Durchblick bei Legacy-Code

Erste Hilfe beim Refactoring

Die Arbeit an Legacy-Code gehört zu den undankbarsten Aufgaben der Entwickler. Sich in gewachsenem, meist kaum dokumentierten Quellcode zurecht zu finden, ist zeitraubend und fehleranfällig. Allerdings gibt es Werkzeuge, die dabei helfen können. Etwa, indem die Software und die zugrunde liegende Architektur grafisch so aufbereitet werden, dass ein effizientes Refactoring möglich wird.

Von Klaus Lambertz, Geschäftsführer der Verifysoft Technology GmbH

„Fehler sind das Tor zu neuen Entdeckungen." Dieses Bonmot des irischen Schriftstellers James Joyce ist sicher in vielen Lebensbereichen anregend. In der Software-Entwicklung wird man aber in der Regel gerne auf diese Art der Entdeckungen verzichten. Die Fehlerbeseitigung hat höchste Priorität. Ein Element dabei ist das so genannte Refactoring, mit dem der Code verbessert werden soll. Da Code nicht von Beginn an perfekt ist, müssen alle Teile permanenten Reviews unterzogen werden. Bei den heute üblichen agilen Methoden ist die Sicherstellung der Code-Qualität ein etablierter Bestandteil innerhalb des Software Development Lifecycles. Tests und Refactoring haben feste Plätze im Entwicklungszyklus, eine angemessene Dokumentation kann in vielen Fällen vorausgesetzt werden. Damit ist es in der Regel ohne große Probleme möglich, den vorhandenen Code zu analysieren und seine Funktionsweise zu verstehen.

Die Aufgabe des Refactorings ist es, den Code in eine für die Entwickler wünschenswerte Form zu bringen. Nur dann sind agile Methoden sinnvoll und effektiv anzuwenden. Dabei verfolgt das Refactoring zwei Hauptziele: Die Erweiterbarkeit des Codes so einfach wie möglich zu machen und gleichzeitig die Wartbarkeit zu gewährleisten. Zudem soll erreicht werden, dass der Code ganz oder in Teilen auch in späteren Projekten wiederverwendet werden kann. Im Unterschied zum Debugging hat das Refactoring allerdings keine Auswirkungen auf das Verhalten des Programms. Der Code wird nicht funktional verändert. Streng genommen werden beim Refactoring gefundene Fehler oder Sicherheitsprobleme nicht beseitigt, sondern nur für eine Bereinigung vorgemerkt. In der Praxis werden gefundene Fehler natürlich umgehend ausgebessert, die reine Lehre dient mehr der Anschauung als der praktischen Prozessgestaltung.

Immer schwerer zu durchschauen

Im Projektalltag zeigt sich jedoch häufig, dass das Refactoring – ebenso wie die Dokumentation – wegen der knappen Ressourcen, den eng getaktete Release-Terminen und schnell wechselnden Anforderungen nicht so durchgeführt werden kann, wie es wünschenswert wäre.

Doch Software reift nicht, sie altert einfach. Je länger eine Anwendung ohne Optimierung des Codes im operativen Betrieb ist, desto schwerer wird dieser Code durchschaubar. Über den Lebenszyklus einer Anwendung hinweg müssen immer wieder Änderungen und Anpassungen vorgenommen werden, die das Verhalten der Software beeinflussen. Der Aufwand bei Wartung und Modernisierung steigt rapide an, da das Wissen um die Architektur und Funktionalitäten schwindet. Der Extremfall ist Legacy-Code: Legacy-Code ist im Hinblick auf Wartung oder Erweiterung eine Herausforderung. Besonders, wenn der Code nicht im Rahmen eines agilen Entwicklungsansatzes regelmäßig dokumentiert, sondern im heute veralteten Wasserfallmodell entwickelt und über die Jahre immer wieder verändert wurde. Meist ist der Code dann extrem unübersichtlich und oft fehlen grundlegendste Dokumentationen. Die damals verantwortlichen Entwickler sind oft bereits im Ruhestand oder in alle Winde zerstreut. Dennoch muss die Software um neue, aktuelle Funktionalitäten erweitert werden. Oft sind dabei Fehler sind zu beseitigen, die bislang unentdeckt blieben. Für die damit betrauten Entwickler beginnt dann eine detektivische Spurensuche in der alten Struktur, die durchaus auch archäologische Qualitäten fordert.

Um alten Code so aufzubereiten, dass ein planvolles Refactoring mit sinnvollem Aufwand durchgeführt werden kann, sollten zuerst folgende Aspekte der Software untersucht werden:

- Dateien: Im C-Kontext sind Dateien entweder Header oder kompilierbare Dateien, also die physikalischen Komponenten der Software. Hier ist wichtig zu wissen, in welchen Relationen diese zueinander stehen - etwa, welche gemeinsamen Header diese haben.
- Subsysteme: Welche Subsysteme gibt es und in welchen Relationen stehen diese zueinander? Welche Architektur liegt den Subsystemen zugrunde?
- Datentypen: Als Typen werden üblicherweise Pointer, Enums, Classes, Structs und dergleichen bezeichnet. Hier interessieren besonders die Beziehungen zwischen Typen und Variablen.
- Funktionen: Die Aufrufhierarchie von Funktionen innerhalb eines Projekts ist elementar wichtig, um den Code zu verstehen. Dabei sollten sowohl eingehende als auch ausgehende Aufrufe berücksichtigt werden. Auch der Kontrollfluss zwischen den Funktionen ist relevant, also an welcher Stelle in eine andere Funktion gesprungen wird. Dazu wiederum müssen Verzweigungen und Schleifen im Programm bekannt sein, da diese den Kontrollfluss steuern.

Nicht manuell machbar

Diese Analysen sind ab einer gewissen Projektkomplexität manuell nicht zu leisten. Alleine die Beziehungen zwischen den unterschiedlichen Dateien eines Projekts zu erschließen ist eine fehleranfällige Fleißarbeit. Der Einsatz geeigneter Tools ist unausweichlich, um so weit als möglich automatisierbare Analysen vorzunehmen. Ein bewährtes Werkzeug für die Untersuchung von Quelltexten in C/C++ und Java ist das von Imagix entwickelte und von Verifysoft Technology vertriebene Tool Imagix 4D.

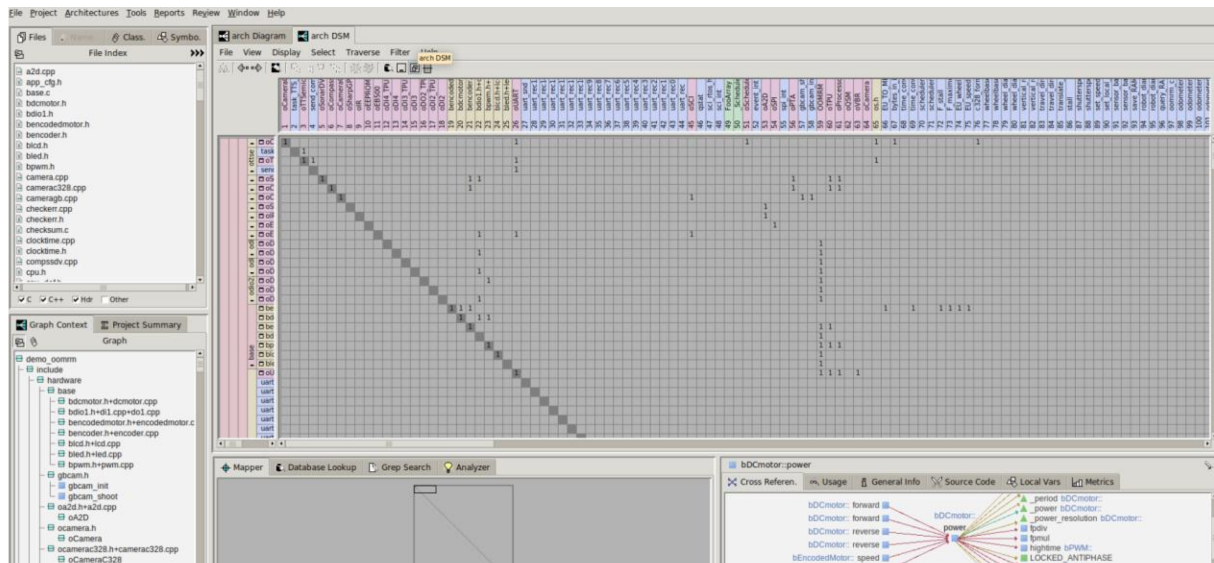

Abbildung 1: Die Design-Struktur-Matrix zeigt Beziehungen von Subsystemen der Zeilen zu Subsystemen der Spalten in variabler Auflösung

Imagix 4D analysiert den Quelltext einer Software und bereitet die für das Refactoring relevanten Informationen grafisch auf. Damit steht den Entwicklern bei Refactoring eine Repräsentation des gesamten Projekts zur Verfügung, die alle Relationen in der jeweils benötigten Detailtiefe darstellt. Je nach Fragestellung verfügt das Werkzeug über unterschiedliche Darstellungsweisen. Zum Überblick über die Abhängigkeiten aller in einem Projekt vorhandenen Subsysteme werden die Informationen in Form einer Design-Struktur-Matrix aufbereitet. Darin lässt sich zum Beispiel die Granularität der Subsysteme vom Wurzelverzeichnis bis auf die Ebene einzelner Funktionen herunterbrechen. Zum besseren Verständnis der Subsystem-Architektur wiederum kann diese als Diagramm dargestellt werden. Bei unübersichtlichen Architekturen, etwa mit sehr vielen Dateien direkt im Stammverzeichnis, helfen Filter, den richtigen Fokus zu finden. Zahlreiche weitere Ansichten, etwa für die Darstellung der Funktionsabhängigkeiten oder der Kontrollflüsse, geben den Entwicklern weitere detaillierte Informationen. Ein weiteres wichtiges Merkmal ist die Suche nach Auffälligkeiten im Code,

um gezielt die Qualität der Anwendung zu steigern. Dazu zählen unter anderem Rekursionen, Deadlocks, nicht verwendete Variablen oder unpassende Typenkonvertierungen.

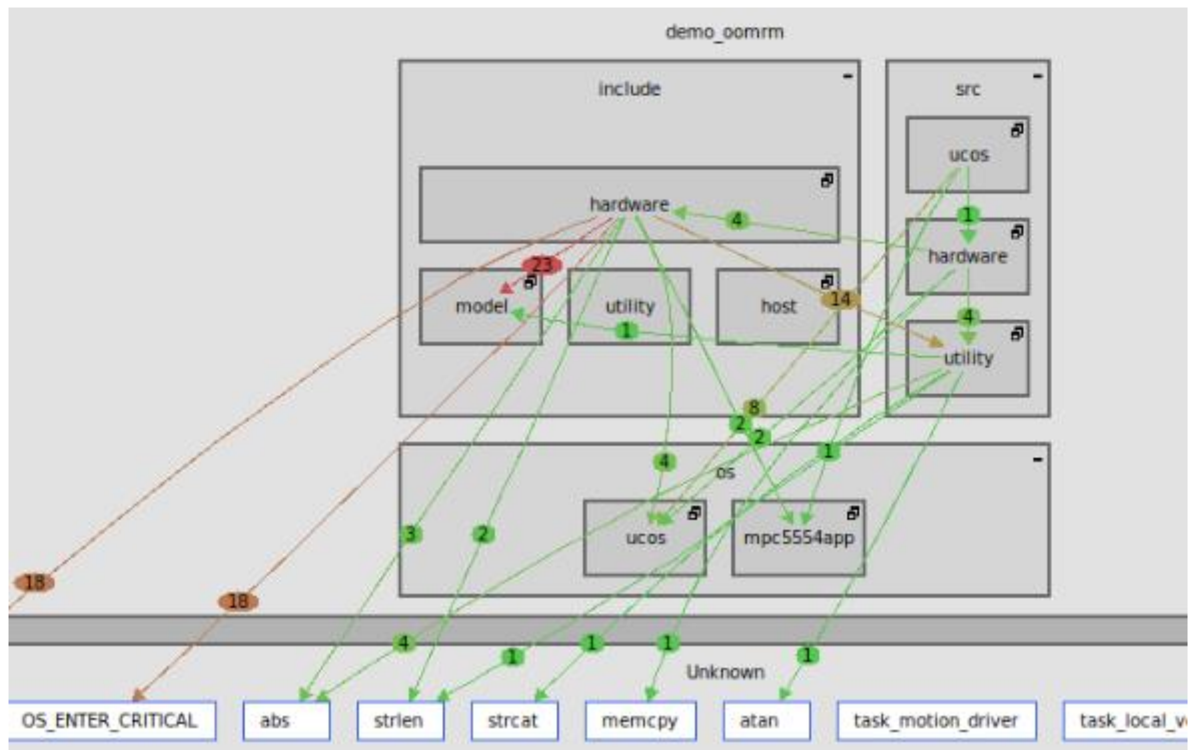

Abbildung 2: Das Subsystem-Diagramm zeigt Hierarchie und Beziehungen eines Projektes

Fazit

Das Refactoring hat sich nicht ohne Grund als integraler Bestandteil der agilen Entwicklungsmethoden etabliert. Auch Altanwendungen profitieren davon – besonders, wenn sie noch nicht am Ende ihres Lebenszyklus angekommen sind. Dafür muss der vorhandene Quellcode jedoch genau analysiert werden. Denn das Refactoring soll keinesfalls das Verhalten der Software ändern; Trial-and-Error-Ansätze sind hier fehl am Platz. Ohne geeignete Werkzeuge ist die Analyse komplexer Anwendungen kaum zu leisten, Fehler lassen sich nicht mit wirtschaftlich vertretbarem Aufwand ausschließen. Durch eine grafische Aufbereitung der Architekturen und den zugrunde liegenden Strukturen bekommen die Entwickler ein gutes Verständnis davon, wie eine Anwendung aufgebaut ist und wo die richtigen Einstiegspunkte in das Refactoring sind.