Vorlesungen zum Thema Software-Reengineering

Rainer Koschke, Universität Stuttgart
Breitwiesenstr. 20-22, 70565 Stuttgart
koschke@informatik.uni-stuttgart.de
http://www.informatik.uni-stuttgart.de/ifi/ps/rainer

1. Einleitung

Die praktische Bedeutung von Software Reengineering[1] ist allgemein anerkannt und die mittlerweile mehr als 20 Jahre laufende Forschung in diesem Bereich hat viele interessante Methoden, Techniken und Werkzeuge zu verschiedenen Problemen des Reengineerings hervorgebracht. In der Lehre wird das Thema jedoch nur unzureichend behandelt. Einzelne Universitäten (z.B. Passau, Koblenz oder Stuttgart) behandeln diese Thematik zwar zumindest in Seminaren oder widmen ihr eine Stunde innerhalb einer allgemeineren Software-Engineering-Vorlesung, ganze Vorlesungen über Reengineering sind jedoch selten. Dies hat mehrere Gründe. Zum einen ist das Thema als solches für viele Außenstehende wenig reizvoll, da es um Analyse von Altsoftware geht, die häufig auch noch in unmodernen Programmiersprachen geschrieben ist. Die Konstruktion neuer Systeme auf der grünen Wiese mit Java erscheint sowohl vielen Lehrenden als auch insbesondere der Mehrzahl der Studenten als weitaus interessanter. Diese psychologischen Barrieren existieren sowohl gegen das Reengineering im Speziellen als auch gegen die Wartung im Allgemeinen, sind jedoch bei ersterem besonders ausgeprägt. Außerdem gelten die erzielten Ergebnisse in diesem Bereich unter vielen Lehrenden als unreif und damit als nicht lehrenswert. Diese Kritik ist sicherlich zum Teil gerechtfertigt (wenngleich dasselbe Argument auch jederzeit gegen viele andere Themengebiete der Informatik angeführt werden kann), sie übersieht jedoch die vielen wirklichen Errungenschaften, die im Bereich Reengineering in den letzten Jahren gemacht wurden. Viele Methoden des Reengineerings sind theoretisch fundiert (z.B. Techniken, die auf der Begriffsanalyse basieren) und praktisch erprobt (z.B. Klonerkennung). Nichtzuletzt hat diese Thematik große praktische Bedeutung, und viele Studenten werden die meiste Zeit ihres späteren Berufsleben mit der Wartung von Altsoftware verbringen. Es besteht also eine Notwendigkeit, sie mit dem dafür notwendigen Rüstzeug auszustatten, das sich aus dem heutigen Stand der Wissenschaft ergibt.

Ein weiteres praktisches Hemmnis für Lehrende ist, dass kein allgemeines Lehrbuch existiert. Es gibt sehr viele Bücher zum Thema *Business Reengineering*, die wenigen Bücher jedoch, die über *Software Reengineering* geschrieben wurden, sind meist nur einem speziellen Thema gewidmet (OO-Migration [1], Internet-Anbindung [3], Reuse [4]) oder verzichten auf die technischen Aspekte [5]. Eine abgerundete Behandlung des Themas mit ausreichendem Detail wird insbesondere durch die große Bandbreite verschiedenster Themen des Gebiets erschwert.

Im Rahmen des Studienprojekts *Wartung* des Studiengangs *Software-Technik* (zum damaligen Zeitpunkt noch Modellstudiengang) an der Universität Stuttgart wurde im Wintersemester 1999/2000 zum ersten Male eine Vorlesung Reengineering angeboten. Dieser Beitrag gibt die wesentlichen Ziele und Inhalte dieser Vorlesung sowie die bei der ersten Durchführung gemachten Erfahrungen wieder. Der Beitrag will ein Beispiel geben zu einer Vorlesung über Reengineering und anregen zu einer Diskussion über mögliche alternative Vorlesungsinhalte.

2. Wozu eine Spezialvorlesung?

Insbesondere zwei Fachgebiete der Informatik treffen in einer Reengineering-Vorlesung aufeinander: Software Engineering und Compilerbau. Software Engineering, das sich der Erforschung und Anwendung von Ansätzen zur Entwicklung, Betrieb und Wartung von Software widmet [6], kann als das übergeordnete Gebiet des Reengineerings betrachtet werden. Da Reengineering in den meisten Fällen vom Quellcode ausgeht, kommen hier zusätzlich viele Techniken des Compilerbaus zum Einsatz.

Müssen jedoch wirklich sowohl Aspekte des Software-Engineerings als auch Techniken aus dem Compilerbau in *einer* Lehrveranstaltung zum Thema Reengineering behandelt werden? Traditionell werden diese beiden Fachgebiete an den Universitäten durch verschiedene Lehrstühle vertreten und man könnte vermuten, dass durch den Besuch verschiedener Veranstaltungen im Software-Engineering und Compilerbau auch das Thema Reengineering genügend abgedeckt wird. Meines Erachtens ist hierzu eine eigenständige Vorlesung über Reengineering besser geeignet. Nicht nur weil es Themen im Reengineering gibt (wie z.B. die Remodularisierung oder die Klonerkennung), die weder im traditionellen Software Engineering noch im Compilerbau eine Rolle spielen, sondern auch weil sich der Blickwinkel auf Software-Engineering und Compilerbau im Reengineering-Kontext von jenem unterscheidet, der von einem Software- Ingenieur des Forward Engineerings oder einem Entwickler eines optimierenden Compilers typischerweise eingenommen wird. Die Redewendung “Forward engineering is reengineering on the empty system” ist falsch und irreführend. Der zentrale Unterschied zwischen Forward Engineering, d.h. der Erstentwicklung eines neuen Sy-

1. Wenn im folgenden Text *Reengineering* nicht näher eingegrenzt ist, ist darunter stets *Software Reengineering* zu verstehen.

stems, und Reengineering ist, dass im Falle des Reengineerings ein System bereits existiert, das verstanden und verändert werden soll, und die Anforderungen, die zu den Veränderungen zwingen, verhältnismäßig klar sind; während im Falle des Forward Engineerings das Verständnis des Anwendungsproblems selbst zunächst einen sehr großen Stellenwert einnimmt und für die Implementierung noch ein weiter Spielraum existiert. Die eigentliche Schwierigkeit beim Reengineering ist also weniger das Anwendungsproblem, sondern das bereits existierende System (nach einer engen Auslegung der Definition von Cross und Chikofsky kommen beim Reengineering noch nicht einmal neue funktionale Anforderungen hinzu [1]). Dieser Unterschied zwingt zur Anpassung von Prozessen, Methoden und Werkzeugen, die ursprünglich für das Forward Engineering entwickelt wurden. Gleichfalls hat das Reengineering auch seine eigenen Anforderungen an die Analysetechniken, die im Compilerbau eigentlich für die Codegenerierung entwickelt wurden. So muss z.B. unvollständiger Code syntaktisch und semantisch analysiert sowie nach Möglichkeit auch Präprozessoranweisungen mitanalysiert werden. Außerdem sind im Reengineering bevorzugt globale Kontroll- und Datenflussanalysen notwendig. Ob diese Analysen innerhalb des Reengineerings im selben Grade konservativ sein müssen wie dies bei der Codegenerierung notwendig ist, ist eine interessante, noch offene Frage.

3. Rahmenbedingungen und Zielsetzungen

Vorweg muss auf den besonderen Rahmen dieser an der Universität Stuttgart im Wintersemester 1999/2000 durchgeführten Lehrveranstaltung hingewiesen werden. Die Vorlesung war Teil eines Studienprojektes, das die Studenten der Software-Technik im 6. Semester durchzuführen hatten und das explizit der Problematik Wartung gewidmet sein musste. Bestandteile eines Studienprojekts sind eine einsemestrige Vorlesung, ein Seminar sowie eine zweisemestrige Projektarbeit zum Thema des Studienprojekts. Übungen zur Vorlesung sind explizit nicht vorgesehen, da es Ziel der Vorlesung ist, Inhalte zu vermitteln, die im Projekt dann praktisch umgesetzt werden (die Vorlesung soll aber dennoch ein allgemeineres Verständnis des Themas erreichen, das über das allein für die Durchführung des spezifischen Projekts notwendige Wissen hinausgeht). Die besondere Situation des angegliederten Seminars, das im selben Semester wie die Vorlesung stattfand, bot die Möglichkeit, angrenzende und projektspezifische Themen im Seminar zu behandeln, so dass zwischen Vorlesung, Seminar und Projekt ein ständiger Austausch stattfinden konnte.

Bei einer Durchführung der Reengineering-Vorlesung als eigenständige Vorlesung muss der praktische Anteil in jedem Falle mit mindestens zweiwöchigen Übungen abgedeckt werden. Eine weitere Variante ist, die Vorlesung mit einem Fachpraktikum zu kombinieren, wie es manche Universitäten im Lehrplan vorsehen, so dass der praktische Übungsanteil über mehr oder minder Papierübungen hinausgehen kann. Der in Abschnitt 4 dargestellte Inhalt der Vorlesung schlägt Übungen für ein begleitendes Praktikum vor. Diese Übungen wurden aber nicht in der hier genannten Lehrveranstaltung durchgeführt, da keine für diese besondere Veranstaltung vorgesehen waren. Interessanterweise haben manche Studenten dem zum Trotz bei einer Befragung am Ende des Semesters die fehlenden Übungen eingefordert.

Ein weiterer Vorteil der Vorlesung im Rahmen des Studienprojekts war, dass die Zuhörerschaft gesichert war, da das Studienprojekt *Wartung* für alle Studenten der Software-Technik zu diesem Zeitpunkt noch verbindlich war[1]. Wie eingangs erwähnt muss eine Reengineering-Vorlesung “auf dem freien Markt” die psychologische Hemmschwelle auf Seiten der Studenten wegen des negativen Images von Reengineering abbauen, um eine angemessene Zuhörerschaft für sich zu gewinnen. Für eine neu angekündigte Vorlesung zu diesem Thema sollte also unbedingt ein ausreichendes Maß an Werbung vorgesehen werden. Den Studenten sollte dargestellt werden, dass zwar Altsoftware analysiert und renoviert wird, dazu jedoch interessante und anspruchsvolle Techniken verwendet werden. Weniger die Software selbst, vielmehr die Methoden und Techniken stehen im Vordergrund einer solchen Vorlesung. Zudem sind zwar viele Altsysteme tatsächlich in COBOL geschrieben, die Methoden und Techniken sind jedoch auch auf andere Programmiersprachen anwendbar. Interessanterweise stellt uns ja gerade die Wartung objekt-orientierter Software in C++ und Java vor neue interessante Fragen.

4. Inhalte

Dieser Abschnitt stellt die Struktur und die Inhalte der Vorlesung *Reengineering* vor, wie sie von mir im Rahmen des Studienprojekts Wartung an der Universität Stuttgart im Wintersemester 1999/2000 gelesen wurde. Bei der Themenauswahl war ich darauf bedacht, ein weites und dennoch detailliertes Spektrum an Methoden und Techniken des Reengineerings abzudecken und dabei möglichst wenige Inhalte anderer Vorlesungen des Software-Engineerings und des Compilerbaus zu duplizieren. Sofern allgemeinere Themen des Software Engineerings und des Compilerbaus zur Sprache kamen, wies ich auf die spezifischen Aspekte dieser Themen im Kontext des Reengineerings hin.

Die Vorlesung wurde über einen Zeitraum von 15 Wochen mit 2 Vorlesungsstunden (1,5 h) pro Woche ohne Übungen abgehalten. Nachfolgend werden die Inhalte der einzelnen Blöcke der Vorlesung stichpunkthaft in der Reihenfolge ihrer Abfolge in der Vorlesung mit der Angabe der dafür tatsächlich aufgewandten Zeit in der Einheit V = eine Vorlesung, d.h. 1,5 h, wiedergegeben. Eine mündliche

1. Nach Evaluation der damals noch als Modellstudiengang geführten Software-Technik empfahlen die Peers eine Reduktion der ursprünglich vorgesehenen Zahl von drei Studienprojekte auf zwei. Im geänderten Lehrplan wurde daraufhin das Studienprojekt *Wartung* gestrichen.

Prüfung ist über das gesamte Studienprojekt erst am Ende des Sommersemesters 2000 vorgesehen.

Die erste Vorlesung diente einer allgemeinen Übersicht, so dass die Studenten in der üblichen Orientierungsphase am Anfang des Semesters entscheiden konnten, ob ihnen die Vorlesung zusagt. Daran schloss sich eine Stunde über Management-Aspekte von Reengineering-Projekten an, in der insbesondere ein Vorgehensmodell interaktiv mit den Studenten erarbeitet wurde. Danach folgte die Reihe ausgewählter technischer Themen, beginnend auf niedrigeren Abstraktionsebenen und dann zu solchen höherer Ebenen fortschreitend. Die letzte Vorlesungsstunde diskutierte das mit den Studenten in einer der ersten Vorlesungsstunden gemeinsam entwickelte Vorgehensmodell noch einmal und setzte es in Beziehung zu den verschiedenen in der Vorlesung besprochenen Methoden und Techniken.

Als Termin zur Vorlesung zum Thema Jahr-2000-Umstellung wählte ich die letzte Vorlesungsstunde im Jahr 2000, was einen gewissen Ausklang vor Weihnachten bereiten und die Spannung auf den Jahreswechsel erhöhen sollte. Der Termin zur Vorlesung über Produktlinien als Anwendungsfall für Wrapping und Reengineering wurde bestimmt durch den hierzu eingeladenen Gastdozenten.

Die im Folgenden vorgeschlagenen Übungen sind als begleitende praktische Arbeiten im Rahmen eines vierstündigen Fachpraktikums gedacht. Um den zur Verfügung stehenden Zeitrahmen wirklich einhalten zu können, müssen hierzu feste halbfertige Rahmen zur Lösung der Aufgaben vorbereitet werden, so dass die Studenten so wenig wie möglich mit der Einarbeitung in die Werkzeuge zu tun haben und sich weitestgehend auf die eigentliche Aufgabe konzentrieren können.

4.1. Einführung und Übersicht (1V)

- Bedeutung von Wartung und Reengineering
- Begriffe: Reverse Engineering, Reengineering, Software-Wartung, Wrapping, Business Process Reengineering
- Ziele, Aufgaben, Dimensionen des Reengineerings
- Unterschiede von Forward Engineering und Reengineering
- Übersicht über die Gebiete des Reengineerings

4.2. Reengineering-Projekte (1 V)

- Projektrechtfertigung
- Inventur
- Portfolio-Analyse
- Kostenschätzung, Kosten/Nutzenanalyse
- Vertragsabschluss
- Risiken und häufige Fehler
- Prozessmodell (dies wurde zusammen mit den Studenten erarbeitet)

Praktische Übung

Für ein ausgewähltes C-System und eine vorgegebene Aufgabenstellung ist ein Reengineering-Projekt zu planen und ein entsprechendes Projektangebot zu unterbreiten. Beispiel: Das C-System *XCoral* (72 KLOC) soll teilweise in ein objekt-orientiertes Programm in C++ migriert werden.

4.3. Compilerbau im Reengineering (2 V)

- Abstraktionsebenen (Text, lexikalisch, syntaktisch, semantisch, pragmatisch)
- besondere Anforderungen des Reengineerings
- Begriffe: Dominanz, Postdominanz, Kontrollabhängigkeit, Datenabhängigkeiten, Aliasing
- Zwischendarstellungen: abstrakte Syntaxbäume (AST), Kontrollflussgraphen, Single Static Assignment Form, Program Dependency Graph (PDG), System Dependency Graph (SDG)

Praktische Übung

Für einen ausgewählten Teil der Sprache C soll ein Schema eines abstrakten Syntaxbaums aufgestellt werden. Als semantische Annotation sollen Kontrollfluss sowie Kontroll- und Datenabhängigkeiten explizit repräsentiert werden. Die Datenabhängigkeiten sollen als SSA dargestellt werden. Ein kleines Beispielprogramm (ein kleiner Teil von *XCoral*) soll von Hand in die definierte Zwischendarstellung übertragen werden (einschließlich von Kontroll- und Datenfluss).

4.4. Program Slicing (1,5 V)

- Begriffe Forward und Backward Slice
- Slicing mit PDG
- interprozedurales Slicing mit SDG
- Aufbau des SDG
- Slicing-Anwendungen (insbesondere Messung von Kohäsion)

Praktische Übung

Ein Modul von *XCoral* soll genauer untersucht werden. Dazu wird mit Hilfe eines Programm-Slicers (z.B. [10]) das Modul nach verschiedenen Kriterien vorwärts und rückwärts zerlegt. Konkrete Fragestellungen könnten sein: die Suche nach einem Fehler oder die Analyse, ob die Funktionen des Moduls nicht besser in verschiedene Einzelfunktionen zerlegt werden sollten, um ihre Kohäsion zu erhöhen.

4.5. Remodularisierung (1 V)

- Grundlegende Informationsquellen
- Klassen von strukturellen Remodularisierungstechniken (verbindungsbasiert, metrik-basiert, graph-basiert, begriffs-basiert) sowie deren Vertreter

Praktische Übung

XCoral enthält einen eingebauten Interpreter für eine Teilmenge von C. Der Interpreter wird im Subsystem *Smac*

implementiert. Mit Hilfe des Bauhaus-Remodularisierungswerkzeugs ([9]) sollen innerhalb des Subsystems *Smac* einzelne logische kohäsive Module wiedergewonnen und mit der tatsächlichen Struktur kontrastiert werden. Die dabei gewonnenen logischen Module implementieren Konzepte, die in einer logischen Vererbungsstruktur stehen, z.B. gibt es allgemeine Instruktionen sowie speziellere Konstrukte wie For- oder While-Schleifen in Datenstrukturen des Interpreters. Die Vererbungshierarchie soll später in der Übung zur Begriffsanalyse rekonstruiert werden.

4.6. Begriffsanalyse (1,5 V)

- Begriffe: Objekt, Attribut, Relation, Kontext, Begriff, Begriffsverband, horizontal zerlegbarer Verband, Interferenzen, negative Attribute
- Interpretation des Begriffsverbands
- Anwendung auf Modulerkennung
- Anwendung zur Restrukturierung von Vererbungshierarchien

Praktische Übung

Die in der Übung zur Remodularisierung wiedergewonnenen Konzepte des C-Interpreters *Smac* sind pseudo-objektorientiert implementiert, d.h. in ihrer Implementierung werden objektorientierte Konzepte durch C-Sprachmittel simuliert. Die benutzerdefinierten Datentypen haben dazu eine überlappende Menge gleicher Record-Komponenten. Mit Hilfe dieser Information soll mittels Begriffsanalyse die im Code steckende Vererbungshierarchie rekonstruiert werden. Dazu kann das Concept-Analysis-Tool von Christian Lindig verwendet werden [8].

4.7. Muster- und Planerkennung (1,25 V)

- Begriffe: Cliché, Plan
- Formalismen und Erkennungsalgorithmen für Planerkennung (AST-Muster, Kontroll-/Datenflussmuster, Graphgrammatiken)
- Probleme bei der Planerkennung

Praktische Übung

Mit Hilfe verschiedener Formalismen (textuelle Suche, AST-Mustern und Datenflussmustern) soll nach spezifischen Codierungsmustern gesucht werden, z.B. nach Iterationen über eine verkettete Liste mittels eines Next-Zeigers.

4.8. Anwendungsfall Y2K (1 V)

- Probleme bei der Jahr-2000-Umstellung
- typischer Prozess beim Reengineering des Y2K-Problems
- Suche nach Daten und Datumsmanipulationen
- Korrekturmöglichkeiten (Field Widening, Windowing, Data Encoding, Bridges
- Lessons learned
- Bezüge zu anderen Themen in der Vorlesung

4.9. Anwendungsfall Produktlinien (1 V)

- Begriff Produktlinien
- kombinierte Prozesse zu Produktlinien / Reengineering (Identifikation / Modellierung / Entwurf)
- Beispiel für Wrapping und Reengineering

Praktische Übung

Das Subsystem Smac soll zukünftig ersetzt werden gegen eine Neuimplementierung in C++. Der restliche Teil von *XCoral* soll aber nicht geändert werden. Aufgabe ist dazu, zunächst die tatsächliche Schnittstelle der beiden Komponenten zu identifizieren und dann einen detaillierten Migrations- und Änderungsplan zu erstellen.

4.10. Klonerkennung (1,75 V)

- Entstehung und Arten von Klonen
- Granularität und Abstraktionsebene der Erkennung
- lexem-basierter Ansatz (Baker)
- AST-basierter Ansatz (Baxter et al.)
- metrik-basierter Ansatz (Mayrand et al.)

Praktische Übung

XCoral enthält sehr viele Code-Klone. Mit Hilfe verschiedener Techniken sollen diese identifiziert werden. Die Information über Code-Klone kann auch für die Rekonstruktion der Vererbungshierarchie im Subsystem *Smac* verwendet werden.

4.11. Transformationen (1,5 V)

- Transformationen bezüglich der Abstraktionsebenen
- Beispiele für Code-Transformationen
- Eigenschaften von Transformationen und Transformationssystemen
- Schritte einer Transformation
- Umsetzung einer Transformation
- Klassifikation von Transformationen
- interne Repräsentationen für Transformationen
- Formalismen (Text-, Token-, AST-, Graph-Rewrite-Regeln)
- Beispiele von Transformationssystemen

Praktische Übung

Der ungeschützte Zugriff auf die Interna einer Listendatenstruktur soll beseitigt werden (Iteration über den Next-Zeiger; vgl. die praktische Übung zu 4.7). Dazu wird die Implementierung der Liste durch die in C zur Verfügung stehenden Mittel des Information Hidings verborgen. Die Aufgabe ist nun, durch geeignete automatische Code-Transformationen den direkten Zugriff im Verwender-Code gegen einen entsprechenden Aufruf einer Zugriffsoperation zu ersetzen. Dazu sollen AST-Rewrite-Regeln implementiert werden.

4.12. Zusammenfassung (0,5 V)

- Wiederholung der Kernpunkte der Vorlesung
- Überarbeitung des in der Vorlesungsstunde “Reengineering-Projekte” eingangs erarbeiteten Prozessmodells, wobei die in der Vorlesung vermittelten Methoden und Techniken in Bezug zu diesem Prozessmodell gesetzt wurden

5. Angrenzende und weiterführende Themen

Angrenzende und weiterführende Themen wurden im zum Studienprojekt gehörenden Seminar behandelt. Dazu zählen Wartung im Allgemeinen, kognitive Prozesse beim Programmverstehen, Configuration Management, Regressionstest und Wrapping. Diese Themen könnten auch direkt mit in die Vorlesung einfließen, wenn die Vorlesung als eigenständige Veranstaltung abgehalten werden würde. Darüber hinaus wäre auch zum Beispiel zu erwägen, ob nicht auch über Reverse Engineering von ausführbaren Programmen in Binärform sowie auf die Problematik der geeigneten Visualisierung umfangreicher Artefakte noch eingegangen werden sollte.

Falls man sich dazu entscheidet, die oben angeführten Themen noch miteinfließen zu lassen, sollte man jedoch gleichzeitig andere Themen kürzen oder streichen bzw. die Vorlesungsdauer erhöhen, da das oben beschriebene Vorlesungsprogramm bereits an die Grenze des zeitlich Machbaren stößt.

6. Erfahrungen und Konsequenzen

Eine am Ende der Vorlesung durchgeführte anonyme Befragung der Studenten ergab folgende Kritikpunkte, die bei einer Wiederholung der Veranstaltung berücksichtigt werden:

- es gab keine Übungen
- es gab kein Vorlesungsskript (außer den in der Vorlesung aufgelegten Folien)
- die Vorlesung war manchen Studenten zu umfangreich

Für mich ergeben sich dadurch die folgenden Konsequenzen. Falls die Vorlesung im nächsten Wintersemester eigenständig angeboten wird, werden in jedem Falle Übungen dazu angeboten. Eine begleitende und abgestimmte Durchführung eines Fachpraktikums zu diesem Thema wäre darüber hinaus zu erwägen. Hierzu könnte man verschiedene Werkzeuge wie das Transformationssystem Refine [7], Christian Lindigs Begriffsanalyse [8], das Slicing Tool von GrammaTech [10] oder unsere Bauhaus-Infrastruktur zur Programmanalyse und Remodularisierung [9] verwenden. Um den Stoff besser zu begrenzen, werde ich im nächsten Semester das ohnehin obsolete Thema Jahr-2000-Umstellung ausfallen lassen. Die Vorlesung über Transformationen wird weniger allgemein gehalten werden und stattdessen Refactoring als spezifischeres (und gerade populäres) Beispiel für Code-Transformationen behandelt.

Ingesamt kann die Vorlesung als Erfolg angesehen werden, wie die überdurchschnittlich gute Bewertung durch die Studenten ergab. In einem Kreis von etwa 20 gut motivierten Studenten konnte die Vorlesung interaktiv gestaltet werden. Diese positive Erfahrung motiviert mich und hoffentlich auch andere Lehrende zu einer Wiederholung einer Reengineering-Vorlesung.

Referenzen

[1] Chikofsky, E.J., Cross II, J. H., ‘Reverse Engineering and Design Recovery: A Taxonomy’, IEEE Software, January 1990.

[2] Robert Levey, Reengineering Cobol With Objects : Step by Step to Sustainable Legacy Systems, McGraw-Hill Book Company; ISBN: 007037774X, Januar 1996.

[3] Amjad Umar, Application (Re)Engineering : Building Web-Based Applications and Dealing With Legacies, Prentice Hall; ISBN: 0137500351, April 1997.

[4] Roy Rada, Reengineering Software: How to Reuse Programming to Build New, State-of-the-Art Software, Amacom; ISBN: 0814405096, Januar 1999.

[5] Howard Wilbert Miller, Reengineering Legacy Software Systems, Digital Press; ISBN: 1555581951, November 1997.

[6] IEEE Standard Glossary of Software Engineering Terminology, 610.12-1990.

[7] Reasoning Systems, http://www.reasoning.com.

[8] C. Lindig, Concepts, ftp://ftp.ips.cs.tu-bs.de:/pub/local/softech/misc

[9] Bauhaus, Universität Stuttgart, http://www.informatik.uni-stuttgart.de/ifi/ps/bauhaus/

[10] GrammaTech, http://www.grammatech.com.