Patrick Schiel - MA97 - Fachbericht

Thema: Beschreiben Sie notwendige bzw. nützliche Eigenschaften, die ein Debugging-Tool haben sollte.

Gliederung:

Einleitung
Notwendige Eigenschaften
Nützliche Eigenschaften
Abschluss

Einleitung:

Gerade in der heutigen Softwareentwicklung ist ein modernes Debugging-Tool das wohl wichtigste Werkzeug eines Entwicklers. Die zunehmende Größe und Komplexität der Projekte steigert dabei zudem die Anforderungen an die Software.
Vor diesem Hintergrund ist es somit wichtig, sich genauer mit den Eigenschaften der doch recht zahlreichen Debugger zu befassen. Dabei ist es schwer eine Grenze zwischen notwendig und nützlich zu ziehen, da diese stetig mit den steigenden Anforderungen wandert. Ein Tool, das die gleich noch zu beschreibenden notwendigen Fähigkeiten lediglich erfüllt, ist wohl als veraltet und wenig effektiv einzuschätzen.

Notwendige Eigenschaften:

Als notwendige Eigenschaften werden die charakteristischen Merkmale eines Debuggers angesehen, die diesen als solchen kennzeichnen und von anderen Anwendungen unterscheiden.

Die Verknüpfung von Maschinen- und Quellcode ist wohl die grundlegendste Funktionalität, die geleistet werden muß, um Debugging überhaupt sinnvoll betreiben zu können. Es steht wohl außer Frage, dass man nur mit dem erstellten Quelltext und nicht mit hoch optimiertem „Maschinensalat“ arbeiten kann.
Um diese Verknüpfung zu realisieren, bringen die Compiler im übersetzten Programm zusätzliche Debugging-Informationen unter. Diese werden natürlich in der fertigen Version nicht mehr benötigt, so dass man beim Erstellen des sogenannten Release den Compiler anweisen kann, diesen zusätzlichen Ballast nicht zu produzieren.

Das Ansehen von Quellcode hat sicherlich noch nichts mit Debugging zu tun. Es dient aber dem „stepping“, einer weiteren Basiseigenschaft eines solchen Tools, als Grundlage.
Hierbei handelt es sich um das schrittweise Ausführen des Programms, Zeile für Zeile. Dadurch ist es möglich, dem Verlauf eines Programms zu folgen. Man kann erkennen, welche Abzweigungen, z.B. bei „if“ oder „switch“ in C, durchlaufen werden. Es muss aber zudem möglich sein, in einen Funktionsaufruf hinein zu gehen, sprich das „stepping“ in einer hierarchisch niedrigeren Ebene des Programms, eben innerhalb der Funktion, fortzusetzen. Aus den so gewonnenen Informationen kann der Entwickler schon recht gut einen möglichen Fehler erkennen oder zumindest einen Hinweis erhalten.

Um aber schließlich mit Sicherheit ein Fehlverhalten zu diagnostizieren, ist es oft notwendig, sich den Inhalt einer benutzten Variablen anzusehen. Dies wird durch „watching“, Betrachtung, erreicht.
Der Programmierer kann somit zur Laufzeit, also beim Debugging-Durchlauf, in eine Variable bzw. Speicherstelle hineinschauen und erhält so den aktuellen Zustand, der erreicht wird, wenn das Programm bis an diese Stelle der Verarbeitung gelangt ist.

Durch die Kombination dieser drei Grundkonzepte fürs Debugging ist es möglich, die Verarbeitung durch ein Programm sehr genau nachzuvollziehen und auf Korrektheit zu überprüfen.
Allerdings lassen sich hier bereits viele Möglichkeiten erkennen, wie man die Leistungsfähigkeit entsprechender Tools noch weiter steigern kann.

Nützliche Eigenschaften:

Die nun hier erwähnten erweiterten Eigenschaften eines Debuggers sind heute wohl auch schon größtenteils als Standard und für Großprojekte absolut notwendig zu erachten.
Zumeist handelt es sich um komfortable und zeitoptimierte Verbesserungen der Grundkonzepte.

Für das „stepping“ gibt es da einige Möglichkeiten.
So z. B. die Funktion „step to cursor“, die dem Anwender so manchen Tastendruck ersparen kann. Ist man davon überzeugt, dass Teile des Programms fehlerfrei laufen, so markiert man die Stelle, an der man interessiert ist, mit dem Cursor und gibt dem Debugger die Anweisung, solange mit der Verarbeitung fortzufahren, bis die entsprechende Position erreicht worden ist.

Um mehrere Debugging-Durchläufe in ähnlicher Weise gestalten zu können und um an mehreren Punkten zu stoppen, kann man sogenannte „breakpoints“ bzw. Haltepunkte setzen. Diese halten die Verarbeitung immer an der Stelle, an der sie eingefügt worden sind, an. Man kann also mehrere markante Bereiche des Quellcodes ohne viele Tastatureingaben sehr schnell untersuchen. Dies optimiert den Zeitaufwand für die Fehlersuche sehr stark; große Teile werden normal, ohne Unterbrechung, ausgeführt.

Da man beim „stepping“ unter Umständen in mehrere aufeinander folgende Funktionsaufrufe hineingeht, ist es hilfreich, wenn man diesen Weg in irgendeiner Form nachvollziehen kann. Um dies zu erreichen, ist der Einblick in den „call stack“, den Funktionsstapel, zu ermöglichen. Dort werden die Aufrufe nach dem FIFO-Prinzip (first in, first out) abgelegt. In C ist üblicherweise main() der erste, sprich tiefste Eintrag im Stapel.

Auch im Bereich des „watching“ hat sich, gerade im Hinblick auf fortgeschrittene Programmierkonzepte, wie z.B. Objektorientierung, einiges als nützlich oder gar notwendig erwiesen.
Als erstes ist hier wohl die Betrachtung von ganzen Objekten zu nennen. Da in modernen Programmen Variablen nicht mehr allein, sondern in einer Struktur oder einem Objekt verarbeitet werden, bringt gerade deren komplette Darstellung enorme Vorteile. Auch komplexe Strukturen, wie Listen oder assoziative Felder, sogenannte „maps“, sollten sinnvoll angezeigt werden können, da dies häufig benutzte und immer wieder vorkommende Konstrukte sind.

Ein weiteres aktuelles Konzept, das ein Debugging-Tool heute darstellen können sollte, sind „exceptions“ oder Ausnahmen. Dabei sollte es möglich sein, die Reaktionen auf nicht behandelte Ausnahmen und deren Klassifizierung zu beeinflussen. Behandelte Ausnahmen

interessieren in diesem Falle nicht, da sie zum „normalen“ Programmablauf gehören, es genügt ihr Auftreten kurz zu dokumentieren.

Bietet das Betriebssystem zudem die Möglichkeit mit einem Programm mehrere „threads“ oder Programmstränge zu kreieren, also eine (pseudo-)parallele Verarbeitung anzustoßen, so ist auch deren Überwachung eine nützliche Erweiterung eines Debuggers. Name, Priorität und Anzahl der „threads“ sind hier die wohl wichtigsten Informationen, die vom Entwickler benötigt werden.

An eher system- und hardwarenahe Programmierprojekte richten sich Funktionalitäten wie das Anzeigen der Inhalte von Registern oder dem Hauptspeicher. Eine Interpretation der Hexwerte als Zeichen durch den Debugger kann manchmal zusätzlich hilfreich sein.

Eine letzte Steigerung der Basiskonzepte eines Fehler-Tools wird durch direkte Datenmanipulation zur Laufzeit erreicht. Somit handelt es sich dann schon nicht mehr nur um ein Betrachten, sondern auch um ein Beeinflussen und Lenken.
Will man z. B. einen bestimmten Teil eines Programms während eines Durchlaufes auf jeden Fall kontrollieren, so setzt man einfach kurz vor der entscheidenden Abzweigung die nötigen Werte in die entsprechenden Variablen ein.
Diese Form der Veränderung sollte sowohl bei der Betrachtung von Objekten, Strukturen und Variablen, als auch durch direkte Beeinflussung der Register oder des Hauptspeichers möglich sein.

Hat man nun durch diese umfassenden Möglichkeiten den Fehler gefunden, so steht man vor dem Problem, dass man den Debugger beenden muss, den Quellcode ändert und einen neuen Durchlauf startet. Dies ist gerade bei nur kleinen Modifikationen, wie der Beseitigung von Schreibfehlern, recht umständlich.
Um auch hier den Zeitaufwand zu minimieren, ist die Fähigkeit, den Quellcode während des Debuggens zu verändern und sofort wirksam werden zu lassen, sehr nützlich.
Aber gerade diese Eigenschaft ist sehr von den Gegebenheiten des Betriebssystems und der Programmiersprache abhängig. Zudem können sich die Änderungen natürlich nur auf sich wiederholende Teile auswirken.

Abschluss:

Wie man an dieser Aufstellung leicht erkennen kann, ist die in der Einleitung erwähnte Schwierigkeit, notwendige von nützlichen Eigenschaften eines Debuggers zu trennen, sicherlich nicht trivial. Wie bereits gesagt steigen mit den Anforderungen an Programmierer und Programme auch die Anforderungen an das Debugging-Tool.
Daher geht der Trend zunehmend in Richtung integrierter Entwicklungsumgebungen, die den Entwickler weitestgehend unterstützen. Sie enthalten meist nicht nur den Debugger, sondern auch leistungsfähige Editoren, Archivierungs- und Versionskontrollprogramme, Klassen- und Funktions-Browser und mächtige Suchwerkzeuge für den Quellcode.
Schließlich seien noch die enormen Auswirkungen moderner Programmiersprachen erwähnt, die sowohl neue Debugging-Möglichkeiten bieten, wie z.B. das oben genannte recompilieren während des Durchlaufs, als auch diesen Tools immer mehr abverlangen, z. B. die Darstellung von Strukturen und Objekten.
Die Notwendigkeit des Debuggens selbst steht dabei gänzlich außer Frage.