Software Reuse

Elmar Stellnberger
9960069
estellnb@uni-klu.ac.at

Zusammenfassung

Reuse kann Kosten und Qualität von Software verbessern und erleichtert die Wartbarkeit. Es kann zwischen verschiedenen Formen des Software Reuse unterschieden werden. Reuse hat einen technischen sowie einen managementorientierten Aspekt und kann bereits auf dem Niveau von Programmiersprachen unterstützt werden. Allerdings muß beim Arbeiten mit großen Sammlungen von wiederverwertbaren Komponenten die Wiederauffindbarkeit gewährleistet sein, wo auch immer der Kosten-Nutzen Faktor zu berücksichtigen ist.

Erreichbare Vorteile und Motivation

Angesichts stetig fallender Hardwarekosten steht nunmehr die Entwicklung geeigneter Software im Mittelpunkt, welche in den meisten Fällen längst teurer als die Hardware kommt. Andererseits ermutigt die Verfügbarkeit leistungsfähiger Rechner auch zu immer ambitionierteren SW-Projekten. Leider schlagen aber gerade solche Großprojekte besonders oft fehl, wie das erstmals in der *Software Crisis* 1968 ganz deutlich klar wurde [10].

In der Praxis werden viele ähnliche Problemstellungen immer wieder von neuem ausprogrammiert; eine Wiederverwendung (*Reuse*) bereits vorhandener Mittel müßte demnach ein großes Einsparungspotential beherbergen. Billigere Entwicklungskosten würden in weiterer Folge wohl zu einer noch weiteren Verbreitung von IT-Anwendungen führen.

In allen anderen Ingenieursdisziplinen wird schon längst erfolgreich Reuse betrieben [4]; oder erfindet ein Architekt für jedes Haus, das er baut, den Ziegel von neuem? Ein Elektriker sucht Bauteile aus Katalogen aus, in der Automobilindustrie finden dieselben Komponenten in verschiedenen Modellen Eingang, um etwas sprechendere Beispiele zu präsentieren [2]. Warum sollte dieses Prinzip also nicht einfach auf die Entwicklung von Software übertragbar sein? Warum Sachen neu programmieren, wenn das Altbewährte bestens funktioniert?

Nehmen wir an, es gelänge eine Anwendung zu 50% aus wiederverwendeten Komponenten zu entwickeln. Man würde sich nicht nur beinahe 50% der **Entwicklungskosten** einsparen, sondern unsere Software würde auch mit einer hohen **Verläßlichkeit** aufwarten können, weil die verwendeten Programmteile bereits mehr als hinreichend erprobt und getestet worden sind [4]. Auch die **Wart**- und **Erweiterbarkeit** profitierte, weil die einzelnen Module so allgemein gehalten sind, daß sie auch bei neuen und geänderten Anforderungen mühelos einsetzbar und funktionsfähig bleiben. Letzteres ist anderenfalls nicht unbedingt garantiert, da eine Routine normalerweise nur mit Eingabeparametern „gefüttert“ wird, die sie tatsächlich vermöge ihrer

speziellen Aufgabe erwartet. Der Wartungsprogrammierer bleibt außerdem von unangenehmen Fernwirkungen weitgehend verschont, da die importierten Module entsprechend gekapselt sind. Man sieht: Wartbarkeit und Reuse ergänzen sich hervorragend.

Das eigentliche Programm wäre außerdem um die Hälfte kürzer und doppelt so übersichtlich: „Small is beautiful“ hat Erwin Schumacher einmal gesagt [5]. Noch dazu gäbe uns eine umfangreiche Bibliothek die Möglichkeit verschiedenster Lösungswege ohne großen Aufwand auszuprobieren, wie dies etwa beim *Rapid Prototyping* erforderlich ist.

Andererseits könnte die hohe Generizität mancher Komponenten die Ausführungsgeschwindigkeit verlangsamen, was aber – wie bereits im ersten Absatz erwähnt – zunehmend an Bedeutung verliert. Außerdem kann Software auch veralten. Wir dürfen eben angesichts aller Vorteile leider nicht vergessen, daß auch Software Reuse, abgesehen von den Problemen der Wiederverwendung selbst, kein ultimatives Allheilmittel gegen die weithin bekannt und gefürchtete Software Crisis ist.

Definition und Einteilung

Es gibt eine Reihe von Definitionen für Reuse; ich habe hier nur diese eine ausgewählt, weil sie uns einen impliziten Hinweis über die Unterscheidbarkeit von Wiederverwendung gibt [2]:

„Reuse ist die Verwendung bereits vorhandener Mittel und Konzepte in einer neuen Situation.“

Die Unterscheidung wird zwischen Mitteln und Konzepten getroffen; konkret ergeben sich in unserem Fall die folgenden beiden Punkte:

(1) Wiederverwendung von Ideen und Wissen

(2) Wiederverwendung spezieller Artefakte und Komponenten.

Eine andere Einteilung wird in [9] getroffen:

(1) Daten (standardisierte Dateiformate)

(2) Requirements [12]

(3) Architektur (gröbere Designmerkmale) und (detaillierteres) Design

(4) Code (Quellcode, compilierter Code)

Wobei ich mich hier im speziellen auf (2) bzw. (4) konzentrieren werde, obwohl vieles auf alle Arten von Reuse gleichermaßen zutrifft.

Ad-hoc versus Systematic Reuse

Zu **Ad-hoc** [3] oder opportunistischem Reuse kommt es spontan seitens von Einzelpersonen als auch in Kleingruppen. Man verwendet eine Routine, die zufällig schon irgendwo vorhanden ist, in einem anderen Kontext. Diese wird aber meistens sehr speziell an eine bestimmte Problemstellung gebunden und nur schwer zu adaptieren sein. Auch das bloße Ablegen von Komponenten in einer Bibliothek zur späteren Wiederverwertung kann noch als

Ad-hoc Reuse bezeichnet werden. Es handelt sich in diesem Fall um relativ elementare Prozeduren zur Symbolmanipulation, zum Formatieren von Ausgaben und anderem, die ein paar Zeilen Quellcode sparen, das Programm etwas übersichtlicher machen und in quasi jeder Anwendung zum Einsatz kommen können. Nachteil ist, daß der Programmierer über die Existenz aller Funktionen, die er verwenden will, bescheid wissen muß. Solche Bibliotheken sind in ihrer Größe stark eingeschränkt und zudem oft nicht einmal portabel (d.h. sie funktionieren auf anderer Hardware bzw. einem anderen Betriebssystem nicht). Beispiele dazu wären nicht nur Eigenproduktionen sondern auch Standardbibliotheken, die gewöhnlich beim Erwerb einer Entwicklungsumgebung mitgeliefert werden. Bestimmt haben sie, sofern sie irgendwann einmal programmiert haben, diese Art des Reuse schon kennengelernt.

Systematic Reuse [1] ist im Gegensatz dazu immer an ein bestimmtes Anwendungsgebiet gebunden (*domain specific*) und beschäftigt sich mit der Wiederverwendung von Komponenten höheren Niveaus, die eine große Menge an Funktionalität in sich vereinen (*high level live cycle artefacts*). Erst solche Eigenschaften machen Reuse betriebswirtschaftlich relevant und ermöglichen die Anwendung in größerem Maßstab. Das geht so weit, daß man meistens das „Systematic“ wegläßt und nur mehr von Reuse spricht.

Einzelne Firmen stehen innerhalb eines Anwendungsgebietes miteinander in Konkurrenz. Das „domain understanding“ ist hier ein wesentlicher Erfolgsfaktor. Systematic Reuse erfolglos zu versuchen kostet wertvolle Zeit und Ressourcen; davon abzulassen ist ebenso riskant, denn wenn die Konkurrenz dabei erfolgreich ist, können Marktanteile oder gleich ein ganzer Markt für das Unternehmen verloren gehen.

Organisation: Systematic Reuse

Folgende Faktoren sind für die erfolgreiche Durchführung zu beachten:

(1) Management

(2) Meßbarkeit (Zielerreichung)

(3) Gesetz, rechtliche Bestimmungen

(4) ökonomische Aspekte

(5) Design

Systematic Reuse muß vom Management, speziell auch vom oberen unterstützt werden. Das umfaßt die Einschulung des Personals, das Einrichten von zusätzlichem Anreizen und Belohnungen für die Beteiligten, welche nicht selten auch Vorbehalte gegen eine Einführung haben, und einen allgemeinen Einfluß auf die Unternehmenskultur, um hier vom Schaffen technischer Vorraussetzungen einmal abzusehen. Jedenfalls ist eine längerfristiges Verfolgen dieser Ziele nötig, da erst einmal eine Bibliothek mit Modulen aufgebaut werden muß.

Gemessen werden kann z.B. das *reuse level*, das Verhältnis von wiederverwendeten zu tatsächlich vorhandenen Komponenten. Das *external* reuse level mißt im Gegensatz zum *internal* die Wiederverwendungen außerhalb des ursprünglichen Systems. Komponenten, die nicht mehr gebraucht werden, müssen selbstverständlich aus der Datenbank entfernt werden. Welche Eigenschaften eine Komponente brauchbar machen ist ebenfalls interessant. Oft macht sich ein Trend hin zu generischen, vielseitig verwendbaren Komponenten bemerkbar.

Gesetzliche Rahmenbedingungen sind relevant, wenn Komponenten von einem Unternehmen an ein anderes verkauft werden. Die Rechte und Verantwortung beider Seiten müssen klargestellt werden. Der überbetriebliche Austausch ist jedoch sehr heikel: So war ein Betrieb weltspitze in einer Compilertechnologie, konnte aber sein Wissen nicht verkaufen, wie in [10] zu lesen ist. Auch die Bemühungen eines gewissen Brad Cox haben hier ins leere geschossen.

Eine Kosten-Nutzen Rechnung ist unbedingt durchzuführen; manche Komponenten müssen bis zu 13 mal wiedereingesetzt werden, bevor sie sich rechnen.

Beim Design unterscheidet man zwischen *domain analysis* und *domain implementation*, wobei ersteres auf die Analyse des Anwendungsgebietes abzielt und zweiteres auf die technische Realisierung.

Neben einem funktionierenden Bibliothekssystem wird in [6] auch die Einrichtung folgender Arbeitsgruppen empfohlen:

(1) Managmentunterstützungsgruppe (Initiativen, Budget, Strategien)

(2) Qualifikationsgruppe (stellt Qualität der Komponenten sicher.)

(3) Erhaltungsgruppe (Wartung)

(4) Entwicklungsgruppe

(5) Reuser-Unterstützungsgruppe (Personaltraining, Tests, Meßungen)

Black vs. Whitebox Reuse [5]

Beim Blackbox Reuse weiß der Reuser nichts über die interne Struktur, wohingegen der Whitebox Ansatz auf dem Verständnis und der Änderung von Quellcode beruht.

Sprachliche Unterstützung

Auch Programmiersprachen können ihren Beitrag zur Reusability leisten. In diesem Zusammenhang sind vor allem die Errungenschaften der Objektorientierung zu nennen [11]. Der Benutzer braucht nichts über die interne Struktur eines Objekts zu wissen und kann sein Verhalten durch das Ableiten einer Subklasse an die gegebene Problemstellung anpassen.

Dazu möchte ich gerne einen praktischen Hinweis geben: Virtuelle Methoden (in C++ und Pascal als virtual deklariert, in Modula3 unter Overrides aufgeführt) verlangsamen zwar bei schlechter Sprachimplementierung die Ausführung, sind aber an manchen Stellen für Erweiterbarkeit und Flexibilität unbedingt notwendig, damit beim Austauschen einer Subklassenmethode nicht auch andere neu geschrieben werden müssen!

Außerdem stellt ein Objekt ein übersichtliches Methodeninterface zu einer gegebenen Problemstellung zur Verfügung. Dieses sollte so gestaltet sein, daß häufig gebrauchte und einfache Operationen leicht anzuwenden sind. Sprachen wie etwa Eiffel versuchen nicht nur durch Objektorientierung, sondern auch durch Spezifikation von Ein- und Ausgangsbedingungen (*Assertions*) einzelner Methoden zum Reuse zu ermutigen. Das weit verbreitete Pointer-Konzept, das einem das Leben schwer macht, wenn man eine statische Datenstruktur dynamisch machen will, widerspricht z.B. auch dem Gedanken der

Wiederverwendung, ärgert aber nur beim Whitebox Reuse wirklich. Auf andere Kleinigkeiten im Sprachdesign will ich hier nicht näher eingehen.

Ein ganz anderer Ansatz ist es, keine Bibliothek von Modulen aufzubauen, sondern gleich eine eigene Programmiersprache oder einen Anwendungsgenerator zum Problemgebiet zu schreiben. Diese Methode ist vorerst mit einer Kraftanstrengung verbunden, kommt dem Entwickler aber aufgrund der Einheitlichkeit mehr entgegen und soll zudem nicht unrentabel sein [1]. Hier hat man von außen einen riesigen Block, ein einziges Modul, wie dies etwa auch bei einem DBMS (Datenbankmanagementsystem) der Fall ist; man sagt: die *Granularität* ist hoch.

Geeignete Sprachen können einem das Leben zwar in vieler Hinsicht leichter machen, leztendlich kommt es aber auf die Konsequenz des Programmierers an gut strukturierte Programme zu schreiben und von besonders unangenehmen Sorten der Kopplung nach Möglichkeit abzulassen.

Schritte zur Wiederverwendung einer Komponente [2]:

(1) Auffinden

(2) Schnittstelle und Funktionalität studieren bzw. Quellcode durchsichten

(3) Subklasse ableiten bzw. Quellcode anpassen

Schritte (2) und (3) sind bei White- und Blackbox Reuse leicht verschieden. Schritt (3) entfällt, wenn die Anforderungen an die gesuchte Komponente denen der gefundenen gleichen, was aber in den seltensten Fällen zutreffen wird. Daneben gibt es auch noch zwei Strategien um zu suchen, die auch kombiniert werden können [8]:

(1) Retrieval: Queries und Anfragen werden an das System gerichtet.

(2) Browsing: Der Benutzer navigiert durch eine Hierarchie von Komponenten, was unter anderem mit dem bottom-up-Approach gut harmoniert.

Beim Sichten des Quellcodes hängt der Aufwand einerseits von der Erfahrung des Reusers, andererseits von den Programmcharakteristiken, nämlich Größe, Komplexität, Dokumentation und Sprache ab.

Es ist nicht nur wichtig, bloß Komponenten zu finden, sondern auch den Anpassungsaufwand abschätzen zu können, um die mit dem minimalen auszuwählen. Eine grobe Selektion nimmt das System selbst vor, dem Anwender ist eine übersichtliche Liste zur Auswahl bereitzustellen, deren Länge innerhalb gewisser Grenzen liegen muß, was die Unterscheidbarkeit sehr ähnlicher Module als auch das automatische Generalisieren von Anfragen (Queries) nötig macht.

Enumerative, facettenorientierte Klassifikation [6]

Die einzelnen Eintragungen in die Reuse-DB (*Repository*) müssen nach einem System geordnet und klassifiziert werden. Dabei unterscheidet man zwischen *hierarchischen* und *syntaktischen* Beziehungen. Eine Hierarchie basiert auf Unterordnung von Klassen während syntaktische Beziehungen Elemente zu Klassen zusammensetzen.

Die traditionelle *enumerative Klassifikation* kommt mit hierarchischen Beziehungen aus, was aber nicht für alle Problembereiche eine befriedigende Lösung darstellt: Soll die Physiologie der Gattung untergeordnet werden oder will man doch lieber ein physiologisches Merkmal verschiedener Tierarten miteinander vergleichen? Die Antwort kann hier nicht gegeben werden, denn es handelt sich um eine von den bevorzugten Zugriffswegen abhängige Designentscheidung.

Faceted Classification orientiert sich dagegen an den einzelnen Attributen. Eine neue Klasse kann jederzeit durch Einschränkungen auf gewissen Attributen erzeugt werden, sodaß größtmögliche Flexibilität gewährleistet ist. Dieser Ansatz kommt aus den Bibliothekswissenschaften und kann durch eine relationale Datenbank umgesetzt werden.

Relevante Attribute

Welches sind nun die relevanten Attribute, mit denen sich das Gesuchte identifizieren läßt? Neben den bereits erwähnten Eigenschaften zur Abschätzung des Adaptionsaufwandes, werden hier wichtige Attribute zur Identifikation einer Komponente vorgestellt.

Die Was-Wo-Wie Regel gibt eine grobe Unterscheidung, wobei sich das Was auf die Funktion, das Wo auf Medium und Plattform beziehen kann und das Wie auf Implementation und Leistungsfähigkeit. Einen detaillierteren Vorschlag gibt [2]:

<Funktion, Objekt, Medium>

Funktion bezieht sich auf die Operation, Objekt auf den elementaren Datentyp, Medium auf Datenstruktur oder -strom.

z.B.: <input,characters,buffer>, <compare,lines,file>

<Systemtyp, Funktionsgebiet, Einsatzort>

Weitere Attribute definieren die Umgebung, in der das Modul arbeitet.

Ein voller Deskriptor besteht dann aus einem Sechstupel:

<add, integers, array, matrix inverter, simulation, aircraft manufacturing>

Thesauri, Konzeptuelle Distanzgraphen, 1D-Listen [1]

Daneben gibt es noch Thesauri, um Synonyme zu identifizieren sowie Distanzgraphen und eindimensionale Listen, um die semantische Ähnlichkeit von Begriffen festzustellen. In

eindimensionalen Listen sind zwei Begriffe verwandt, wenn sie nahe beieinander stehen, während Distanzgraphen eine Baumstruktur aufweisen, in der Begriffe Blätter entsprechen und kürzeste Wege semantischer Ähnlichkeit.

Queries

Natürlich könnte man Anfragen an die Reuse-Datenbank in gewöhnlichem SQL richten. Abgesehen vom Schreibaufwand gibt es dort keine Unterstützung für das automatische Erkennen verwandter Begriffe oder um zu kontrollieren, daß eine brauchbare Anzahl an Datensätzen zurückgegeben wird. Außerdem ließe sich Fuzzy Logic gut dazu einsetzen, um den Adaptionsaufwand im Einzelfall abzuschätzen oder bestimmte Unifikationen hervorzuheben und zu gewichten. Erfahrenen Programmierer finden z.B. komplexere Module immer noch einfach, dieselbe Komponente wird in Assembler mehr LOC (Lines Of Code) haben. Jedenfalls muß man zwischen der Zweckdienlichkeit (*relevance*) und Sachdienlichkeit (*pertinence*), der Brauchbarkeit für den einzelnen Benutzer, unterscheiden.

Hier ist ein Vorschlag für die Syntax einer solchen Query in BNF aus [7]:

Query ::= AQT | AQT $\wedge$ Query

AQT ::= Attribute Weight CutOff ListOfKeyPhrases

ListOfKeyPhrases ::= KeyPhrase | KeyPhrase $\vee$ ListOfKeyPhrase

ListOfKeyPhrases gibt eine Liste von Werten an, die für eine Attributausprägung in Frage kommen. Ein AQT (Attribute Query Term) legt zusätzlich fest, wie wichtig ein Merkmal für die gesamt Anfrage ist. Soll z.B. in erster Linie nach einer Spezies oder nach einem physiologischen Merkmal gesucht werden, um beim selben Beispiel zu bleiben? Die gesamte Anfrage schließlich überprüft mehrere Attribute auf ihre Ausprägung. Mehrere Anfragen können letztendlich noch durch mengentheoretische Operatoren wie „$\cap$“, „$\cup$“ bzw. „\“ verknüpft werden.