

Wartung und Reengineering objektorientierter Software

Hans Malte Kern

Hauptseminar Objekt-Orientierter Entwurf

Zusammenfassung

Die Wartung von Software ist mehr als «Flags putzen, Strukturen nachrichten und Variablen mit Öl auffüllen». Sie umfasst Tätigkeiten von der Umstellung und Anpassung der Software, Vorbereitung zur Wiederverwendung bis hin zu Modernisierungsmaßnahmen. Diese beinhalten das weite Spektrum beginnend mit der Neustrukturierung auf Code-Ebene bis hin zur völligen Renovierung der Software (Reengineering).

Objektorientierung war das Schlagwort gegen Ende des letzten Jahrhunderts. Es versprach

- Wiederverwendung von existierendem Code
- Verkürzung der Entwicklungszeit
- signifikante Senkung der Kosten
- Entstehung besserer Software

In wie weit sich diese Versprechen auf die Wartung und die damit verbundenen Aufgaben auswirken wird hier behandelt.

Inhaltsverzeichnis

1 Einleitung	1
2 Wartung von Software	2
2.1 Der Wartungsbegriff	2
2.2 Tätigkeiten nach der Auslieferung	2
2.3 Wartung im SE-Prozess	3
2.4 Probleme der Wartung	3
3 Wartung objektorientierter Software	3
4 Objektorientiertes Reengineering	4
4.1 Problem Alt-Software	4
4.2 Die Lösungsansätze	4
4.3 Softwaremigration nach Sneed	5
4.4 COREM	5
5 Reengineering und Reverseengineering objektorientierter Software	6
5.1 Das FAMOOS-Projekt	6
5.2 Reengineering Pattern	8

6 Fazit	10
----------------	-----------

Literatur	10
------------------	-----------

1 Einleitung

Wartung von Software ist ein Themenbereich, der immer wichtiger wird, denn Software wird heute länger eingesetzt und ist komplexer als früher und somit fehleranfälliger. Dennoch wird diesem Teil des Software Life-Cycle in der Literatur kein so hoher Stellenwert eingeräumt wie dem (Forward) Softwareengineering.

In der Softwareentwicklung steckt auf den ersten Blick mehr Potenzial: Da gibt es zum einen viele Firmen, die ihre Entwicklungsumgebungen an den Mann (und auch die Frau) bringen wollen, zum anderen gibt es da die Paradigmenwechsel. Früher wurde prozedural programmiert, heute ist es objektorientiert und morgen vielleicht mono:: oder .NET. Ganz zu schweigen von den Büchern die man dazu verkaufen kann, sie reichen von Programmiersprache XY für Dummys bis zu staubfangenden Sprachreferenzen.

Doch ist es nicht so, dass bei jedem Sprachwechsel – auch innerhalb eines Programmierparadigmas – man Code in der alten Sprache hat und den am liebsten in seine neue Lieblingssprache umsetzen will? Und genauso verhält es sich bei den Paradigmenwechseln, nur dass sich die Portierung schwieriger gestaltet.

In Standardwerken der Softwareentwicklung wird das Themengebiet Wartung auf ein paar Seiten des Buches behandelt, dabei nehmen die Gesamtkosten der Wartung im Software Life-Cycle zu:

1970	40–70%
1980	50–70%
1990	60–85%
heute	?

Tabelle 1: Prozentualer Anteil der Wartung an den Gesamtkosten der Software [8, 12]

In vielen Büchern sind demnach 85% der tatsächli-

chen Softwarekosten auf 5% der Seiten zusammenfasst.

2 Wartung von Software

Ähnlich wie bei der Wartung technischer Geräte wie zum Beispiel einem Auto dient die Wartung von Software zur Erhaltung der Qualität. Aber anders als bei technischen Geräten gibt es bei Software keine Abnutzung oder offensichtliche Gebrauchsspuren. Allgemein ist es schwer, die Qualität von Software anzugeben.

Nach [1] wurden folgende Qualitätscharakteristika und deren Untercharakteristika für Softwareprodukte definiert:

- **Funktionalität**
Angemessenheit, Genauigkeit, Kompatibilität, Compilierbarkeit, Sicherheit
- **Zuverlässigkeit**
Fehlertoleranz, Wiederherstellbarkeit, Reifegrad
- **Benutzbarkeit**
Erlernbarkeit, Durchführbarkeit, Verständlichkeit
- **Effizienz**
Ressourcenverbrauch, Zeitverhalten
- **Übertragbarkeit**
Adaptierbarkeit, Anpassungsfähigkeit, Installierbarkeit, Ersetzbarkeit
- **Änderbarkeit**
Analysierbarkeit, Änderbarkeit, Stabilität, Testbarkeit

Besonders die letzten beiden Charakteristika spielen für die Wartung eine große Rolle. Aber was genau ist die Wartung von Software?

2.1 Der Wartungsbegriff

IEEE[2] definiert den Wartungsbegriff für Software wie folgt:

„Software Wartung ist die Modifikation eines Softwareproduktes oder einer Komponente nach der Auslieferung, mit dem Zweck

- der Fehlerkorrektur
- der Verbesserung der Performance oder anderer Systemattribute
- der Adaptierung an eine geänderte Umgebung“

Diese Meinung vertritt auch Boehm [3]:

„Software Wartung ist der Prozess der Veränderung existierender Software unter Beibehaltung der ursprünglich angestrebten Funktionalität.“

Jede Erweiterung, die einem Ausbau der Funktionalität gleichkommt, ist eine Programmänderung aufgrund zusätzlicher Anforderungen und fällt demnach nicht unter den Begriff "Wartung".

Diese enge Sicht vertritt nicht jeder. Für viele ist die Erweiterung der Funktionalität Bestandteil der Wartung und so finden sich in vielen Wartungsverträgen auch Regelungen, die die Erweiterung der Software mit einschließen. Die Wartung beginnt in diesem Fall direkt nach der Auslieferung einer Software.

Nochmals die Analogie mit dem Auto betrachtend sieht man, dass die Definition von Wartung dem entspricht, was man darunter versteht: Bei der Wartung eines Autos wird auch kein Spoiler angebracht, statt dessen wird es wieder in einen verkehrssicheren und gebrauchsfähigen Zustand versetzt, was ja seine grundlegende Funktionalität widerspiegelt.

2.2 Tätigkeiten nach der Auslieferung

Mit der Aussage, die Wartung beginnt nach der Auslieferung der Software hat man nur einen Teil abgedeckt. In Wirklichkeit sehen die Zahlen wie folgt aus:

Erweiterungen	40%
Perfektionierende Wartung	30%
Adaptive Wartung	15%
Korrektive Wartung	12.5%
Präventive Wartung	2.5%

Tabelle 2: Aufschlüsselung der Tätigkeiten nach Auslieferung der Software [14]

Legt man den engen Begriff der Wartung zu Grunde, so sind 60% der Arbeit nach Auslieferung der Software reine Wartung. Die Hälfte davon ist *perfektionierende Wartung*, die nicht unbedingt unter die eigentliche Wartung fällt [14]. Hier steht man vor dem Dilemma, wann die Wartung korrektiv oder perfektionierend ist und worunter die wichtige Tätigkeit der nachträglichen Dokumentation des Programmcodes oder das Nachziehen des Entwurfs fällt?

Die über die Jahre steigenden Kosten der Wartung (siehe Tabelle 1) vermitteln auf den ersten Blick das Bild, dass Software länger eingesetzt wird als früher – das ist aber nicht der primäre Kostenfaktor. Software wird komplexer und die Einsatzumgebungen werden immer komplexer. Heute muss die Software in einen existierenden Verbund von Software integriert werden und sie wird immer mächtiger im Funktionsumfang

und damit steigt auch die Anzahl der wahrscheinlichen Fehler ([11] spricht von *error density*).

An der Software müssen immer öfter Anpassungen vorgenommen werden und damit steigen auch überproportional die potentiellen Fehlerquellen ([12],[18]) und bei jeder Fehlerkorrektur werden immer neue Fehler eingebaut (Murphy's Law).

2.3 Wartung im SE-Prozess

Betrachtet wird hier die Wartung als eigenständiger Prozess, der parallel zur Entwicklung läuft.

In diesem Modell wird immer wieder eine Wartungsphase eingeschoben, die die Ergebnisse der vorangegangenen Phase wartet, d.h. fehlerbereinigt und optimiert. Aber wie in jeder Wartung wird hier nicht die Funktionalität erweitert. Das geschieht in der darauf folgenden Phase, an die sich wieder eine Wartungsphase anschließen sollte. Somit dient die Wartung innerhalb des Softwareentwicklungsprozesses der Qualitätssicherung, die frühzeitig und somit kostenschonend Fehler aufzeigt und behebt.

Auch dieses Vorgehen ist an sich nichts neues: In jeder Produktion werden vor der weitergehenden Verarbeitung die zu verarbeitenden Werkstoffe geprüft (wenn auch meist nur stichprobenartig). Man sollte meinen, dass dieses Vorgehen für die Softwareentwicklung selbstverständlich ist, aber aus eigener Erfahrung in mehreren Firmen und nach [15] und [8] bauen viele Entwickler auf den Aussagen der Entwickler der weiter zu verarbeitenden Komponenten auf, die oft wie ein Kartenhaus in sich zusammenfallen.

2.4 Probleme der Wartung

Man kann die Probleme, die man bei der Wartung von Software vorfindet in vier Gebiete aufteilen:

Technische Probleme

Hierunter fallen Punkte, wie die fehlende Erfahrung mit Wartung von Software, die man recht einfach durch Ausbildung/Schulungen lösen könnte. Aber auch die Schwächen im SE-Prozess, insbesondere fehlende Standards behindern eine effiziente Wartung. Und ein wichtiger Punkt der unter diese Kategorie fällt ist das Fehlen von (benutzbaren) Werkzeugen. Jedes Jahr bekommt der Entwickler mindestens ein Update für seine Entwicklungsumgebung und der Wartungstechniker wäre froh, überhaupt eine integrierte Wartungsumgebung (IME – integrated maintenance environment) zu besitzen. Doch dafür gibt es momentan keinen richtigen Markt.

Ökonomische Probleme

Ein Markt für Wartungswerkzeuge existiert (momentan) nicht. Das liegt zum einen an den unklaren Geschäftsstrategien der Firmen, die selbst Wartung betreiben als auch daran, dass Firmen nicht immer die Verwertungsrechte haben und somit in der Wartung eingeschränkt sind. Aber das größte ökonomische Problem sind die schwer zu kalkulierenden Kosten der Wartung. Aber spätestens jetzt sollten die Softwareschmieden einsehen, dass hier ein riesiger Markt für eine IME existiert (siehe Tabelle 1).

Organisatorische Probleme

Die Probleme, die unter diesen Punkt fallen sind meines Erachtens die gravierendsten. In vielen Firmen ist die Wartung nicht fest im SE-Prozess verankert, es gibt keine Verantwortlichen und auch keine Infrastruktur. Hier wird Software auf den Markt geworfen und wenn es ein Erfolg war und genug Geld dahinter steckt, dass sich eine Wartung (die dann meist Produktpflege genannt wird) rentiert, wird ein Mitarbeiter zur Wartung verpflichtet. Damit ist es ein soziologisches Problem: Wartungstechniker sind meist Einzelkämpfer.

Soziologische Probleme

In der Informatik spielen soziologische Einflüsse und Auswirkungen, wenn überhaupt, eine untergeordnete Rolle. Gerade bei der Wartung von Software ist dies ein nicht unerheblicher Faktor. Ein Entwickler, der zur Wartung von fremder Software eingesetzt wird, muss Fehler, die andere gemacht haben, finden und entfernen. Er muss jedesmal gegen die "not my code"-Haltung ankämpfen, die jeder Entwickler hat. Er kann auch nicht Schöpfer der Software sein, sondern steuert sein Können jemandem bei, der dann die Früchte des Erfolgs erntet. Vielen kommt die Tätigkeit des Wartungstechnikers wie ein Abstellgleis vor, man ist kein Entwickler mehr. Erschwerend kommt noch die kontrollierende Funktion dieser Tätigkeit hinzu, man prüft und bewertet die Arbeit der Kollegen und das ist eine potenzielle Quelle für soziale Spannungen.

3 Wartung objektorientierter Software

Mit der Einführung der objektorientierten Programmierung hat sich auf den ersten Blick nichts an der Problematik der Wartung von Software geändert und es gelten alle unter Kapitel 2 genannten Probleme. Neu hinzu kommt der Irrglaube, der noch in vielen Köpfen vorherrscht, dass die objektorientierte Sprachen bis auf Eigenheiten der Notation gleich sind.

Folgende objektorientierten Sprachmerkmale stellen nach [6] die kleinste gemeinsame Menge aller OO-Sprachen dar:

- Data Abstraction (Datenabstraktion)
- Inheritance (Vererbung)
- Dynamic Type Binding (Späte Bindung)

Dass dies zu sprachabhängigen und nicht paradigmaabhängigen Lösungsansätzen in der Wartung führt, ist ein Problem in der Wartung. Dennoch gibt es eine Sammlung von allgemeinen Ansätzen, die nach einem kleinen Einschub in Kapitel 5 betrachtet werden.

4 Objektorientiertes Reengineering

Lösungsansätze, die das Ziel verfolgen nicht objektorientierten Programmcode in eine objektorientierte Zielsprache zu übersetzen fasse ich unter dem Begriff objektorientiertes Reengineering zusammen.

4.1 Problem Alt-Software

Das Problem mit Alt-Software ist, dass man reife Software auf alter Hardware hat. Meist sind diese Programme in einer Programmiersprache geschrieben, deren Dokumentation in einem Museum gesucht werden muss und entwickelt wurden sie in einem Entwicklungsprozeß, der diesen Namen nicht verdient. Auch wurde die Wartung nur auf die akute Fehlerbeseitigung beschränkt, was man als Wartung mit der Feile und Ölkanne bezeichnen kann.

Dennoch existieren Softwaresysteme, die einen sehr hohen Grad an Reife besitzen und deren Knowhow nicht verloren gehen sollte.

4.2 Die Lösungsansätze

Hier nun einige Lösungsansätze, die man in Firmen antrifft, die Analogie zur Medizin ist hier gewollt.

a) Nichts tun oder "never change a running system" ist die kurzsichtigste, aber am weitesten verbreitete Auffassung. Dies kann mehrere Gründe haben:

- Die Führungskräfte, die eine Umstrukturierung anordnen könnten, wissen zum Teil gar nicht, was für Software sie am laufen haben und in welchem Zustand sie sich befindet.
- Die um den Zustand der Software wissen wollen es gar nicht ändern. Oft ist das Wissen über die Software über die Jahre verloren gegangen und man scheut den Aufwand.

- Befasst man sich mit der Wartung, sieht es aus, als ob man nicht genug ausgelastet sei oder man sich vor der wichtigeren Arbeit drücken wollte.

Einen Infektionsherd, den man nicht (er-)kennt, kann einen umbringen, muss aber nicht.

b) Wrapperbetrieb ist eine künstliche Verlängerung der Lebenszeit der Software. Entweder wird die Softwarelösung mit der Hardware aus dem System herausgenommen und in ein neues, künstliches gestellt und weiter betrieben oder man nimmt eine neue Hardwareumgebung und simuliert darauf die alte und kann so die Software weiter nutzen. In beiden Fällen wird der Software-Lifecycle [7] angehalten, denn an der Software wird nicht mehr weiterentwickelt, in der Medizin würde man dazu Koma sagen. Der Vorteil dieses Ansatzes ist, dass man so Wissen über die Schnittstellen der Software erhält was eine sinnvolle Hinarbeitung auf eine Neuentwicklung darstellt. Jedoch bekommt man so keinen Einblick auf die intern ablaufenden Prozesse.

c) Neuentwicklung ist die teuerste und zeitintensivste Lösungsmöglichkeit und ist mit einigen Schwierigkeiten verbunden. So müssen die internen Abläufe, den von der Software durchgeführten Geschäftsprozessen nachempfunden werden und das erweist sich als äußerst problematisch angesichts fehlender Dokumentation, Kommentare, Spezifikation und gewachsenem Programmcode, der gespickt mit totem Code und irreführenden Kommentaren ist. Besonders schwierig ist es, wenn die Software keine Insellösung, sondern Teil eines Systems ist, das mit dem Implantat (Neuentwicklung) weiter so funktionieren soll, wie zuvor. Das ganze ist auch mit einem Risiko für die Firmen verbunden. Sie zeigen Einblicke in die internen Geschäftsabläufe und je mehr ein System gewachsen ist, um so mehr Abhängigkeiten gibt es, die Geschäftslogik enthalten.

d) Architektur-Transformation ist ein Ansatz, die vorhandene Software in eine objektorientierte Sprache zu transformieren. So bleiben die internen Geschäftsprozesse, die ja in Funktionen, Prozeduren oder GOTO-Anweisungen festgehalten sind, erhalten und es ist garantiert, dass auf Eingaben gleiche Ausgaben erfolgen wie unter dem alten System. Dieses Verfahren wird im folgenden weiter ausgeführt.

Im Gegensatz zu Ansätzen, die nur Komponenten der Softwarelösung in eine OO-Sprache transformie-

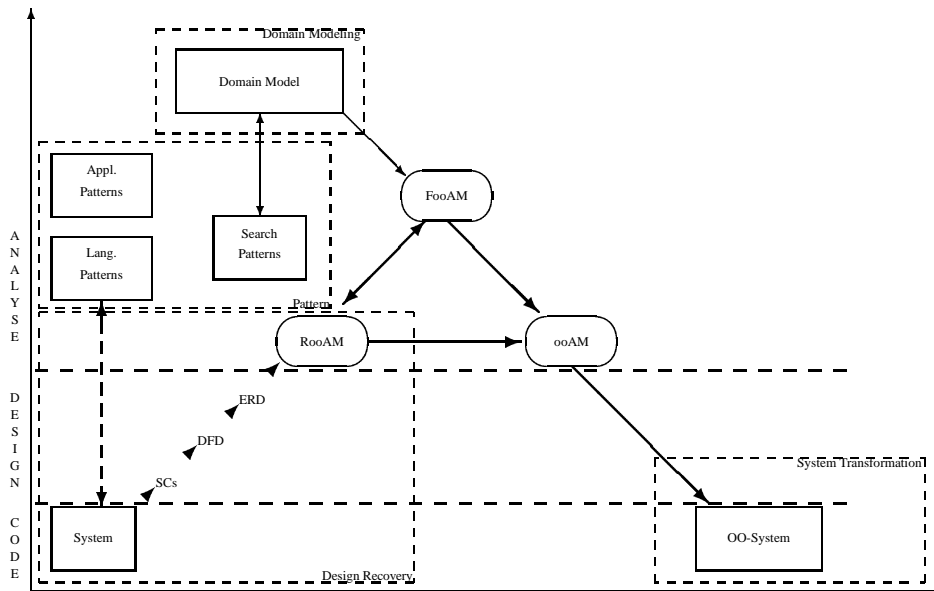


Abbildung 1: Gesamtansicht der COREM Vorgehensweise und die Zwischenergebnisse

ren, versuchen die folgenden Lösungen die gesamte Software in ein neues System umzusetzen.

4.3 Softwemigration nach Sneed

Sneed [20] gliedert das Vorgehen in acht Schritte, in denen ein prozedurales Altsystem in ein objektorientiertes Neusystem umgewandelt wird.

1. Die Entwicklung eines (groben) Objektmodells des Soll-Systems unter Einbezug der neuen Anforderungen.
2. Die Analyse des Altsystems im Hinblick auf Möglichkeiten der Wiederverwendung einzelner Komponenten, sofern überhaupt möglich. Hier wird auch evaluiert, welche Komponenten konvertiert und welche gekapselt werden sollen.
3. Zusammenhänge zwischen den Systembestandteilen nachdokumentieren.
4. Annäherung an das Soll-System durch eine Objektverfeinerung um zu klären, welche Neukomponenten aus vorhandenen Systembestandteilen gebildet werden können.
5. Eine Sanierungsphase in der technisch nicht verwendbare Code- und Datenteile, die aber zu den Anforderungen gehören, in eine verwendbare Form gebracht werden.
6. Implementierung in der neuen Zielsprache
7. Testen gegen das Altsystem
8. Integration in ein lauffähiges Gesamtsystem

4.4 COREM

Das Ziel von COREM[9, 17] (*Capsule Oriented Reverse Engineering Method*) ist ebenfalls eine Umsetzung der Transformation eines prozeduralen Altsystems in ein auf "capsules" basierendes Neusystem. Eine "capsule" stellt hierbei ein Objekt dar, jedoch ohne Unterscheidung zwischen öffentlichen und privaten Methoden und ohne Vererbungsbeziehungen. Das Verfahren hat einen entscheidenden Vorteil gegenüber dem von Sneed: einige Schritte laufen automatisiert ab.

Der gesamte Prozess zur Transformation ist in Abbildung 1 dargestellt.

Die Programmtransformation von COREM wird in folgende vier Schritte unterteilt (siehe Abbildung 2):

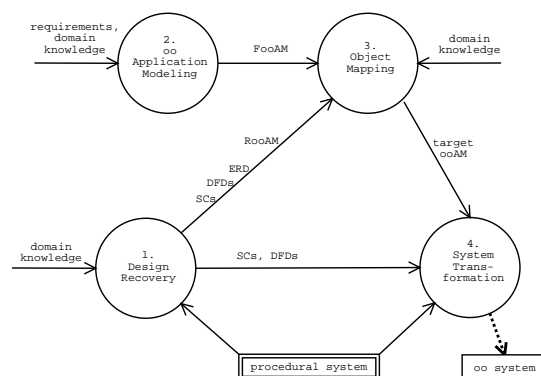


Abbildung 2: Object-Oriented Re-Architecturing

1. Design Recovery

Dieser Reverseengineeringsschritt läuft vollständig automatisch ab und stützt sich auf den Sourcecode und verlässt sich nicht auf die Dokumentation, die entweder veraltet oder nicht vorhanden ist. In diesem Schritt werden Strukturdiagramme (SC), Datenflussdiagramme (DFD), Entity-Relationship-Diagramme (ERD) – alles für Menschen "gut" lesbare Formate – sowie ein objektorientiertes Applikationsmodell (*reverse generated object-oriented application model*, *RooAM*) aus dem Sourcecode erzeugt.

2. Application Modeling

Dies ist ein Forwardengineeringsschritt, in dem das FooAM (*forward generated object-oriented application model*), basierend auf den Anforderungen des existierenden Programms, erzeugt wird.

3. Object Mapping

In diesem Schritt wird das RooAM auf das FooAM abgebildet und als Ergebnis erhält man das Ziel-Modell (*target ooAM*). Dies geschieht in 3 Teilschritten, die man in Abbildung 3 sieht.

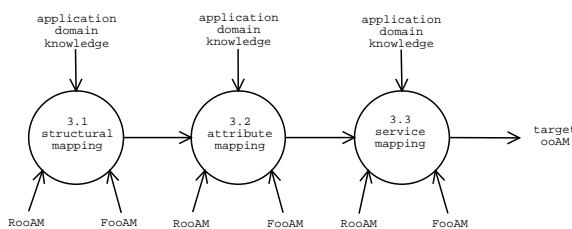


Abbildung 3: Der Object Mapping Prozess

Das ooAM repräsentiert die gewünschte objektorientierte Architektur und dient als Grundlage für die darauf folgende Anpassung des Sourcecodes. Aufgrund der unterschiedlichen Paradigmen lassen sich nicht alle Elemente der prozeduralen Sprache in ein objektorientiertes Pendant umwandeln. Diese prozeduralen Überbleibsel müssen dann gesondert und von Hand transformiert werden.

4. Source-Code Adaption

Im letzten Schritt wird die Code-Transformation abgeschlossen. Mit Hilfe der Ergebnisse vorangegangener Schritte wird der objektorientierte Quellcode erzeugt.

5 Reengineering und Reverseengineering objektorientierter Software

In der Literatur findet man vier Hauptgründe für das Reengineering von Software [16] :

1. Dokumentation des Systems

Die Wiederherstellung verlorener Dokumentation umfasst sowohl die Erstellung nicht vorhandener Entwicklungsdokumentation, als auch die Dokumentation der weitergeführten Entwicklungen.

Die wichtigste Technik die dafür eingesetzt wird ist die Darstellung des Systems mittels alternativer Repräsentationen wie z.B. Struktogramme, Datenflussdiagramme, Entity-Relationship Diagramme usw. (siehe [13]).

2. Unterstützung der Wartung

Die Haupttätigkeit ist das Erkennen und Beseitigen von Seiteneffekten und anderen Anomalien. Dieses Ziel wird z.B. durch zusätzliche Dokumentation und Restrukturierung [10] erreicht.

3. Migration

Darunter fallen die Migration auf eine neue Software-/Hardwareplattform als auch die Migration zu einem neuen CASE-Tool.

4. Software Reuse

Die Wiederverwendung von Softwarekomponenten ist nicht erst seit der Einführung der OO-Sprachen ein wichtiger Grund Reengineering zu betreiben. Ziel ist es Komponenten zu identifizieren, die einen hohen Reifegrad haben und somit in weiteren Softwareentwicklungen verwendet werden können [21].

5. **Reverseengineering vorbereiten** Diese Pattern werden wegen ihres extrem hohen Aufwands nur dann angewandt, wenn ein Reverseengineering folgt.

5.1 Das FAMOOS-Projekt

Wenn man sich mit Reengineering von Software befasste, ging es um Altsoftware. Die Problematik, objektorientierte Software einem Reengineering zu unterziehen stand nicht zur Diskussion. Das FAMOOS-Projekt erstellte ein frei erhältliches Handbuch[19], das sich mit der Problematik von Alt-OO-Software befasst und Reengineeringmethoden evaluiert.

Das Projekt definiert die Ziele für das Reengineering von objektorientierter Software:

- Entwirrung, Entkopplung
- Performancezugewinn
- Portierung auf andere Plattformen
- Design-Extraktion
- Nutzung neuer Technologien

Probleme beim Reengineering

Die meisten Probleme des Reengineering sind nicht vom Programmierparadigma abhängig und gelten somit nicht nur für OO-Sprachen, sondern sind viel mehr ein Zeichen von Fehlern im Entwicklungsprozess wie er z.B. von [7, 18, 12, 14] aufgezeigt wird.

- Unzureichende Dokumentation
- Mangel an Modularisierung
- Mehrfach implementierte Funktionen
- Unsaubere Abgrenzung der Schichten
- Missbrauch der Vererbung
- Operationen an falscher Stelle in der Vererbungshierarchie
- Aushebelung der Datenkapselung
- Missbrauch von Klassen
- Missverständene Pattern
- ...

Reverse-/Reengineering-Techniken

Die wichtigsten Helfer die zum Programmverstehen beitragen sind:

- Metriken. Sie helfen beim Grundverständnis des Codes und zeigen Abweichungen von guter Codierung. Viel wichtiger ist der Einsatz von Metriken um die Auswirkungen von Änderungen aufzuzeigen. Für Beispiele und Klassifizierung von OO-Metriken siehe [4].
- Programm-Visualisierungen ermöglichen es komplexe Zusammenhänge wie Abhängigkeiten, Aufrufgraphen oder Modulinteraktionen am Computer aufzuzeigen. Hierfür bedient man sich der Graphenkonstrukte, die in der Informatik weit verbreitet sind:
 - Bäume
 - Korrelationsgraphen
 - Histogramme
 - Graphen mit Layout
 - Spezielle Graphen (z.B. bipartite Graphen)

Der Einsatz von Graphen bietet die Möglichkeit alle Techniken der Graphentheorie anzuwenden.

- Abstraktion und Umgruppierung ist eine Möglichkeit große und unüberschaubare Softwaresysteme zu überblicken. Hierfür stellt der Computer eine abstraktere Sicht, wie in Abbildung 4, zur Verfügung.

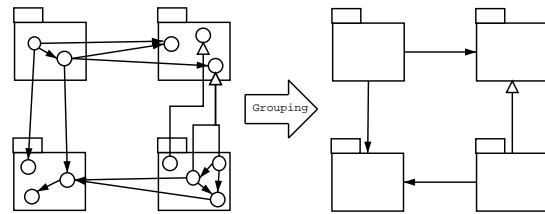


Abbildung 4: Abstraktion durch Grouping

- Refactoring/Reorganisierung ist auch eine Umgruppierung, nur nicht abstrakt und temporär auf der Darstellungsebene, sondern auf der Code-Ebene und damit kann auch die Entwurfsebene beeinflusst werden (siehe [10]).

Reverse Engineering Pattern

In Tabelle 3 sind die einzelnen Ansätze nach folgenden Kategorien untergliedert:

First Contact: Diese Pattern sind, wie der Name schon sagt, für den ersten Kontakt mit dem Software-Patienten gedacht. Sie zeichnen sich durch ressourcenschonende Methoden aus, da in diesem Schritt evaluiert werden soll, ob sich weitere, teurere Reverseengineeringsschritte überhaupt lohnen.

Extract Architecture: Nachdem man den ersten Kontakt mit dem System überstanden hat und auch ein Gefühl für die Software entwickelt hat, ist es an der Zeit, die Baupläne des Gesamtsystems (nach) zu zeichnen, die dann als Grundlage für alle weiteren Vorgehen im Reverseengineeringprozess dienen.

Focus On Hot Areas: Diese Pattern bauen auf den vorangegangenen auf und zeigen, wo und (mit gewissen Einschränkungen) wie man detailliertes Wissen über das Programm erhalten kann. Die Ergebnisse dieses Schritts hängen stärker von dem Können des Wartungsingenieurs ab, als alle anderen und sollte möglichst programmunterstützt ablaufen, da man sonst schnell den Überblick verliert.

Prepare Reengineering: Diese Pattern werden nur dann angewendet, wenn das Reverseengineering Teil eines Reengineerings ist – also die Zielsetzung über das reine Verständnis der internen Abläufe und Logik hinausgehen.

Die einzelnen Pattern sind nach folgenden Kriterien bewertet:

Ressourcenverhalten bewertet die für das Verfahren aufzubringenden Ressourcen, wobei

Pattern	Ressourcenverhalten	Einfache Anwendung	Verlässlichkeit der Erg.	Abstraktionsebene	Soz. Konsequenzen
First Contact					
Read all the code in one hour	++	++	+	-	+
Skim the documentation	++	++	-	+	-
Interview during demo	++	+	0	+	-
Extract Architecture					
Guess objects	-	-	++	++	+
Check the database	-	-	++	+	++
Focus On Hot Areas					
Inspect the largest	0	-	0	0	0
Visualize the structure	-	--	+	+	+
Check the method invocation	-	-	+	+	0
Exploit the changes	--	--	++	+	++
Step through the execution	-	0	++	-	+
Prepare Reengineering					
Write the tests	--	-	++	++	0
Refactor to understand	--	-	0	+	0
Build a prototype	--	-	+	--	++
Focus by wrapping	--	0	0	0	0

++: sehr gut, +: gut, 0: neutral, -: eher schlecht, --: sehr schlecht

Tabelle 3: Reengineering Pattern nach FAMOOS[19]

”++” für einen geringen und ”- -” für einen extrem hohen Ressourcenaufwand steht.

Einfache Anwendung bewertet die Komplexität der angewandten Methoden, wobei ”++” für eine einfache und ”- -” für eine komplexe Anwendung steht.

Verlässlichkeit der Ergebnisse hier steht ”++” für eine sehr hohe Verlässlichkeit und ”- -” steht für unverlässliche Ergebnisse.

Abstraktionsebene gibt an, wie hoch die Wahrscheinlichkeit ist, die Gesamtheit der Komplexität zu verstehen, indem man es in ein mentales Modell umwandelt. Je höher (”++”) die Abstraktionsebene ist, desto genauer kommt das erzeugte Modell an das Vorbild heran.

Soziale Konsequenzen Wenn man Reverseengineering betreibt, kann man auf drei Arten von Kollegen treffen. Die einen glauben an die Wichtigkeit Reengineering zu betreiben und unterstützen die Tätigkeit. Die zweite Gruppe sind die Skeptiker, die Reverseengineering als Zeitverschwendung ansehen und die Arbeit lieber in

ein neues Projekt (in dem man etwas erschaffen kann) investieren. Die letzte Gruppe glaubt nicht, dass sich das Reengineering bezahlt macht und wartet einfach ab, was sich ergibt. Um zu verhindern, dass ein zaghafter Reverseengineering Ansatz nicht gleich in der Mülltonne landet sollte, man aufpassen, dass man nicht unnötig seinen sozialen Kredit verspielt.

Die Skala geht von ”++” was einer prestigefördernden Tätigkeit gleichkommt über eine ”0”, das keine Veränderung im Ansehen darstellt bis hin zum ”-”, das einem Elefanten im Porzellanladen gleichkommt.

5.2 Reengineering Pattern

Reengineering Pattern beschreiben wie man von einer Legacy-Software-Lösung zu einer überarbeiteten Lösung gelangt, die den aktuellen Anforderungen entspricht. Die Auswirkungen der einzelnen Pattern sind in Tabelle 4 zusammengefaßt.

Flexibilität: Sie zu steigern, ist ein Hauptanliegen des Reengineering. Dies erreicht man z.B. durch das Senken der Kopplung zwischen Client- und

	Flexibilität	Verständnis	Wiederverwendung	Aufwand	Skalierbarkeit	Globale Auswirkungen	Anz. der Klassen
Type check elimination in clients	+	+	+			-	
Type check elimination within a provider hierarchy	+	+					-
Detection of duplicated code				-	+		
Repairing a broken architecture	+	+					
Transforming inheritance into compositum	+		+				
Distribute Responsibilities	+	+	+				-

Tabelle 4: Auswirkungen der Reengineering-Pattern nach FAMOOS[19]

Server-Klasse, wie es in *Type Check Elimination In Clients* verfolgt wird.

Bei *Type Check Elimination Within A Provider Hierarchy* werden case- oder komplexe if-Konstrukte, die zur Feststellung der Typen verwendet werden, durch Polymorphismus ersetzt.

Repairing A Broken Architecture hilft beim Auffinden und Beseitigen von Abhängigkeiten, die über die angedachten Grenzen (z.B. Schichten) gehen und somit ungewollte Seiteneffekte mit sich bringen.

Das Pattern *Transforming Inheritance Into Compositum* ist die nachträgliche Hinführung zu dem Compositum Design-Pattern, das im Forward-Engineering eingesetzt wird [5].

Große, schwer zu überschaubare Klassen werden mit *Distribute Responsibilities* auf kleinere Klassen aufgeteilt.

Verständnis: Das Verständnis des Codes und der internen Abläufe zu verbessern ist ein weiteres Anliegen beim Reengineering. Durch *Type Check Elimination In Clients* wird die Kopplung gesenkt und damit verbunden ist eine Überarbeitung des Interface der Klassenhierarchie. Damit erhält man eine modularere Sicht auf das, was die Kernkomponente darstellt.

Ähnlich verhält es sich bei der *Type Check Elimination Within A Provider Hierarchy*. Hier wird eine komplexe Klasse in einfachere und spezialisiertere Klassen aufgeteilt, die jeweils einen Problemfall behandeln.

Bei *Repairing A Broken Architecture* werden verbotene oder ungewollte Abhängigkeiten beseitigt und somit die Entwurfsstrukturen, die angedacht waren, wieder hergestellt.

Der Ansatz, den *Distribute Responsibilities* verfolgt, ist einfach ein Verteilen der Aufgaben im objektorientierten System, um so einer großen und damit schwer zu handhabenden Klasse entgegenzuwirken. Ein weiterer Nebeneffekt: Die dort entstandenen Klassen lassen sich auch besser wiederverwenden, da sie nur atomare Funktionen bieten.

Wiederverwendung: Die Steigerung der Wiederverwendungsmöglichkeiten erreicht man durch eine bessere Kapselung und Spezialisierung der Klassen auf atomare Aufgaben.

Aufwand: Das Sammeln von mehrfach implementierten Funktionen ist vollständig automatisierbar, das automatische Auswählen der potentiellen Kandidaten für das Refactoring geht in den seltensten Fällen. Hier benötigt man den Sachverstand eines Experten und dies kann eine sehr aufwändige Aufgabe werden.

Skalierbarkeit: Das Auffinden von redundantem Code ist ein maschinenunterstützter Prozess und die Programme skalieren (meist) gut.

Globale Auswirkungen: Ein wichtiger Aspekt beim Reengineering sind die globalen Auswirkungen, also in wie weit sich Änderungen durch das gesamte System durchziehen. Da durch *Type Check Elimination In Clients* am Interface der Klassenhierarchie Änderungen vorgenommen werden hat dies Auswirkung auf andere Clients im System.

Anzahl der Klassen: Die Anzahl der Klassen nimmt bei *Distribute Responsibilities* und *Type Check Elimination Within A Provider Hierarchy* zwangsläufig zu. Dies ist nicht direkt ein Nachteil, eher ein Vorteil, da Fehler besser eingegrenzt werden können.

6 Fazit

Das FAMOOS-Projekt ist ein Vorgeschmack auf das, was kommen wird. Vielen Firmen ist noch nicht klar, dass ihre neuen objektorientierten Programme eines Tages einem Reversengineering unterzogen werden müssen und das wird immer so sein, solange es neue Sprachentwicklungen gibt. Und es wird auch immer so sein, dass die Wartung hinter der Entwicklung im Forwardengineering her hinkt.

Die mit der Objektorientierung eingeführten Konzepte können, sofern sie richtig eingesetzt und dokumentiert sind, zum Verständnis des Programms beitragen. Werden die Entwurfsentscheidungen nicht dokumentiert, beginnt das heitere Entwurfsmuster raten, denn bevor man nicht die eingesetzten Pattern erkannt hat, ist die Chance die internen Abläufe zu verstehen sehr gering. Ferner zwingt das Objektkonzept einen Entwickler Daten in Objekten zu kapseln und die dazu implementierte Funktionalität lokal zu halten. Dies sind alles Punkte, die sich positiv auf die Wartung auswirken.

Momentan gibt es noch zu wenig Erfahrungen – insbesondere publizierte – um eine abschließende Bewertung zu machen. Also wird man es so machen, wie bei jedem Sprachparadigmawechsel: abwarten und schauen was sich dann ergibt.

Literatur

- [1] ISO/IEC 9126: *Information technology – Software Product Evaluation – Quality characteristics and guidelines for their use*, 1991.
- [2] ANSI/IEEE Std 610.12-1990 – *Glossary of Software Engineering Terminology*, 1991.
- [3] BOEHM, B. W.: *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ, 1981.
- [4] CIMIANO LAVIN, P.: *OO-Metiken*. Hauptseminar Objektorientierter Entwurf, 2002.
- [5] GAMMA, E., R. HELM, R. JOHNSON und J. VLISSIDES: *Design Patterns*. Addison Wesley, Reading, Mass., 1995.
- [6] HAAK, D.: *Konzepte objektorientierter Programmiersprachen*. Hauptseminar Objektorientierter Entwurf, 2002.
- [7] H. BALZERT: *Lehrbuch der Softwaretechnik – Software-Entwicklung*. Spektrum Verlag, 1996.
- [8] H. BALZERT: *Lehrbuch der Softwaretechnik – Software-Management, Software Qualitätssicherung, Unternehmensmodellierung*. Spektrum Verlag, 1998.
- [9] H. GALL, R. KLÖSCH und R. MITTMEIR: *Object-Oriented Re-Architecturing*. In: SCHÄFER, W. und P. BOTELLA (Herausgeber): *Proceedings of the Fifth European Software Engineering Conference ESEC'95*, Seiten 499–519, Berlin, Germany, September 1995. Springer-Verlag.
- [10] HOH, J.: *Refactoring*. Hauptseminar Objektorientierter Entwurf, 2002.
- [11] INTERNATIONAL FUNCTION POINTS USER GROUP (IFPUG): *Function Point Counting Practices Manual*, 1994.
- [12] I. SOMMERVILLE: *Software Engineering*. Addison-Wesley, 1992.
- [13] L. D. LANDIS, P. M. HYLAND, A.L. GILBERT und A.J. FINE: *Documentation in a software maintenance environment*. In: *Proceedings of the IEEE Conference on Software Maintenance*, IEEE Computer Society, Seiten 66–73, 1988.
- [14] LUDEWIG, J.: *Software Engineering*. 1998.
- [15] P. W. METZGER: *Managing A Programming Project*. Prentice-Hall Inc., 1973.
- [16] R. KLÖSCH: *Reverse Engineering: Why and How to Reverse Engineer Software*. In: *Proceedings of the California Software Symposium CSS'96*, Lecture Notes in Computer Science, Seiten 92–99, 1996.
- [17] R. KLÖSCH und H. GALL: *Objektorientiertes Reverse Engineering*. Springer-Verlag, 1995.
- [18] R. S. PRESSMAN: *Software Engineering – a practitioner's approach*. McGraw Hill, 1997.
- [19] S. DUCASSE und S. DEMEYER (Herausgeber): *The FAMOOS Object-Oriented Reengineering Handbook*. University of Bern, Oktober 1999. URL: <http://iamwww.unibe.ch/~scg/Archive/famoos/handbook>.
- [20] SNEED, H. M.: *Planning the Reengineering of Legacy Systems*. IEEE Software, 12(1):24–34, Januar 1995.
- [21] V. R. BASILI und G. CALDERIA: *Identify and qualify reusable software components*. In: *IEEE Computer*, IEEE Computer Society, Seiten 24: 61–70, feb 1991.