Verbessern von Code mit Refactoring

Max Kleiner

kleiner kommunikation
CH-3001 Bern
max@kleiner.com

Abstract:

Als Refactoring betrachtet man allgemein ein Vereinfachen, Verbessern und Stabilisieren einer bestehenden Codestruktur, welche keine Änderung auf das „beobachtbare“ Verhalten der Applikation und dessen Ergonomie bewirken soll.

Wie man mit Refactoring einem zugehörigen Code Review auch die sogenannten Technischen Anforderungen (Skalierbarkeit, Robustheit, Wartbarkeit und Erweiterbarkeit) erfüllen kann, will ich mit diesem Referat schrittweise aufzeigen.

“Refactoring is improving the design of code after it has been written"

1 Zusammenhang von Refactoring und Code Review aufzeigen

Das Ziel eines Code Review besteht darin, ein klares Verständnis darüber zu erhalten, wie ein Software-System strukturiert ist, was die nicht funktionalen Anforderungen beinhalten und welche Protokolle, Formate und Schnittstellen anhand der Architektur vorgesehen sind; Eine Architektur [1] oder deren Entwicklungsumgebung basiert auf: Plattform, Framework und Topologie.

Ein Code Review prüft und beurteilt die Technischen Vorgaben, die Qualität (als Fehlerminimum und Erfüllungsgrad) bezüglich Architektur und Substanzwert nach einem geplanten Refactoring. Refactoring und Code Review haben das Ziel, den jeweiligen Aufwand für die Fehleranalyse (siehe 2.1 Wartbarkeit und Testbarkeit) und funktionale Erweiterungen deutlich zu senken.

Gezeigt wird auch eine Schätztechnik, die den Substanzwert als monetäre Größe nach erfolgtem Refactoring mit Hilfe von Metriken ermittelt.

Ein Code Review Tool ist ein Werkzeug, welches nach gewissen Qualitätsregeln und Metriken den Source Code inklusive Kommentare nach einem Refactoring prüft, analysiert und dokumentiert, bspw. fehlende Ausnahmebehandlungen bemängelt, die Klassenstruktur oder Namenskonventionen überprüft!

2 Refactoring erfüllt die Technischen Anforderungen

Gezeigt wird der Zusammenhang, welche Refactoring Techniken die folgenden Technischen Anforderungen erfüllen können: Modularisierung, Erweiterbarkeit, Flexibilität, Wartbarkeit, Wiederverwendung, Testbarkeit der Spezifikation und Stabilität.
Wobei die Sekundären Anforderungen: Skalierbarkeit, Sicherheit, Redundanz und Verfügbarkeit in Wechselwirkung zu ersteren stehen.

[1] Sprachen und Architekturen wie z.B. .NET, J2EE, CLX, SOA und MDA als jüngere Beispiele

2.1 Wartbarkeit und Testbarkeit

Die Dokumentation, insbesondere eine solide und definierte Spezifikation von Schnittstellen (Interfaces), die lokale Verständlichkeit von Anweisungen resp. von Kommentar im Code und der Parameter sind das Hauptziel der Wartbarkeit, die wiederum durch Refactoring und der resultierenden Simplifikation von Code erreicht wird.

Ein in das Programm eingebaute Prüfungen der Annahmen, die man über Zustände hat (Assertions) und ein möglichst großer Umfang von automatisch ausführbaren Tests für das System (JUnit) [2] erhöhen die Testbarkeit, die ein sekundäres Ziel der Wartbarkeit darstellen und in der Regel vor dem Umbau erstellt werden.

Viele haben ja ein eigenes, „automatisiertes“ Verfahren [3]entwickelt, um Code auf Fehler hin zu durchsuchen (Testprotokolle, Stati und Userchecks), siehe Abb. 1.

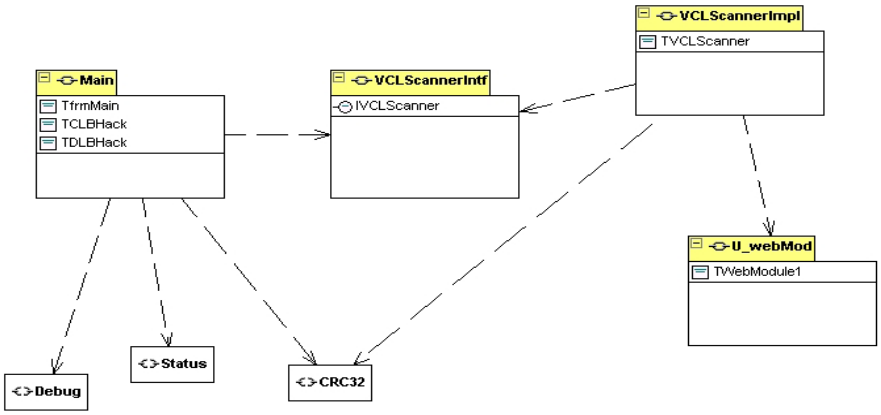

Abb.1: Aufzeigen von Unit Tests nach dem Refactoring

2.2 Refactoring und Simplifikation

Wir stehen immer noch vor der Frage, wie Prozesse der Simplifikation zu erklären sind. Und welche technischen Strukturen oder Pakete sind in besonderer Weise "anfällig" für Vereinfachungen? Um einer Beantwortung dieser Fragen näher zu kommen, sind die Technischen Anforderungen der Schlüssel. Die angewandte Simplifikation, also eine "Verschlankung" und Vereinfachung der IT-Landschaft, sowie die Konsolidierung der Daten und Funktionen auf einer einheitlichen Plattform stehen seit Jahren im Interesse vieler Firmen, doch fehlt dann der Mut die Prozesse auch wirklich zu vereinfachen. Ein Lösungsansatz lässt sich mit Patterns zeigen!

Anforderungen oder Designfragen lassen sich heutzutage mit Standards wie Patterns, Profile oder Referenzen einfacher lösen sofern man vom "not invented here" Syndrom wegkommt.

Literaturverzeichnis

[MF00] Martin Fowler: Refactoring. Wie Sie das Design vorhandener Software verbessern. Addison-Wesley Verlag, ISBN 3-8273-1630-8.
[JK01] Joshua Kerievsky: Refactoring To Patterns. Addison-Wesley, ISBN 0321213351.
[MK03] Kleiner et. al.: Patterns konkret, 2003, Software & Support Verlag, ISBN 3-935042-46-9

[2] wobei jede Sprache ein Testframework hat, DUnit, NUnit etc.
[3] In Abhängigkeit des Vorgehensmodell wie XP oder Scrum