

SOFTWAREEVOLUTION UND WARTUNG: SITUATIONSANALYSE UND ENTWICKLUNGSMÖGLICHKEITEN

Wie der Mensch so altert auch die Software, aber nicht in jedem Fall wird sie dadurch reifer. Untersuchungen von Softwaresystemen über einen jahrzehntelangen Zeitraum hinweg haben gezeigt, dass reale – im echten Kundeneinsatz befindliche – Software einer Evolution unterliegt. Was bei der meist als interessanter angesehenen Entwicklungsphase gern übersehen wird: Die eigentliche Entwicklung eines Softwaresystems fängt mit seiner Auslieferung an den Nutzer gerade erst an. Software, die für den Kunden einen dauerhaften Nutzen darstellen soll, muss kontinuierlich gewartet und weiterentwickelt werden. Dennoch erfreut sich gerade die Disziplin der Softwarewartung nicht unbedingt großer Beliebtheit; dass in dieser längsten Phase im Software-Lebenszyklus dennoch großes Potenzial auch für ambitionierte Entwickler steckt, versucht dieser Artikel etwas zu beleuchten.

Die Verlockung ist groß: Nach erfolgreicher Fertigstellung einer ersten Version eines Softwaresystems wendet sich die talentierte Entwicklungsmannschaft neuen Herausforderungen in neuen Softwareprojekten zu. Das gerade abgelieferte Produkt zur „Serienreife“ zu bringen überlässt man dann anderen, alt gedienten Mitarbeitern, die schon nichts mehr außer Wartung kennen, neu rekrutierten Programmierern, die sich noch nicht wehren können, oder zur Not gar Werkstudenten oder Praktikanten, damit sie was aus dem „richtigen Leben“ lernen. Und zu lernen gibt es in der Wartungsphase sicherlich mehr als genug. Oft kann schon ein einziger Blick auf einen repräsentativen Quellcode genügen, um festzustellen, dass Softwarewartung durchaus Herausforderungen der besonderen Art bereithalten kann. Noch interessanter wird die Aufgabe, wenn man sich die Mühe macht, die frische Software einer umfassenden Qualitätsuntersuchung auf Basis von Code-Reviews, statischer und dynamischer Analyse zu unterziehen. Für manchen Projektleiter mag es unter Umständen erhellend sein, was gute Tools – richtig angewandt – an Aussagen über die vor kurzem noch so gelobte, tolle Architektur zu Tage fördern. Doch wie schaut es dann mit der Umsetzung der Soll-Architektur, mit Design und Codequalität erst *nach* der Einführungsphase aus? Obwohl Software keinem Verschleiß unterliegt, also der ursprüngliche Sinn einer Wartung nicht auf Software übertragbar ist, werden die

„Programs, like people, get old. We can't prevent aging, but we can understand its causes, take steps to limit its effects, temporarily reverse some of the damage it has caused, and prepare for the day when the software is no longer viable. A sign that the Software Engineering profession has matured will be that we lose our pre-occupation with the first release and focus on the long term health of our products.“

D. L. Parnas in [Par94]

Grundsteine für nachfolgende Probleme meistens schon während der Entwicklungsphase gelegt. Abnutzungserscheinungen als Folge von Alterung kann man daher nicht erst nach Produkteinführung ausmachen. Sie lassen sich oft schon nach der dritten Iteration erkennen, wenn nicht frühzeitig und ausreichend gegengesteuert wird.

Trotz der unbestreitbar wichtigen Rolle der Wartung muss man ihren negativen Stellenwert und im Besonderen den der korrigierenden Systemarbeiten wohl als gegeben ansehen (vgl. [Cur89]). Gründe für das negative Image und den allgemeinen „Wartungsunwillen“ liegen sowohl in technischen Randbedingungen als auch im Komplex der so genannten „Peopleware“ (vgl. [DeM99]). Die anstehenden Aufgaben werden lange nicht als so anspruchsvoll gewertet wie diejenigen in der Entwicklungsphase und manch ein Mitarbeiter hegt sicherlich die Befürchtung, nach Projektende unattraktive Aufgaben übernehmen

► der autor



Klaus P. Berg (E-Mail: Klaus-Peter.Berg@siemens.com) arbeitet bei der Siemens AG, München schwerpunktmäßig im Bereich Qualitätssicherung und Systemtest von (J2EE-basierten) Java-Applikationen.

zu müssen, wie eben die der Softwarewartung. Daher möchte ich im Folgenden drei Themen näher beleuchten:

- Softwareevolution und Softwarealterung als Gründe für eine notwendige Wartung,
- Definition, Merkmale und Tätigkeiten der Wartungsphase sowie
- Anforderungen, Herausforderungen und Lösungsansätze, insbesondere unter Berücksichtigung der Werkzeugunterstützung.

Nebenbei wird sich vielleicht auch herausstellen, dass Softwarewartung kombiniert mit Qualitätsanalyse und Qualitätsverbesserung durchaus das Potenzial hat, dem Negativ-Image dieses Aufgabengebiets entgegenzusteuern.

Von der Erschaffung der Software und ihrem Alterungsprozess

„Und Gott sah alles an, was er gemacht hat, und siehe, es war sehr gut“

[1. Mose, 1.31]

Mit der Software ist es wie mit der Erschaffung der Welt im Alten Testament. Ich will damit nicht behaupten, dass manche Entwickler sich ähnlich fühlen, nachdem sie einen letzten Blick auf ihren Quellcode geworfen haben, bevor ihn die Wartungsmannschaft übernehmen muss. Dennoch, irgendwie ist da doch so ein Hochgefühl, obwohl – zugegeben – mit der Dokumentation ist man aus Zeitdruck nicht so weit gekommen wie gewollt, das ursprüngliche klare Design konnte man,



ebenfalls aus Zeitdruck, nicht 100-prozentig umsetzen, die Modularität könnte natürlich noch besser sein, zyklische Abhängigkeiten zwischen Klassen und Modulen ließen sich leider nicht ganz vermeiden, hier und da mag es noch duplizierten Code und die eine oder andere Redundanz geben, und, na ja, die Trace-Ausgaben könnten ebenso wie die Variablen- und Methodennamen verständlicher sein, aber immerhin, die meisten Interfaces haben Dokumentation (wenn auch nur als *Skeleton* von der Entwicklungsumgebung generiert), und das Wichtigste, jede Datei hat den Copyright-Header.

Natürlich ist das alles nur publizistische Übertreibung, zumindest was Ihre Projekte betrifft, doch was ist mit dem Rest? Fakt ist: In vielen lang laufenden Softwareprojekten verschlechtert sich das anfänglich gute Design zunehmend aufgrund unter Zeit- und Kostendruck durchgeführter Erweiterungen und Änderungen. Die Qualität des Quellcodes sinkt aufgrund von Redundanzen und Inkonsistenzen und man muss frühzeitig auf Refactoring-Maßnahmen zurückgreifen, bzw. sie – wie in einem agilen Entwicklungsprozess wie *eXtreme Programming (XP)* – zu einem festen Bestandteil der Projektdurchführung machen.

Refactoring bietet durch semantik-erhaltende Transformationen des Codes die Möglichkeit, vorhandene Programmstrukturen zu verbessern, ohne dass die Software von Grund auf neu geschrieben werden muss. Langlebige Softwaresysteme stellen Entwickler darüber hinaus häufig vor Probleme, die nur noch mit aufwändigem und teilweise fehleranfälliger Reverse-Engineering zu bewältigen sind, weil Dokumentation nicht vorhanden oder nicht aktuell ist. Darüber hinaus ist es einsichtig, dass die innere Struktur des Softwaresystems wesentlichen Einfluss auf dessen Flexibilität und Erweiterbarkeit – kurz Wartbarkeit – hat. Eigentlich ist eine Wartung in diesem Zusammenhang nur dann notwendig, wenn eine offensichtliche Diskrepanz zwischen Anforderungen und Realisierung aufgetreten oder absehbar ist. Doch wie erklärt sich dann die fast zwangsläufige Abnahme der inneren Qualität einer Software schon während ihrer Entwicklung oder nach ihrem echten Einsatz? Die Antwort liegt in einer einfachen Erkenntnis, die schon Parnas so formuliert hat: „Programs, like people, get old and soft-

ware aging will occur in all successful products” (vgl. [Par94]). Und wie wir aus der Erfahrung mit dem menschlichen Alterungsprozess wissen: Mit dem Alter kommen die Krankheiten.

Von den Schwierigkeiten mit alternder Software

„Veränderung und Verfall, wohin man auch blickt...” [H.F. Lyte, Liedtext aus „Abide With Me”]

Sie kennen sicher alle die Maxime „Never change a running system”. Doch spätestens seit den Problemen mit der Jahr-2000-Umstellung ist klar:

- Viele Softwaresysteme leben deutlich länger, als ursprünglich vorgesehen (und auch länger, als von manchen Mitarbeitern vielleicht gewünscht).
- Softwaresysteme sind vielerlei kleinen und größeren Änderungen unterworfen, um sie an neue Anforderungen bzw. neue Plattformen, Frameworks oder Technologien anzupassen. Das kann, wie bei XP, durch die starke Einbeziehung des Kunden und mögliche Technologie-Explorationsphasen durchaus schon während der Entwicklung geschehen. Leider werden diese Änderungen nicht immer adäquat dokumentiert oder sie sind nicht kompatibel zum ursprünglichen Design. Das kann dann dazu führen, dass weder alle akkumulierten Änderungen noch die Anforderungen oder ihre Implementierung letztendlich (allen) klar sind.
- Organisationen sind oft abhängig vom Know-how einiger weniger Schlüsselpersonen („Gurus”). Sie sind die Einzigen, die wirklich mit der internen Struktur des Systems vertraut sind und auch die undokumentierten Zusammenhänge bzw. Gründe für einzelne Designentscheidungen kennen. Streuen sie ihr Wissen nicht ausreichend oder verlassen gar die Mannschaft, entstehen schwerwiegende Lücken. XP versucht dem mit *Pair-Programming* und wechselnden Arbeitspaketen für Entwicklergruppen entgegenzusteuern, doch ob dies wirklich gelingt, hängt nicht nur von der Hartnäckigkeit der Projektleitung ab, sondern vor allem vom aufrichtigen Willen aller Beteiligten.

Zur Abmilderung des Alterungsprozesses wurde die *Softwarewartung* „erfunden”. Sie bezeichnet als *Tätigkeit* eine Änderung an einem Softwaresystem nach dessen Auslieferung, als *Prozess* die Schritte, die in einem Wartungsfall sequenziell durchzuführen sind, und als *Phase* den Abschnitt des Lebenszyklus eines Softwaresystems von dessen Auslieferung bis zur Stilllegung [WR05]. Den Prozess der *Veränderung* eines Softwaresystems – von der Erstellung bis zur Stilllegung – bezeichnet man als *Softwareevolution*. Sie umfasst damit die Entwicklung, Wartung, Migration, und letztendlich auch die Stilllegung: „*Evolution is what actually happens to the software*” (M.W. Godfrey, siehe: plg.uwa.terloo.ca/~migod/papers/icsm00-slides.ppt). Die Gesetze dieser Evolution lassen sich relativ schnell zusammenfassen:

1. Die Uhr läuft ab: Jede nützliche Software muss weiterentwickelt und gepflegt werden oder sie stirbt.
2. Auch wenn ein Softwaresystem größer wird, beschränkt doch seine immanente Komplexität seine letztendliche Größe.
3. Der Entwicklungsaufwand und -Fortschritt (!) gemessen über die Zeit ist ungefähr konstant.

Daraus lassen sich unter anderem folgende *Ratschläge für erfolgreiche Softwareprojekte* ableiten:

- Wir müssen vor allem die Komplexität eines Softwaresystems frühzeitig in den Griff bekommen. Zugegeben, keine unbedingt neue Erkenntnis, aber neben „entwicklerischer Genialität” sind dazu auch analytische Verfahren zum Verstehen und Bewerten von Software und deren Änderungsauswirkungen notwendig (wir werden in einem späteren Abschnitt noch sehen, mit welchen Methoden und Tools dies erreicht werden kann).
- Erfolgreiche Softwaresysteme erfordern periodische Re-Designs und/oder relativ kurze Iterationen (mit der Möglichkeit zu Refactoring).
- Wir müssen den Softwareentwicklungsprozess als ein Feedback-System begreifen und nicht als einen passiven Lehrsatz.



Aus den Gesetzen der Softwareevolution ergibt sich, dass Softwaresysteme im Laufe ihres Lebens wachsen, altern, komplexer werden und an Qualität verlieren – es sei denn, sie werden ausreichend gewartet. Wartung findet in der Regel nach Abschluss der Entwicklungsphase statt, sie kann jedoch auch schon während der Entwicklung nötig werden. Umgekehrt endet die Entwicklung nicht automatisch mit der Inbetriebnahme der Software. Bei einer inkrementellen Entwicklung, z.B. gemäß geplanten Iterationen in einem agilen Prozess, erfolgen die Erweiterungen des Systems nicht überraschend, sondern sind schon zu Beginn der Entwicklung geplant [OL05]. Das Thema Wartung wird leider in der Literatur – im Vergleich zur Aufmerksamkeit, die man der Entwicklungsphase schenkt – eher stiefmütterlich behandelt. Deshalb hier einige Grundbegriffe.

Softwarewartung als Software-Geriatrie

„Seltsam: nach den Jahren der Last hat man die Last der Jahre.“ [B. Shaw]

Nach [IEEE] bezeichnet man als *Wartung* die Veränderung eines Softwareprodukts nach dessen Auslieferung, um Fehler zu beheben, Performance oder andere Attribute zu verbessern oder Anpassungen an die veränderte Umgebung vorzunehmen. Die Wartung von Softwaresystemen als letzte Phase des Software-Lebenszyklus wurde historisch in ihrer Komplexität und bezüglich der Notwendigkeit, geeignete Methoden und Werkzeuge zur angemessenen Durchführung zu verwenden, lange Zeit unterschätzt. Dabei beansprucht die Wartungsphase nach mehreren Untersuchungen ca. zwei Drittel des Gesamtbudgets für den Bau und Betrieb eines Softwaresystems von der Vision bis zur Stilllegung [WR05]. Folgende Arten der Wartung können unterschieden werden [WR04]:

- **Korrigierende Wartung:** Die korrigierende Wartung beschäftigt sich mit der Behebung von Fehlern, die nach der Auslieferung der Software auftreten (reaktive Maßnahme).
- **Präventive Wartung:** Die präventive Wartung versucht latente Fehler zu finden, bevor sie zu effektiven Fehlern werden. Mit Tools wie „FindBugs“ (vgl. [Fin]) oder „Jtest“ (vgl. [Par]) im Java-Umfeld können bestimmte, potenzielle Fehlerarten schon vorab aufge-

spürt werden, z. B. `NullPointerException`, `IndexOutOfBoundsException`, „Double Checked Locking“ (sollte in Java im Gegensatz zu C++ vermieden werden) oder Multithreading-Probleme durch Synchronisationsschwächen. Ähnliches strebt auch „PC-lint/FlexeLint“ im C/C++-Bereich an (z. B. Aufdecken von Speicherlecks, vgl. [Gim]).

- **Adaptive Wartung:** Die Lebensdauer von Software überschreitet oft eine Dauer von zehn Jahren. Es ist nicht außergewöhnlich, dass sich in dieser Zeit die ursprüngliche Systemumgebung verändert. Das hat zur Folge, dass die Software diesen Veränderungen angepasst und weiterentwickelt werden muss. Alle Tätigkeiten, die in diesen Bereich fallen, werden als adaptive Wartung bezeichnet.
- **Perfektionierende Wartung:** Perfektionierende Wartung bedeutet die Verbesserung der Software. Dies betrifft die Performance, Wartbarkeit oder andere Eigenschaften der Software, insbesondere aber auch die Dokumentation.
- **Funktionale Erweiterungen:** Hierbei handelt es sich um eine Art der Wartung, bei der neue Leistungsmerkmale hinzugefügt werden.

Quantitative Merkmale aller Wartungsarten sind die Anzahl der jährlich hinzugefügten, gelöschten und geänderten Codezeilen. Für die korrigierende und perfektionierende Wartung ist auch entscheidend, wie die Software verändert wurde. Handelt es sich z. B. nur um schnelle Bug-Fixes oder hat man die Möglichkeiten einer Softwaresanierung durch Refactoring genutzt? Insbesondere die Art, *wie* man die notwendigen Änderungen in das existierende System einbaut, und nicht nur, *wie schnell*, entscheidet darüber, wie gut die Wartungsmannschaft ihren Job macht. Softwarewartung ist damit durchaus eine nicht triviale Angelegenheit.

Sachgerechte Softwarewartung basiert vor allem auf einem gründlichem, inhaltlichem Verständnis des Anwendungsprogramms, damit nicht nur syntaktisch sauber gearbeitet wird, sondern auch bei Wartung und Refactoring/Re-Engineering adäquat für die Gesamtanwendung Eingriffe und Änderungen beherrscht werden. Optimal wäre es natürlich spätere Wartungsmöglichkeiten bereits in der

Entwurfsphase vorzusehen. Dies geschieht durch die Berücksichtigung von so genannten nicht-funktionalen Anforderungen wie Modifizierbarkeit (*Modifiability*). „Modifiability“ wird bisweilen auch als *Maintainability* bezeichnet, womit wir direkt wieder bei der Wartbarkeit wären. Architektur und Design, die dieser Anforderung Rechnung tragen, bezeichnet man als „Design for change“ oder „Design for success“.

Solche Architekturen können mit der Zeit wachsen und sich verändern, ohne dass ihre Kernelemente und Grundgedanken verwässert werden. Sie erlauben somit den Einbau neuer Features, ohne dass ein größeres Redesign vorgenommen werden muss. Dies betrifft z. B. die Art, in der Funktionalität auf Module und Subsysteme verteilt ist, oder das Bereitstellen adäquater Abstraktionen und Interfaces für leichte Erweiterbarkeit.

Stimuli für diese nicht-funktionalen Anforderungen sind: neue Funktionen, neue Plattformen, neue Betriebssysteme oder geänderte Qualitätsanforderungen. Das am häufigsten auftretende Ereignis ist dabei sicher das der funktionalen Erweiterungen. Obwohl in manch scharfer Definition von Softwarewartung gar nicht erfasst, spielen sie doch eine große Rolle. Das soll aber nicht heißen, dass die ursprüngliche Software – quasi vorausschauend – schon versuchen soll, alle denkbaren Erweiterungen vorwegzunehmen und zumindest rudimentär schon einmal zur Verfügung zu stellen. Das würde nur zu aufgeblähtem und vorerst nicht genutztem Code („dead code“) führen. Hier ist also Augenmaß gefordert, nach der Maxime: „You ain't gonna need it (yet)“.

Neben dem bereits erwähnten „Design for success“ kommen als weitere *präventive Maßnahmen* der Software-Geriatrie noch in Betracht: gute Dokumentation, rigoroses Testen (auf Basis von Unit/Integrations- und Systemtests) sowie Code-Reviews, d. h. das Einholen einer zweiten Meinung. Wenn das alles nichts mehr hilft und der Softwareverfall einmal eingetreten ist, bleiben nur noch Amputation und Re-Engineering/Restrukturierung, d. h. ein ziemlicher Umbau der existierenden Software. Spätestens jetzt stellt sich die Frage nach Unterstützung. Man braucht „Hilfstruppen“ – und das nicht nur personell. Werkzeugeinsatz halte ich daher für eine adäquate Durchführung von War-



Impactanalyse:
 Alle Subsysteme, die das Subsystem "IG.util" direkt nutzen, sind dunkelrot eingefärbt.
 Alle Subsysteme, die das Subsystem "IG.util" nur indirekt nutzen, sind hellrot eingefärbt.
 Nicht gefärbte Subsysteme nutzen "IG.util" weder direkt noch indirekt.

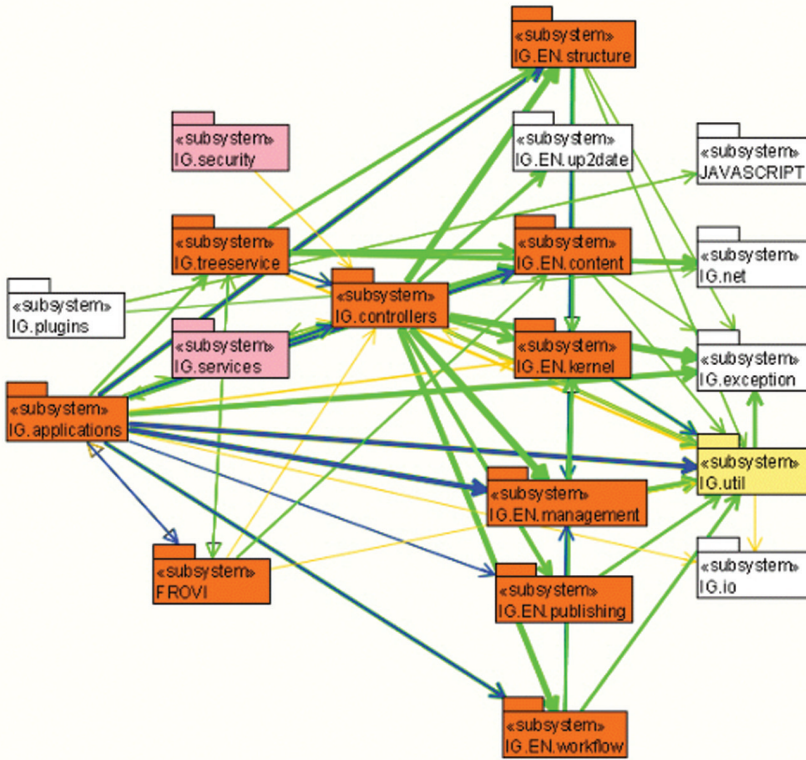


Abb. 1: Impact-Analyse mit dem Sotographen

tungsaufgaben für unabdingbar. Deshalb möchte ich im nächsten Abschnitt einen zentralen Punkt – wenn nicht gar die *Grundvoraussetzung* – jeder Wartungstätigkeit betrachten: das Thema „Verstehen von Software“ und, wie gute Tools uns hierbei unterstützen können.

Analysieren, Visualisieren und Verstehen von Software

„Es gibt immer eine andere Lösung.“

[Daniel Goedevert]

„Span of Understanding“ nennt man die Zeitspanne, die der Programmierer zum Verstehen eines definierten Programmstückes benötigt. Ziel ist natürlich, diese Zeitspanne zu minimieren, und da können Werkzeuge zur statischen Analyse und Visualisierung von Software immens helfen. Softwarevisualisierung [ZE04] zeigt, wie Software strukturiert ist, was sie macht, wie sie funktioniert, warum sie funktioniert oder warum auch nicht. Die bildliche Darstellung von Datenstrukturen und dem Programmverhalten stellt somit

eine wertvolle Ergänzung des reinen Programmcodes dar. Softwarevisualisierung kann unabhängig vom Ablauf des untersuchten Programms (*statische Analyse*) oder während dessen Laufzeit (*dynamische Analyse*) erfolgen und dabei entweder den Code, die Daten oder das Verhalten des Algorithmus illustrieren. Die Ergebnisse der dynamischen Analyse werden dabei oft nicht animiert visualisiert, sondern als *State-Charts* oder Sequenzdiagramme dargestellt. Ich würde dabei generell die Animation auch nicht überbewerten, oft ist sie nicht mehr als eine interessante Auflockerung und bietet als eigenständige Methode keine unbedingt neuen oder zusätzlichen Informationen.

Ein wichtiges Anwendungsfeld der Softwarevisualisierung ist die Hilfe beim Verstehen bereits existierender Software [ZE04], wie wir sie insbesondere in der Wartungsphase benötigen. Diese Aufgabe gestaltet sich in der Regel schwierig und ist sehr zeitintensiv. Vor allem bei älteren und recht umfangreichen Softwaresystemen

geht häufig ein Großteil der Zeit des Programmierers für das Verstehen des alten Codes verloren. Die Erforschung eines Programmcodes ist für alle Programmierer relevant, die versuchen, einen ihnen nicht vertrauten Codeabschnitt zu ändern. Programmierer, die zusätzliche Funktionalität in ein Programm einfügen, müssen zunächst den schon vorhandenen Quelltext analysieren, um zu erkennen, welche Dateien die existierende Funktionalität enthalten und welche Codefragmente dementsprechend verändert werden müssen.

Solcherart durchgeführtes Code-Verstehen – angewandt auf elaborierten Code von „genialen“ Programmierern – ist durchaus eine in der Wartungsphase anspruchsvolle Tätigkeit, womit wir bei den anfangs im Artikel erwähnten Herausforderungen in der Softwarewartung angekommen sind, obwohl eine Profilierung durch Wartungsaktivitäten nur schwer vorstellbar scheint.

Eines der größten Probleme bei der Visualisierung ist das Handling massenhafter Informationen. Schwierigkeiten ergeben sich vor allem bei umfangreichen Programmen oder großen Datenmengen, die dargestellt werden sollen. Punktwolkendarstellungen, Informationsgebirge oder ähnliches geben in der Regel nur einen groben Überblick, sodass Skalierbarkeit, Filterfunktionen und Zooming gefragt sind. Bei vielen grafischen Darstellungen ist es auch ein schwieriges Unterfangen, ein geeignetes Layout, und gänzlich unmöglich, ein perfektes Layout für die Informationsvermittlung zu generieren. Dennoch, Werkzeuge im Java-Umfeld, die sich gerade auf dem Gebiet „Informationsdarstellung bei mittleren und großen Systemen“ positiv hervortun, sind beispielsweise IBMs „Structural Analysis for Java“ und der „Sotograph“ (Computer-Tomograph für Java/C/C++) der deutschschweizerischen Firma Software Tomography GmbH (vgl. [Ber05], [Ben03]). Für die Wartungsaufgaben sind insbesondere die Features des Explorers und die Impact-Analyse („What-if-Analyse“) von Bedeutung. Die *What-if-Analyse* visualisiert die Fragestellung: „Wie würden sich Änderungen an einer Komponente (Klasse/Interface) auf andere Komponenten desselben oder anderer Module auswirken?“

Dabei kann der Sotograph die Abhängigkeiten sogar auf der höheren Abstraktionsebene der Subsysteme visualisieren, was



insbesondere bei sehr großen Systemen von Vorteil ist (siehe Abb. 1). Bedenkt man Murphys „Law of Change Propagation“ – „If a change to a source statement can introduce an error, it will“ – dann ist das durchaus für jeden Wartungstechniker eine ernstzunehmende Problematik, bei deren Bewältigung man jede Hilfe nutzen sollte.

Ebenfalls für die Visualisierung von Java und C/C++-Programmen geeignet, wenn auch mit bescheideneren Mitteln (aber auch bei bescheidenerem Anschaffungspreis), ist das Werkzeug „Understand for C++/Java“ (vgl. [Sci-a]).

Bei der gesamten Thematik der Softwarevisualisierung zeigt sich auch, dass gute Software etwas mit Eleganz und Schönheit – aber auch mit Einfachheit – zu tun hat. Das bestätigt auch ein Zitat von David Gelernter: „Most computer technologists don't like to discuss it, but the importance of beauty is a consistent (if sometimes inconspicuous) thread in the software literature. Beauty is more important in computing than anywhere else in technology [...] Beauty is the best guide we have“ (vgl. [Gel98]).

Man darf sich jedoch bei allen Tool-Möglichkeiten nicht vom wahren Problem der Softwarewartung ablenken lassen, das Harry M. Sneed so formuliert: „Trotz aller automatischer Hilfen, Software-Sanierung bleibt aber im wesentlichen eine arbeitsintensive manuelle Aufgabe. Man braucht qualifizierte Software-Techniker mit klar definierten Qualitätszielen, um schlechte Software in gute Software umzuwandeln“ (vgl. [Sne84]).

Refactoring und die Möglichkeiten der Softwaresanierung

„Simplify, combine, eliminate.“

[Rechtin & Maier,

The Art of System Architecting]

Bei jeder Wartungsmaßnahme stellt sich somit unweigerlich die Frage: Wie setze ich die Ergebnisse des Analyseprozesses, d.h. des Verstehens der Software und der Auswirkungen von notwendigen Änderungen, in adäquate Code-Modifikationen um? Hier liegt neben der Versuchung eines „Quick Hack“, d. h. eines schnellen Bug-Fixing, auch die Möglichkeit für mehr oder weniger umfangreiches Refactoring – wenn das Management mitspielt. In der Vergangenheit wurde vor allem ein defensives

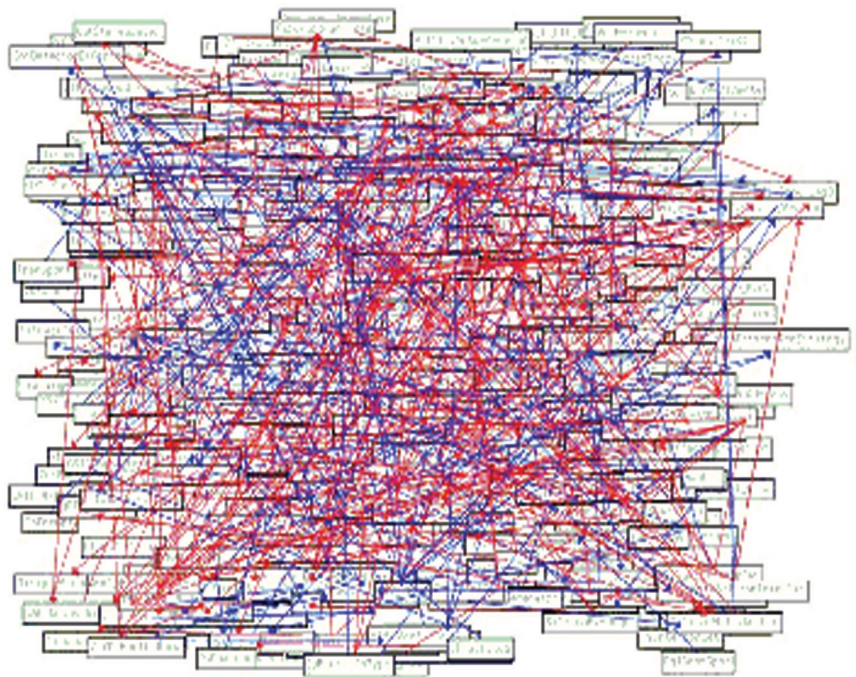


Abb. 2: Wechselseitige Abhängigkeiten zwischen Modulen

Änderungsverhalten mit minimal-invasiven chirurgischen Eingriffen propagiert bzw. gelebt. Heute bieten sich uns andere Möglichkeiten: das Buzzword in diesem Zusammenhang heißt „Refactoring“. Auf der Basis von Martin Fowlers 72 Refactorings existieren in immer mehr Entwicklungsumgebungen integrierte „Refactoring Browser“, die automatisierte und sichere Codetransformationen ermöglichen und damit einen Beitrag zu „mehr Softwarequalität“ leisten.

Die Entwicklungsumgebungen „Eclipse“ (open-source, siehe www.eclipse.org), „IntelliJ IDEA“ (von JetBrains) oder der „JBuilder“ (von Borland) enthalten beispielsweise solche Funktionalität. Einen guten Tool-Überblick erhält man auf www.refactoring.com/tools.html. Große Refactorings (vgl. [Roo04]) verlangen meist auch größere Design-Umbauten und lassen sich derzeit noch nicht automatisieren.

Für kleinere Design-Refactorings im Java-Umfeld bietet das Werkzeug „OptimalAdvisor“ von Compuware (vgl. [Com]) in einzelnen Fällen sogar schon eine Hilfestellung, z. B. bei der Auflösung von Zyklen durch das Verschieben von Klassen. Wenn sich dagegen die Abhängigkeiten zwischen Modulen wie in **Abbildung 2** darstellen, braucht es wohl keine große Vorstellungskraft, um zu wissen, dass zur Sanierung dieser Software Mut und

Selbstvertrauen gefragt sind. Ganz zu schweigen, was ein solches Analyseergebnis über die oben genannte Eleganz der Software vermuten lässt... Für die Softwarewartung erschließen sich jedenfalls mit den Themen Refactoring und Re-Engineering sehr gute Möglichkeiten auf die diversen Anforderungen angemessen zu reagieren.

Sanieren wir in die richtige Richtung?

„Wer bessern will, macht oft das Gute schlecht.“ [König Lear, 1.4]

Meist wird in Diskussionen über Software und Softwarequalität nur die Entwicklungsphase betrachtet, d.h. die Qualitätssicherung im Software-Lebenszyklus wird häufig getrennt von der späteren Pflege der Software gesehen. Hier sollte man zu einer mehr ganzheitlichen Betrachtungsweise kommen. Für die Qualitätssicherung im Wartungsprozess existieren allerdings keine separaten Modelle, obwohl die Wartung ein überaus sensibler Bereich ist, da eine funktionsfähige Anwendung durch unsachgemäße Behandlung unbrauchbar gemacht werden kann. Die Gefahr von sich potenzierenden Fehlern ist mit jeder neuen Version gegeben. Da unkontrollierte Änderungen zu Inkonsistenzen im System oder dessen Dokumentation führen würden, ist eine Ände-



rungs-Kontrollprozedur festzulegen, in der jedem genehmigten Änderungswunsch eine Priorität zugeordnet wird, die über den Zeitpunkt seiner Ausführung entscheidet. Diese Bewertung wird z.B. durch eine Änderungs-Review-Gruppe vorgenommen, an der neben Mitgliedern des Wartungsteams auch Benutzer beteiligt sind. Nicht zu vergessen ist, dass die Qualitätssicherung nicht erst beim Abnahmetest eines gewarteten Produktes ansetzt. Sie muss vielmehr sicherstellen, dass die Standards, die im Software-Entwicklungsprozess definiert wurden, auch in späteren Phasen eingehalten werden.

Softwaresysteme haben die Tendenz, durch jede Änderung komplexer und umfangreicher zu werden. Dadurch verringern sich die Verständlichkeit des Codes und die Wartbarkeit. Um dem entgegenzusteuern sind Metriken und Trendanalysen notwendig, die über den Erfolg oder Misserfolg einer eingeschlagenen Sanierungsrichtung Auskunft geben. Trendanalysen verfolgen die Entwicklung von Architektur und Qualität von Softwaresystemen über die Zeit. Geeignete Metriken (vgl. [Mar02]) und ihre werkzeuggestützte Erfassung sind ein weiteres mögliches Hilfsmittel. Damit können in der Softwarewartung eine gezielte Verfolgung und Analyse der Weiterentwicklung eines Systems betrieben werden, denn es ist doch wichtig, dass wenigstens die Richtung stimmt, oder?

Ein poetischer Rückblick und Ausblick

„Design and programming are human activities; forget that and all is lost.“

[Bjarne Stroustrup]

Wie wir gesehen haben, wird sicherlich ein Großteil der Wartungsproblematik durch einen inadäquaten Umgang mit der Softwarealterung und durch die Auslieferung von qualitativ minderwertiger Software verursacht, die oft gerade bezüglich ihrer Wartungsfreundlichkeit Defizite aufweist. Neben Sanierungs- und Qualitäts-

sicherungsmaßnahmen wurde auch auf die Rolle von Werkzeugen eingegangen, mit deren Hilfe auch bei zum Teil aussichtslos erscheinenden Applikationen noch Verbesserungen erzielt werden können. Diese Tools dienen auch im eigentlichen Wartungsprozess als Hilfsmittel, um die Potenzierung von Fehlern zu vermeiden und eine qualitativ hochwertige Wartung sicherzustellen.

Trotz der unbestreitbaren Rolle der eingesetzten Technik ist der psychologische Aspekt jedoch nicht zu vernachlässigen (vgl. [Mer95]). Sämtliche Maßnahmen müssen insbesondere auch den Faktor Mensch im „Wartungsgetto“ berücksichtigen (alle engagierten Wartungstechniker mögen mir diese Begriffsbildung nachsehen). Da auch dieser Bereich auf Engagement und talentierte Mitarbeiter angewiesen ist, liegt im Einsatz von Motivationsmaßnahmen und Incentives ein erhebliches Produktivitätspotenzial. Lassen Sie mich daher unsere Situationsanalyse der Softwareevolution- und -wartung mit einer poetischen Betrachtung abschließen, die Mensch und Technik gleichermaßen im Visier hat. ■

POEM – David H. H. Diamond

*The fellow who designed it,
Is working far away;
The spec's not been updated,
For many a livelong day.*

*The guy who implemented it is
Promoted up the line;
And some of the enhancements
Didn't match to the design.*

*They haven't kept the flowcharts,
The manual's a mess,
And most of what you need to know
You'll simply have to guess.*

*We do not know the reason,
Why the bugs pour in like rain,
But don't just stand here gaping,
Get out there and MAINTAIN.*

Literatur & Links

- [Ben03] M. Bennis, D. Beyer, W. Bischofberger, Eclipse auf dem Prüfstand: eine Fallstudie zur statischen Programmanalyse, in: OBJEKTSpektrum 5/03 (siehe: www.sigs.de/publications/os/2003/05/bischofberger_05_05_03.pdf)
- [Ber05] K.P. Berg, Diagnostik und Therapie von Softwarekrankheiten, Teil 2 „Tools“, in: JavaSPEKTRUM 5/04
- [Com] Compuware, OptimalAdvisor, siehe: javacentral.compuware.com/products/optimaladvisor/
- [Cur89] M.A. Curth, M.L. Giebel, Management der Software-Wartung, Teubner 1989
- [DeM99] T. DeMarco, T. Lister, Wien wartet auf dich! Der Faktor Mensch im DV-Management, Hanser 1999 (2. Auflage)
- [Fin] FindBugs (open-source), siehe: findbugs.sourceforge.net/
- [Gel98] D. Gelernter, Machine Beauty: Elegance and the Heart of Technology (Master Minds Series), Basic Books 1998
- [Gim] Gimpel Software, PC-Lint/FlexeLint, siehe: www.gimpel.com/html/products.htm
- [IBM] IBM, Structural Analysis For Java: <http://www.alphaworks.ibm.com/tech/sa4j>
- [IEEE] IEEE Standard for software maintenance: IEEE Std 1219-1998
- [Mar02] R.C. Martin, Agile Software Development, Principles, Patterns, and Practices, Prentice Hall 2002 (1st Edition)
- [Mer95] O. Merkel, Softwarewartung, siehe: www.merkert.clarinet.de/deutscheEssays/spl.html
- [OL05] S. Opferkuch, J. Ludewig: Software-Wartung – eine Taxonomie, Softwaretechnik-Trends, Band 24 Heft 2, Gesellschaft für Informatik, 2005
- [Par] Parasoft, siehe: www.parasoft.com/jsp/products/home.jsp?product=Jtest&itemId=12
- [Par94] D.L. Parnas, Software Aging, in: Proc. ICSE 16, 16.-21. Mai 1994, Sorrento (Italien)
- [Roo04] S. Roock, M. Lippert, Refactorings in großen Softwareprojekten, dpunkt Verlag 2004
- [Sci-a] Scientific Toolworks, Understand for C/C++, siehe: www.scitools.com
- [Sci-b] Scientific Toolworks, Understand for Java, siehe: www.scitools.com/uj.html
- [Sne84] H.M. Sneed, Software Renewal – a Case Study, in: IEEE Software, Vol. 1, Nr. 3, 7/84
- [Sof] Software-Tomography GmbH, Sotograph, siehe: www.software-tomography.com
- [WR04] Vorlesungsunterlagen zu „Software Wartung und Evolution“, TU Wien, Johannes Weidl-Rektenwald, 2005, <http://www.infosys.tuwien.ac.at/Teaching/Courses/SWE/swe.html>
- [WR05] Vorlesungsunterlagen zu „Software Wartung und Evolution“, TU Wien, Johannes Weidl-Rektenwald, 2004, http://www.infosys.tuwien.ac.at/Teaching/Courses/SWE/SWE_Lecture1.pdf
- [ZE04] Thomas Zehler, Software-Visualisierung, ViSEK-Report 45D, 2004