reduziert werden soll.

6.2 Mittel der Qualitätsprüfung

Bei einer Qualitätsprüfung werden die Fakten der Situation bewertet, wobei drei verschiedene Arten von Fakten zu unterscheiden sind:

- Automatisch durch Werkzeug ermittelbar.
- Manuell zu prüfen.
- Semi-automatisch Prüfung.

Zu den automatisch zu bewertenden Fakten gehören bekannte Software-Metriken, wie Anzahl Zeilen, Tiefe der Vererbungshierarchie, zyklomatische Komplexität, etc. Manuell zu untersuchende Fakten sind in der Regel semantischer Natur, wie z. B. die „korrekte“ Verwendung von Datenstrukturen. Redundanz kann in Programmen mit einem Werkzeug identifiziert werden, falls es sich um textuelle Kopien handelt, inhaltliche Redundanz erfordert jedoch manuelle Inspektion. Zur Ermittlung der Redundanz ist deshalb eine semiautomatische Prüfung notwendig.

6.3 Hierarchische Struktur

Sowohl Fakten als auch Aktivitäten sind hierarchisch gegliedert, so dass sowohl in Bezug auf den Aufwand als auch für die Situationsbeschreibung

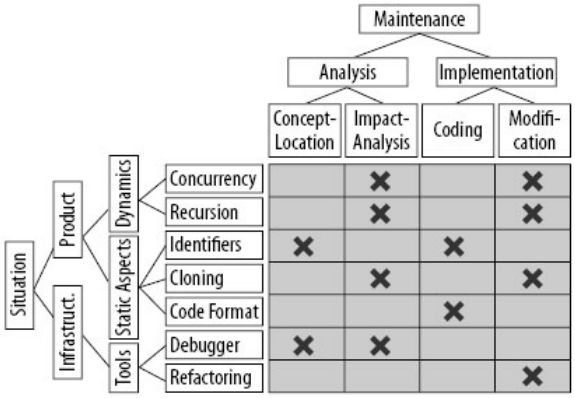

Abbildung 5: Ausschnitt aus der Qualitätsmatrix

separate Profile gebildet werden können. Die Aktivität „Wartung“ beinhaltet beispielsweise „Analyse“ und „Modifikation“, der Fakt „Produkteigenschaften“ teilt sich in dynamische und statische Aspekte auf.

TUM-QM setzt Fakten und Aktivitäten mit Hilfe einer zweidimensionalen Matrix in Beziehung. Wird eine Aktivität von einem Fakt beeinflusst, so wird der entsprechende Matrixeintrag markiert und eine Erklärung zu dieser Wechselwirkung angegeben (siehe Abbildung 5). Zusätzlich können die einzelnen Fakten, die eine Aktivität beeinflussen, gewichtet werden.

6.4 Tailoring

Welche Aktivitäten tatsächlich in der Matrix auftauchen, hängt von dem Software-Entwicklungs- oder Wartungs-Prozess ab, der auf das Software-System künftig angewendet werden soll. Beispielsweise würde für das Reverse Engineering aus Abschnitt 4 besonderer Wert auf die Spalte der Aktivität *Concept Location* gelegt werden, während für das System ETY in erster Linie *Modification* von Bedeutung wäre.

TUM-QM vermeidet mit diesem Schema Pauschalkriterien, wie „keine Prozedur sollte länger als 50 Zeilen sein“, und liefert stattdessen eine exakte Erklärung welche Auswirkung ein Merkmal eines Systems auf den künftigen Pflegeaufwand hat. Die Kriterien ergeben sich damit aus der Kombination eines Faktums über die Situation und dem Optimierungsziel. TUM-QM, von dem hier nur ein

kleiner Ausschnitt dargestellt wurde, bietet derzeit einen Katalog von ca. 200 Fakten und ca. 25 Aktivitäten an [8].

7 Verwandte Arbeiten

Neben den bereits zitierten Arbeiten zum Thema Legacy-Systeme wurde diese Problemstellung in einer Reihe weiterer Arbeiten untersucht. Als Beispiel seien die Bewertung von Legacy-Systemen nach technischen, organisatorischen und betriebswirtschaftlichen Gesichtspunkten in [25], aber auch Arbeiten im Zusammenhang mit der Migration von Legacy-Daten [4] genannt. Eine Übersicht zu Literatur über die Migration von Legacy-Systemen ist in [18] zu finden. In der Regel wird Legacy jedoch stets mit dem Alter eines Systems assoziiert. Den Autoren dieses Erfahrungsberichtes sind aus dem Umfeld Software-Maintenance keine Arbeiten bekannt, welche die Legacy-Problematik unabhängig von historischen Betrachtungen diskutieren.

Im Gegensatz hierzu gibt es eine Reihe von Arbeiten, die sich dem Thema Modellierung von Qualität widmen. Die wichtigsten Arbeiten hierzu, die teilweise als Grundlage für TUM-QM dienten, werden im Folgenden skizziert.

7.1 McCall

1977 schlug McCall ein hierarchisches Modell vor, das auf breite Akzeptanz stieß [21]. Dieses Modell von Software-Qualität basiert auf drei Arten von Qualitätsmerkmalen:

Factors beschreiben die externe Sicht auf die Software aus der Perspektive des Nutzers.

Criteria beschreiben die interne Sicht der Software.

Metrics definieren Skalen und Methoden für Messungen.

McCall definierte elf Faktoren (u. a. Funktionalität, Zuverlässigkeit und Performanz) die auf 25 Kriterien abgebildet werden. Die zugehörigen Metriken sind als Fragen formuliert, wobei die Fragestellungen teilweise subjektive Antworten zulassen. Zusätzlich werden zwei Unterscheidungen eingeführt:

- Jedes Merkmal ist einer „product activity“ zugeordnet (operation, revision oder transition), da unterschiedliche Aktivitäten unterschiedliche Qualitätsbegriffe erfordern.
- Jedem Merkmal sind Anwendungsklassen zugeordnet (z. B. eingebettete Systeme), wodurch eine Gewichtung der Merkmale nach Anwendungssituation vorgenommen wird.

Dieser Ansatz wurde von zahlreiche späteren Qualitätsmodellen, wie ISO 9126 [16], aufgegriffen und verfeinert.

7.2 Boehm

Das Modell von Boehm sieht analog zu McCall ebenfalls elf top-level Qualitätsmerkmale vor [6]. Für jedes Merkmal sind Metriken definiert, wobei einige Metriken mehreren Merkmalen zugeordnet sind. Durch die Metriken ergibt sich eine Ordnung der verschiedenen Merkmale: enthält ein Merkmal A sämtliche Metriken, die Merkmal B zugeordnet sind, so impliziert Merkmal A Merkmal B. Hieraus ergibt sich eine partielle Ordnung über den Qualitätsmerkmalen.

Zusätzlich werden drei Faktoren für die Qualität von Software herausgestellt:

- Wie gut kann ein Produkt in seinem momentanen Zustand benutzt werden?
- Wie einfach kann es angepasst werden?
- Ist es möglich das Produkt weiter zu verwenden, falls sich die Umgebung verändert?

7.3 Weitere Modelle

Weitere hierarchische Qualitätsmodellen sind Unternehmensstandards [12] oder der internationale Standard ISO 9126 [16] sowie [10]. Das gemeinsame Defizit dieser Modelle ist einerseits die mangelnde Vollständigkeit der Kriterien und andererseits eine ausreichende rationale Begründungen [17], warum ein Merkmal durch ein Kriterium beeinflusst wird und wie der Grad der Beeinflussung je nach aktueller Projektsituation angepasst werden kann. Dies schränkt die Eignung dieser Modelle für die Praxis ein. Das TUM-QM erlaubt

mit seiner Separierung von Situation und Aktivitäten systematisch und werkzeuggestützt ein höhere Abdeckung relevanter Kriterien zu erreichen und liefert an den Schnittpunkten der beiden Dimensionen präzise Erklärungen.

8 Fazit

Die in diesem Erfahrungsbericht vorgestellten Anwendungen belegen, dass Legacy keine Frage des Alters sind. „Junge“ Systeme, die unter engen Zeitvorgaben, knappen Budgets und ohne ausreichende Qualifikation erstellt werden, entarten oft schon während ihrer Entwicklung zu Legacy-Systemen, deren Weiterentwicklung überproportional teuer und fehlerträchtig wird. Die Fallbespiele zeigen deutlich, dass die wahren Ursachen dieser Symptome schwerwiegende Qualitätsmängel sind. Dabei sei betont, dass es sich bei den hier geschilderten Fällen nicht um Einzelfälle handelt sondern die Autoren dieses Berichts diese und andere gravierende Mängel in einer Vielzahl kommerziell eingesetzter Software-Systeme vorfinden. Die Konsequenz hieraus sind stets erhebliche Mehrkosten für die Nutzer, eingeschränkte Handlungsfähigkeit, da Änderungen von Geschäftsprozessen nicht zeitnah umgesetzt werden können und das Scheitern von Projekten. Tatsächlich hat es oftmals den Anschein als würden bei der Ablösung vorhandener Systeme lediglich alte durch junge Legacy-Systeme ersetzt.

Aus der Sicht der Autoren ist die Wurzel dieser Probleme das gemeinhin mangelnde Verständnis für Qualität von Software sowie die mangelnde Bereitschaft, Produkt-Qualität im Detail durchzusetzen. Die Fallbeispiele zeigen, dass Qualität im Detail entscheidend ist — s. ungenügende Bezeichnernamen, fehlende Normalisierung von Datenmodellen und mangelnde Kenntnis der Plattform. Jeder dieser Mängel hat katastrophale Auswirkungen und ist bei der Entwicklung leicht zu vermeiden.

Eine wichtige Voraussetzung zur Vermeidung dieser negativen Effekte ist eine präzise Definition des Begriffs Qualität. Hierzu ist ein Qualitätsmodell notwendig, das spezifisch für das zu entwickelnde System objektiv bewertbare Kriterien angibt, deren Relevanz nachvollziehbar begründet und mit dem System fortentwickelt wird. Wird dieses Qualitätsmodell zusätzlich zeitnah und konsequent bei allen Entwicklungs-, Wartungs- und Änderungsarbeiten eingesetzt, so ist die Gefahr der Degenerierung zur Legacy und des Verlusts der getätigten Investitionen sowohl kurz- als auch langfristig gering.

Literatur

[1] Nicolas Anquetil and Timothy Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *CASCON '98*, page 4. IBM Press, 1998.

[2] Marcel Bennicke and Heinrich Rust. Programmverstehen und statische analysetechniken im kontext des reverse engineering und der qualitätssicherung. Technical Report ViSEK/025/D, ViSEK, February 2004.

[3] John Bergey, Dennis Smith, and Nelson Weiderman. Dod legacy system migration guidelines. Technical Report CMU/SEI-99-TN-013, SEI, 1999.

[4] Jesús Bisbal, Deirdre Lawless, Bing Wu, and Jane Grimson. Legacy information systems: Issues and directions. *IEEE Software*, 16(5):103–111, 1999.

[5] Barry Boehm. *Software Engineering Economics*. Prentice-Hall, 1981. incomplete.

[6] Barry W. Boehm, John R. Brown, Hans Kaspar, Myron Lipow, Gordon J. Macleod, and Michael J. Merrit. *Characteristics of Software Quality*. North-Holland, 1978.

[7] Manfred Broy, Florian Deissenboeck, and Markus Pizka. Demystifying maintainability. In *Proc. of the 4th Workshop on Software Quality, Shanghai, P.R.C.* ACM Press, 2006.

[8] Florian Deißenböck and Markus Pizka. Projekt PQL qualitätsmodell. Technical report, Technische Universität München, Boltzmannstr. 3 - 85748 Garching, March 2006.

[9] Florian Deissenboeck and Markus Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, September 2006.

[10] R. Geoff Dromey. A model for software product quality. *IEEE Trans. Softw. Eng.*, 21(2):146–162, 1995.

[11] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J.S. Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):12, January 2001.

[12] Robert B. Grady and Deborah L. Caswell. *Software Metrics: Establishing a Company-wide Program.* Prentice-Hall, Upper Saddle River, NJ, USA,, 1987.

[13] PHP Group. Php programming language. World Wide Web, July 2006. http://www.php.net/.

[14] Carl S. Hartzman and Charles F. Austin. Maintenance productivity: observations based on an experience in a large system environment. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering*, volume 1, pages 138 – 170, Toronto, Canada, 1993. IBM Centre for Advanced Studies Conference, IBM Press.

[15] The Standish Group International Inc. Chaos: A recipe for success, 1999.

[16] ISO/IEC. Software engineering – product quality – part 1: Quality model. Technical Report 9126-1, ISO/IEC, June 2001.

[17] Barbara Kitchenham and Shari Lawrence Pfleeger. Software quality: The elusive target. *IEEE Softw.*, 13(1):12–21, 1996.

[18] D. Lawless., J. Bisbal, B. Wu, J. Grimson, and V. Wade. A survey of research into legacy system migration. Technical report, Trinity College Dublin, 1997.

[19] B. P. Lientz, P. Bennet, E. B. Swanson, and E. Burton. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations.* Addison Wesley, Reading, 1980.

[20] Edwin Martin. What i don't like about php. World Wide Web, April 2006. http://www.bitstorm.org/edwin/en/php/.

[21] J.A. McCall, P.K. Richards, and G.F. Walters. *Factors in Software Quality.* Nat'l Tech. Information Service, Springfield, Va., 1977.

[22] M. Ohlsson, A. von Mayrhauser, B. McGuire, and C. Wohlin. Code decay analysis of legacy software through successive releases. In *Proceedings of the IEEE Aerospace Conference*, March 1999.

[23] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, December 1972.

[24] Markus Pizka. Software wartung. Lecture script Technische Universität München, July 2004.

[25] Jan Ransom, Ian Sommerville, and Ian Warren. A Method for Assessing Legacy Systems for Evolution. In *Proceedings of Reengineering Forum'98*, 1998.

[26] Janice Singer and Timothy Lethbridge. What's so great about grep? implications for program comprehension tools. submitted to IWPC 1996, 1996.

[27] STSC. Software Reengineering Assessment Handbook v3.0. Technical report, STSC, U.S. Department of Defense, March 1997.

[28] W3C. Html 4.01 specification. World Wide Web, December 1999. http://www.w3.org/TR/html4/.

[29] W3C. Hypertext transfer protocol. World Wide Web, June 1999.