GEFAHRENPOTENTIALE IN GROßEN JAVA-SYSTEMEN ERKENNEN UND BEHEBEN, - ERFAHRUNGEN -

Dr. Frank Simon, Dr. Dirk Meyerhoff, SQS Software-Quality-Systems AG, Köln

1 Objektorientiert: Die große Hoffnung

Anfang der 70-er Jahre wurde der Begriff der *Softwarekrise* für den Bereich der EDV geprägt, um die großen Probleme der Softwareerstellung zu verdeutlichen. Diese bezogen sich sowohl auf die frühen Analyse- und Definitionsphasen als auch auf die häufig über 70% der Gesamtkosten ausmachende Softwarewartung. Kritikpunkte waren vor allen Dingen fehlende Techniken zur disziplinierten Beherrschung komplexer Probleme und deren Bearbeitung mittels Prozessen. Ende der 80-iger Jahre hatte das objektorientierte Vorgehen einen Stand erreicht, der die Bewältigung der Softwarekrise versprach. In dieselbe Richtung zielen heute eine Vielzahl neuer, darauf aufbauender Techniken (z.B. EJB, JCA, EAI, MDA, etc.). Zusammen mit Erweiterungen auf der Prozeßseite (z.B. neue Modellierungsansätze wie OOA, OOD, einheitliche Notationen (z.B. UML) und neue Prozesse (z.B. RUP, XP)) hat dies dazu geführt, daß das objektorientierte Vorgehen mittlerweile in alle Industrie-Bereiche Einzug gehalten. Dabei tritt es häufig unter der Annahme an, die Probleme der alten Softwarekrise besser bewältigen zu können.

2 Objektorientiert: Die große Ernüchterung

Viele Großprojekte, die aufgrund der „objektorientierten Hoffnung“ mittlerweile auf Programmiersprachen wie z.B. JAVA zurückgreifen, sind von der „klassischen Softwarekrise“ in eine „objektorientierte Softwarekrise“ gestürzt. Typische Charakteristika für derartige Projekte unterscheiden sich kaum von denen der anfänglichen Softwarekrise: Zeit- und Budgetüberschreitungen, übermäßige Aufwände für Änderungen, überdurchschnittliche Abhängigkeiten zu Key-Playern, etc. [Simo01]. Diese nach wie vor gültige Problematik wird u.a. durch den CHAOS-Report 2002 bestätigt, nach dem heute nur ca. 34% aller Projekte erfolgreich, d.h. zeit- und budgetgerecht, fertiggestellt werden [Stan02].

3 Objektorientiert: Die großen Probleme

Die Begründungen für die „objektorientierte Softwarekrise“ sind vielfältig. Für eine Analyse, die jeder Verbesserung vorausgehen sollte, hat sich in vielen Projekten ein Zwei-Schritt-Verfahren bewährt:

1. Zuerst wird die Software detailliert betrachtet und nach Projektrisiken und konstruktiven Verbesserungsmaßnahmen gesucht, und
2. werden die im ersten Schritt gefundenen objektorientierten Probleme als Wirkung aufgefaßt und die dazu gehörigen Ursachen (insbesondere im Prozeßumfeld) ermittelt.

Beide Analysen wurden in vielen Groß-Projekten durchgeführt. Viele jeweils entdeckte Gefahrenpotentiale waren dabei bis zur Untersuchung unbekannt, erklärten aber im Nachhinein viele Probleme innerhalb des jeweiligen Projektes. Im folgenden werden typische Gefahrenpotentiale in großen JAVA-Systemen beschrieben und Erfahrungen bei deren Behebung berichtet.

3.1 Gefahrenpotentiale in JAVA-Systemen

Die Analyse bestehender Quelltexte ist ein hervorragendes Mittel zur Feststellung, welche Qualität die erstellten Software-Produkte besitzen [BiLeSi02]. Unter Qualität wird hierbei nicht das unmittelbare Erfüllen funktionaler Anforderungen verstanden, sondern die Analyse der internen Struktur selbst, da diese maßgeblich die Softwarekrise begründen [MeSi02], [Fent91]. Die Gefahrenpotentiale, die in vielen Projekten anzutreffen sind, betreffen unterschiedliche Abstraktionsstufen, angefangen von der einzelnen Programmierzeile, über den Entwurf bis hin zur Architektur. Einige typische Beispiele für jede dieser Stufen werden im folgenden vorgestellt:

Typische Quelltext-Probleme

Typische Beispiele auf dieser untersten Abstraktionsstufe, die in modernen Systemen häufig anzutreffen sind, entsprechen denen, wie sie bereits vor über 20 Jahren in Cobol und Fortran-Programmen anzutreffen waren. Beispiele aus untersuchten Großprojekten sind u.a.:

- In vielen Projekten wurden im Quelltext an vielen Stellen immer noch feste Literale verwendet. So konnten harte Pfadangaben (z.B. `C:\tmp\`), harte Dateiangaben (z.B. `picture.gif`) und harte Paßwörter (z.B. `pw="pwtest"`) ebenso gefunden werden wie harte Domainnamen, Portnummern oder Ausgabetexte.
- Ein Großteil des Quelltextes ist unleserlich. Dies umfaßte u.a. vollständig fehlende Kommentierung (bis zu 15% aller Dateien), falsche Einrückungen und kryptische, case-sensitive Bezeichner.

Design-Probleme

Ein Großteil der Verbesserungen durch die Verwendung objektorientierter Techniken zielt auf die Design-Ebene, da hier Klassen als Implementierung von abstrakten Datentypen realisiert werden können, mit denen dann höherwertige Konstrukte gebildet werden können. Typische Probleme auf dieser Ebene sind u.a.:

- Vererbung wurde punktuell sehr intensiv, im wesentlichen aber zu selten eingesetzt. So fand sich häufig in Vererbungsbäumen direkt kopierte Funktionalität in allen Unterklassen. Dies konnte ebenfalls für viele Klassen nachgewiesen werden, die in keiner Vererbungsbeziehung standen (z.B. mittels Code-Duplikat-Analyse). Übertiefe Vererbungsstrukturen waren häufig überabstrahiert und erschwerten deutlich die Wartung.
- Das Kapseln von Funktionalität fand häufig nicht oder nur ungenügend statt: Statt eine Datenbank (und vor allen Dingen deren interne Struktur) sauber zu kapseln, wurden häufig an verschiedenen Stellen im Quelltext eigene Queries zusammengebaut, die alle bei Datenbank-Modifikationen nachgeführt werden mußten.

Architektur-Probleme
Die Architektur stellt die höchste Abstraktionsstufe des Gesamtsystems dar. Auf sprachlicher Seite hat sich die Paketebene zur Explizierung der gewünschten Architektur durchgesetzt. Problembeispiele hierbei sind:

- Die einzelnen Entwicklungsgruppen haben i.d.R. nur jeweils ihre eigene Komponente gesehen. Mehrfachimplementierungen und Inkonsistenzen in der Programmierung waren damit unausweichlich (z.B. waren komponentenübergreifend ca. 15% aller LOC mindestens einmal kopiert).
- Eine bestehende technische Architektur wurde häufig nicht eingehalten, da die gewählte Architektur auch von niemandem überprüft wurde. Die dadurch entstehenden „unbekannten" Abhängigkeiten erhöhten jede Wartungsaktivität.

3.2 Gefahrenpotentiale beseitigen

Zusätzlich zur Analyse des Gefahrenpotential muß ermittelt werden, welche Maßnahmen für ihre Beseitigung und die Qualitätsverbesserung notwendig sind. Die Lösungsstrategien können bereits während der Analyse erarbeitet werden. Nach einer Auflistung der notwendigen, lediglich auf der Analyse basierten Reengineering-Empfehlungen müssen diese im konkreten Projekt priorisiert, geplant, durchgeführt und wiederum überprüft werden. Typische Erfahrungen während dieser Verbesserungsprozesse sind:

- In einem Projekt wurde duplizierter Code als Gefahrenpotential identifiziert. In der Planungsphase wurde dieser Punkt aufgrund damit in Zusammenhang stehender Probleme (enorme Code-Größe, lange Wartezeiten beim Ein- und Auschecken, Probleme beim Bug-Fixen, etc.) dieser Punkt entsprechend priorisiert und geplant. Mit einem Aufwand von weniger als einer Personenwoche konnte der Source-Code-Umfang um 25% von 40MByte auf 30MByte reduziert werden. Das funktionale Verhalten der Applikation blieb dabei unverändert. Typische Aktionen für dieses Reengineering waren gleiche Dateien entfernen, ähnliche Dateien zusammenführen, Vererbungsstrukturen überarbeiten und Generatoren optimieren.
- Die Konventionen, die Entwicklern auf konstruktiver Seite an die Hand gegeben wurden, waren häufig ungenügend: Zwar wurden meist ausgiebige Vorgaben gemacht, in welcher Form z.B. Kommentare vorzunehmen sind oder wie bestimmte Programmkonstrukte einzurücken sind. Höherwertige Angaben für z.B. den Entwurf fehlten aber meistens völlig. Erst eine an eine Analyse anschließende Diskussion hat innerhalb der Entwicklung konsensfähige, projektspezifische Konventionen entstehen lassen. Anschließend geschriebener Code, dem diese Konventionen zugrundegelegt wurden, war sehr viel homogener bzgl. der verwendeten Strukturen und besaß deutlich weniger Gefahrenpotentiale.
- Aufwandsschätzungen für spezielle, primär die Technik betreffenden Änderungsanforderungen konnten korrigiert werden. So wurde das Risiko einer Anforderung, eine einer Applikation zugrundeliegende Datenbank bzgl. ihrer Struktur signifikant zu verändern erst dadurch transparent werden, in dem detailliert aufgezeigt wurde, welche Aufwände mit der Anforderung verbunden sind. Die im Vorfeld durchgeführte Aufwandsschätzung stellte sich als nicht haltbar heraus.

4 Prozesse, Werkzeuge für Analyse

Das Erkennen von Gefahrenpotentialen und das Vorschlagen von konstruktiven Verbesserungsmaßnahmen wird innerhalb der SQS als Service-Leistung *SQS-Product Quality Assessment* (PQA) durchgeführt [BiLeSi02]. Der Aufwand für ein PQA für eine große JAVA-Applikation (mehrere Millionen LOC) beträgt 15 Tage. Als Werkzeuge kommen insbesondere *Quali-T* (Software-Tomography GmbH), *SNiFF+* (WindRiver), aber auch selbst geschriebene Werkzeuge (z.B. für die Code-Duplikat-Analyse) und Utilities vom Markt zum Einsatz. Der Aufwand für die projektbegleitende Überprüfung der Umsetzung der Verbesserungsmaßnahmen hängt jeweils vom Projekt ab; es haben sich allerdings wöchentliche Überprüfungen mit jeweils 1 PT bewährt.

5 Referenzen

[BiLeSi02] Bischofberger, Walter, Lewerentz, Claus, Simon, Frank: Software Quality Assessment. In Meyerhoff, Dirk (Hrsg.): Software Quality and Testing in Internet Times. Springer, Heidelberg, 2002.

[Fent91] Fenton, N.: Software Metrics – a Rigorous Approach. Chapman & Hall, 1991.

[MeSi02] Meyerhoff, Dirk und Simon, Frank: Nachhaltige Kostensenkung durch Qualitätsverbesserung in großen OO-Projekten: Erkennen von Qualitätspotenzial in großen Softwareystemen mit Hilfe von OO-Metriken. ObjektSpektrum 6/2002.

[Simo01] Simon, Frank: Meßwertbasierte Qualitätssicherung. Fakultät für Mathematik, Naturwissenschaften und Informatik, Brandenburgische Technische Universität Cottbus, 2001.

[Stan02] Standish Group: CHAOS-Report 2002