# Refactoring

## *Begriffsbestimmung und Einsatzgebiete*

*„Never change a working program“*.

Dieses Sprichwort blieb lange Zeit unangefochten. Denkt man jedoch etwas genauer über diese Aussage nach, so wird einem unweigerlich klar, daß dieser Wunsch weit hinter den realen Anforderungen, die heutzutage an eine Software gestellt werden, zurückbleiben muß.

Erfolgreiche Anwendungen sollen erweitert werden. In Verwendung befindliche Programme müssen andauernd kleineren Anpassungen und Änderungen unterzogen werden. Letzteres wird Software Maintenance genannt und verschlingt bis zu 50% der Kosten im gesamten Softwarelebenszyklus. Ein Buchhaltsystem ist z.B. laufend an die neuesten Gesetzesnovellen zu akkomodieren.

Der britische Grundsatz „never change a winning horse“ mag zwar für Pferdewetten ein erfolgversprechender Leitgedanke sein, in einem Geschäftsfeld, das nur von der ständigen Fort- und Weiterentwicklung lebt, scheint er jedoch sein Ziel zu verfehlen. Deshalb werde ich in diesem Artikel das sog. „Refactoring“ als Art radikale Antithese zum Grundsatz der minimalst möglichen Änderung vorstellen:

*Unter Refactoring versteht man die Umgestaltung der internen Struktur eines Programmes ohne dabei sein beobachtbares Verhalten zu ändern.*

Diese Aussage charakterisiert schon recht gut, was mit Refactoring gemeint ist; und zwar geht man in der Regel von einem korrekten und lauffähigen System aus und versucht durch die gezielte Anwendung einer Reihe elementarer Umformungen das Design sohingehend zu verbessern, daß künftige Änderungen und Erweiterungen möglichst einfach durchzuführen sind.

Überlegen wir uns jetzt einmal vom Wortstamm her, was der Begriff Refakturierung(eng.: Refactoring) ausdrücken könnte: Refactoring kommt von re-facere(lat.) und bedeutet auf Deutsch soviel wie nocheinmal - bearbeiten/gestalten. „Facere“ heißt in der Grundbedeutung eigentlich machen, steht aber unter vielem auch für durchführen und vollenden, was mir in diesem Zusammenhang besonders gut gefällt.

Die Idee das System nicht als ganzes umzuwerfen und möglichst sofort und direkt in den gewünschten Zustand zu versetzen sondern dabei gezielt, strukturiert und eben schrittweise vorzugehen soll die Möglichkeit, daß sich im Laufe des Wandlungsprozesses Fehler einschleichen, auf ein Minimum reduzieren.

Ein einzelner Schritt kann dabei etwas so einfaches wie die Umbennung einer Variablen sein. Als Programmierer haben wahrscheinlich auch Sie schon einmal kleinere Änderungen wie diese vorgenommen, weil es kaum möglich ist ein komplizierteres Programm sofort aus dem Gedächtnis hinzuschreiben. Gute Programmierer schauen auch darauf nach getaner Arbeit die Übersicht im Programm wiederherzustellen. Mittels Refactoring soll es nun auch möglich werden, etwas tiefergreifende Modifikationen, welche Designentscheidungen betreffen, vorzunehmen. Leider kann auch eine relativ einfache Umformung wie die Umbenennung einer Variablen aufwändiger als erwartet ausfallen, da nicht selten

gleichnamige Bezeichner auch anderswo im Programm existieren oder sogar in einem darunterliegenden Gültigkeitsbereichen redeklariert sind, weshalb eine Unterstützung durch die Entwicklungsumgebung oder durch externe Werkzeuge sehr wünschenswert wäre. Allerdings kommt man mit der Zeit allemal in Übung (Für eine vollständigere Liste von Refactorings möchte ich in diesem Zusammenhang auf Abschnitt XXX verweisen).

Die Umgestaltung der internen Struktur erfolgt häufig dem Design in Hinblick auf Wartung und Erweiterung wegen, was jedoch nicht heißen soll, daß man beim Refactoring nicht auch andere Ziele wie die Verbesserung von Leistung und Performanz ebenso wie rein technische Belange nämlich u.a. die Anpassung an eine neue Umgebung oder Sprache im Sinn haben kann. Sinngemäß müssen nicht nur die Ergebnisse weiterhin korrekt berechnet werden, sondern es dürfen auch spezielle durch die Systemumgebung auferlegte nichtfunktionale Anforderungen oder Korrekteheitsbedingungen nicht plötzlich verletzt werden:

- die Einhaltung von Zeitschranken in sogenannten Echtzeitsystemen
- in sicherheitskritischen Systemen: Kosistenzbedingungen an die Ablauflogik, welche ein Eindringen durch gezielte Störung vermeiden sollen; Verschlüsselung und ähnliches
- in embedded Systems: Batterielebensdauer, begrenzter Speicher, Leistung usw.
- andere nichtfunktionale Anforderungen: Verfügbarkeit, Ausfallssicherheit, Verläßlichkeit und Fehlertoleranz neben Wartbarkeit, Performanz, Ressourcensparsamkeit und Benutzbarkeit.

Die Refakturierung gibt uns nunmehr die Möglichkeit das Design im nachhinein zu verbessern. Im idealtypischen Softwareentwicklungsprozeß sind alle Designentscheidungen vor der Codierung zu treffen. Wozu also das ganze?

Ein anfangs festgelegtes Design, so ausgefeilt und gut durchdacht es auch sein mag, verfällt im Laufe der Zeit. Anpassungen und Änderungen werden oft ohne hinreichende Rücksichtnahme auf das Design vorgenommen. Der Wartungsprogrammierer sieht häufig nur jenen Teil, den er ändern soll; sein Einblick in die übergeordnete Gesamtstruktur ist sehr begrenzt. Außerdem ist es schlichtweg unmöglich, bereits beim Design alle in Zukunft möglichen Änderungen vorauszusehen oder miteinzubeziehen. Unvorhergesehene Neuerungen lassen sich aber nur allzuoft nicht gut in das bestehende System eingliedern, was unangenehme Effekte wie die (bewußte oder unbemerkte) Verdoppelung von Code und der damit einhergehenden später notwendigen Ausbesserung an vielen verschiedenen Punkten im Programm mit sich ziehen kann, obwohl nur eine einzige Eigenschaft des Programms umzustellen wäre (Eine vollständige Liste aller Strukturmängel findet sich in Kapitel XXX). Demnach scheint der allmähliche aber stete Strukturverfall in Softwareprodukten ein unausweichliches Naturgesetz zu sein, dem es nichts entgegenzusetzen gibt (2. Hauptsatz: Die Entropie nimmt im abgeschlossenen System fortwährend zu.). Somit wäre das Ende im Softwarelebenszyklus eine vorprogrammierte Sache; spätestens wenn die Erhaltungskosten die Ausgaben für eine Neuentwicklung übersteigen, muß die nunmehr spaghettierte Software abgewrackt werden! – oder doch nicht? In schlimmen Fällen kann durch schlechtes Design oder unvorhergesehene Ereignisse die Entropie auch schon während der Entwicklung so weit ansteigen, daß ein Weiterarbeiten unmöglich wird, wie dies Martin Fowler an einem Beispiel aus eigener Erfahrung in der Einleitung seines Buches beschrieben hat [XXX].

Es gibt jedoch eine kleine Chance: Wenn nämlich Programmierer mit neuem Elan und Kenntnissen im Bereich der Refakturierung ins Projekt kommen; dieserfalls darf nämlich das Projekt nicht länger als abgeschlossenes System betrachtet werden. Refactoring kann man sozusagen als Umkehrung des allmählichen Strukturverfalles begreifen, so unglaublich das im ersten Moment auch klingen mag.

Wie bereits angesprochen ist die Refakturierung nicht nur im Bereich der Wartung und Erhaltung interessant, sondern kann auch bei der Entwicklung sehr hilfreich sein. Trotzdem kommt im traditionellen Softwareerstellungsprozeß nichts vor, was auch nur irgendwie in diese Richtung gehen würde. Im Wasserfallmodell muß das gesamte Design auf einmal direkt nach der Anforderungsanalyse festgelegt werden, was man sich vom Topleveldesign nur wünschen kann, im Detail jedoch häufig etwas zu weit greift. Das Prototyping gesteht zwar ein, daß bei technischen Neuentwicklungen die Erfahrung und der Weitblick oft fehlen, um ein gutes Design ex ante hervorbringen zu können, versucht jedoch nichts gegen den sich nur allzuleicht einschleichenden Strukturverfall zu tun, sondern entwickelt gleich in der Absicht den Prototypen später wegzuwerfen und nocheinmal ganz von vorne anzufangen.

In der iterativen, risikogetriebenen Softwareentwicklung, wird jede Phase (also Anforderungsanalyse, Design, Implemtierung und Test) mehrmals durchlaufen, was realitätsnäher ist, da der Kunde nicht selten noch während der Entwicklung mit neuen Wünschen oder Änderungsbestrebungen kommt und weil jetzt bereits gewonnene Erfahrung mit dem System für das Design der neu hinzuzufügenden Komponenten von Nutzen sein kann. Alles mit der Bestrebung am Ende jedes Durchlaufes ausführbaren und bereits einsatzfähigen Code herauszubekommen. Obwohl dieser Ansatz wesentlich flexibler als das Wasserfallmodell ist, wird man auch hier immer wieder feststellen müssen, daß bereits getroffene Entscheidungen in der aktuellen Situation für das weitere Vorgehen nicht ideal sind. Die strikte und unnachgiebige Durchsetzung des TopDown-Paradigmas erlaubt jedoch auch hier kein Einlenken: Was einmal festgelegt worden ist, ist einzementiert und darf nie wieder revidiert werden. Um dennoch auf Neuerungen gut vorbereitet zu sein, wird die Software durch Indirektionen und eine Reihe weiterer Designentscheidugen so flexibel wie möglich gehalten. Ist trotz allem ein Weiterarbeiten ohne Umformung nicht möglich, so wird das eingangs erwähnte Prinzip der minimalst möglichen Änderung angewendet.

Neben der kommerziellen Entwicklung hat sich aber ganz unanbhängig davon die Kultur der Open-Source Entwicklung und des Hackens entfalten. Sie verfolgt den BottomUp-Ansatz. Der Hacker setzt sich vor den Computer, beginnt an einem Programm zu basteln und probiert solange herum, bis er es zum Laufen bringt. Dabei scheut er nicht davor zurück, jede Änderung, die ihm in den Sinn kommt und einen Vorteil, welcher Art auch immer, verspricht gleich durchzuführen und sofort auszuprobieren. Böse Zungen behaupten, ein Code der auf solche Art und Weise zustandekommt, müsse einfach total chaotisch und unkontrollierbar sein; viele Hacker scheinen jedoch das Gegenteil zu beweisen: Sie kommen am Ende mit einem Stück gut strukturierter Software zurück. Wie kann das sein? Sucht man dem Hacker diese Frage zu stellen, so wird man nur selten eine befriedigende Antwort erhalten. Die Art und Weise, in der er seine Programme entwickelt, ist nämlich nicht auf Generalplänen akribisch verzeichnet. Vielmehr spiegelt sich in jeder seiner Entscheidungen die Summe seiner praktischen Erfahrungen wieder. Nun haben Hacker keinen guten Ruf und deshalb ist es, glaube ich, auch noch zu früh einen Zusammenhang zwischen Hacken, BottomUp-Programmierung und Refactoring zu suchen (Martin Fowler hat dies auch aus gutem Grund in seinem Buch tunlichst vermieden). Eigenwillige Hacker sind nämlich der Alptraum jedes Managers, denn sie lassen sich keinen strikten Zeit-, Termin- oder Vorgehensplänen unterwerfen.

Es läßt sich jedoch nicht leugnen, daß tausende über den ganzen Planeten verstreute Nebenerwerbs-Hacker mit Linux ein Betriebssystem erschaffen haben, das mittlerweile zur Spitzenklasse gehört. Der Glaube, daß es eine kritische Komplexitätsstufe gebe, ab der es ohne zentralisiertem Ansatz mit sehr genauer Vorplanung einfach nicht mehr geht, scheint damit besiegt zu sein. Das Zauberwort, das es so vielen Programmierern ermöglicht hat produktiv zusammenzuarbeiten heißt Open Source Developement. Im Artikel „The Cathedral and the Bazaar“ werden die Unterschiede zwischen der proprietären (Kathedrale) und der Open-Source-Entwicklung (Basar) sehr gut herausgearbeitet. Jedenfalls haben sich mit Linux

unzählige freiwillige Programmierer zusammengefunden, um dem Microsoft-Imperium den Kampf anzusagen. Ziemlich erfolgreich sogar, denn Linux gewinnt, seitdem es auf praktisch jedem PC lauffähig ist, immer weiter an Beliebtheit.

Die Frage, die sich jetzt stellt, ist, ob sich auch die kommerzielle Softwareerzeugung von den Praktiken dieser Gemeinschaft etwas abschauen kann. Ein Projektleiter oder Manager mag dies sofort verneinen, denn der Ressourceneinsatz für die Entwicklung muß planbar sein. Die Mitarbeiter eines Softwarehauses arbeiten dort nicht ehrenamtlich und wollen am Ende des Monats ihren Gehalt ausbezahlt bekommen und der Kunde wird bestimmt ungeduldig, wenn das Projekt schon fertig sein sollte, aber noch nichts läuft und keine genauen Angaben über den Projektfortschritt existieren (Ja der Kunde wird auch dann ungeduldig, wenn das Programm einen Tag später fertig werden würde, er es aber nicht weiß!). In der kommerziellen Entwicklung besteht die Möglichkeit nicht ein Projekt zweimal durchzuführen, weil die Mittel sehr knapp bemessen sind und sich auch die Aktionäre oder Eigentümer ihren Gewinnanteil oder Mehrwert erwarten! Anders jedoch bei der OpenSource-Entwicklung: Hier arbeiten viele Gruppen von Hackern gleichzeitig an der Lösung desselben Problems, ohne vielleicht vorerst voneinander zu wissen. Schlußendlich setzen sich die praktikabelsten Ansätze durch. Dies mag durch die Konkurrenz auch am freien Markt gegeben sein, doch kommt es hier niemals zur gegnseitigen Befruchtung, da keine Firma ihre Software gratis zur Weiterentwicklung hergibt und niemand bereit ist die Katze im Sack zu kaufen. Wenn wir jedoch nur an die Form, in der die Entwicklung organisiert ist, denken greift das vielleicht etwas zu kurz. Es ist aber auch nicht ganz einfach, wenn nicht sogar unmöglich die Praktik des Hackens zu erfragen. Vielleicht hilft ja in diesem Fall die eigene Programmiererfahrung oder dieser Artikel weiter.

Eine neuartige Praxis in der organisierten Anwendungsentwicklung stellen die sogenannten leichtgewichtigen Entwicklungsprozesse wie Softwareexpeditionen oder Extreme Programming dar. Diese werfen nicht das Instrument der Planung über Bord, welches für kommerzielle Projekte zweifelsohne unverzichtbar ist, gestehen aber ein, daß nicht alle Möglichkeiten gleich von Anfang an vorhergesehen oder miteinbezogen werden können und lassen eine größere Flexibilität zu. Stellt sich heraus, daß das Projektteam den falschen Weg gegangen ist, dürfen Entscheidungen auch bezüglich des Designs revidiert werden. Damit man solcherfalls nicht von vorne oder von dem Punkt, an dem man abgezweigt ist, neu anfangen muß, kann man den bestehenden aber lauffähigen Code refakturieren und so in den gewünschten Zustand bringen. Hat man es einmal so weit kommen lassen, daß die Vererbungshierarchie im Argen liegt und Code mit ähnlichen Aufgaben über das ganze Programm verstreut ist, so muß man einige Energie investieren, um das Design wiederaufzurichten. Traditionelle Prozesse sind dafür besonders anfällig, da man sich der Möglichkeit der Umgestaltung ganz verweigert oder sie so lange wie nur möglich hinauszögert. Gelingt es jedoch die Refakturierung als Bestandteil des Softwareerstellungsprozesses zu integrieren spart man sich einigen Aufwand. Laufende kleinere Anpassungen sind nicht so kostspielig und riskant wie ganz große Umkrempelungen. Soll die Software um eine neue Fähigkeit erweitert werden, so ist es vielfach leichter zuerst zu restrukturieren und dann zu erweitern. Mit dieser Vorgehensweise ist es nicht mehr unerlässlich von Anfang an so viel an Flexibilität einzubringen wie nur irgendwie möglich. Zusätzliche Indirektionen, Delegationen und Aufspaltungen, die im derzeitigen Entwicklungszustand gar noch nicht gebraucht werden, erschweren nur das Verständnis des bestehenden Codes. Darüberhinaus bringt eine ex-ante Flexibilisierung immer auch unnotwendige und später vielleicht nie genutzte Alternativen ein. Ebenso kann die Umgestaltung von Code bei dessen Verständnis weiterhelfen und dazu beitragen Fehler aufzudecken. Für eine vollständige Integration in die Programmerstellung sind auch Reviews zu beachten. Hier können Änderungsvorschläge eingebracht oder diskutiert werden.

Wer mit der Planung von Software betraut ist, sollte vielleicht eingestehen, daß sich auch der genialste und weitsichtigste Plan in Zukunft irgendwann einmal erschöpft haben wird, weil auch das Umfeld, in dem das Informationssystem eingebettet wird, einer beständigen Veränderung unterliegt. Geben wir also auch unseren Programmen die Chance zu evolvieren und sich der aktuellen Lage bestmöglich anzupasssen. Die Alternative immer wieder von neuem anzufangen hat zwar auch ihren Reiz, weil ein System, solange es noch neu ist, von seinen Gründern leichter verstanden werden kann, wirft aber dabei eine bestehende und funktionsfähige Lösung einfach über Bord.

## *Ein konkretes Beispiel*

Betrachten wir nun ein praktisches Beispiel, welches Entlehnungen in einer Bibliothek verwaltet. Wir starten mit einer einzigen Routine, welche dem Kunden seine Entlehnungen in Rechnung stellt. Die Routine soll so erweitert werden, daß der Entlehnungsbericht sowohl als einfacher Text als auch in Html-Format ausgegeben werden kann.

Ich werde hier dasselbe Beispiel in Object Pascal und Java geben, weil ich einerseits glaube, daß viele Refactorings in Pascal einfacher zu verstehen und leichter durchzuführen sind und weil Java andererseits zu einer der am weitesten verbreiteten Sprachen gehört. Viele Umgestaltungen, die in Java in nur einem Schritt durchgeführt werden müssen, können in Pascal weiter aufgedröselt werden. Zudem vereint Object Pascal die Möglichkeiten blockstrukturierter, prozeduraler Sprachen sowie das Konzept der Objektorientierung, welches relativ unverändert aus C++ übernommen worden ist und somit auch dem nicht mit Pascal vertrauten Leser geläufig sein dürfte.

Werfen wir nun einen Blick auf die bestehende zu erweiternde Pascalroutine zur Anzeige des KundenEntlehnungsBerichtes.

```
TYPE
  TBuch = record
            type BuchArt = (Kinderbuch, reguläres, NeueAusgabe);
            var
              art    :BuchArt;
              titel  :String;
          end;

  TEntlehnung = record
                  buch         : ^TBuch;
                  dauerinTagen : Word;
                end;

  TKunde = record
             name      : String;
             ausleihen : Array Of TEntlehnung;
           end;



function KundenEntlehnungen(IN k:TKunde) : String;
  var dieses,insgesamt : Double;   //Preis für Entlehnung
      StammKundenBonus : Word;
      txt              : String;
  begin
    txt := ' Ausleihen von '+k.name+': '+#10;
    FOR i:=low(k.ausleihen) TO high(k.ausleihen) DO WITH k.ausleihen[i] DO begin
      dieses:=0;
      CASE buch^.art OF
        reguläres  :  begin  inc(dieses,2);
                             IF dauerinTagen>2 THEN inc(dieses, (dauerinTagen-2)*1.5); end;
        NeueAusgabe : inc(dieses,dauerInTagen*3);
```

```
       KinderBuch  : begin inc(dieses,1.5);
                           IF dauerinTagen>2 THEN inc(dieses, (dauerinTagen-2)*1.5); end;
      END;
      inc(StammKundenBonus);
      IF (buch^.art=NeueAusgabe) AND (dauerInTagen>1) THEN inc(StammKundenBonus);
      Concat(txt,' '+buch.name+':'+#9+dieses+'€');
      inc(insgesamt,dieses);
    end;
    Concat(txt,'geschuldeter Betrag : '+insgesamt+'€ '+#10);
    Concat(txt,'Sie haben sich '+StammKundenBonus+' Bonuspunkte erworben
  end;
```

Betrachtet man diese Prozedur, so fällt als erstes auf, daß hier implizite Details über die GebührenVerrechnung (CASE buch^.art OF) in die Routine zur Erstellung des Kundenberichtes eingearbeitet worden sind. Die Verrechnung sollte aber eigens gekapselt werden, weil diese Funktionalität wahrscheinlich unabhängig vom Ausdrucken des kundenspezifischen Entlehnungsberichtes an verschiedenen Stellen im Programm wie bei der Buchhaltung, Kostenrechnung oder beim Preisanschlag Verwendung finden wird; gleiches gilt für die pro Entlehnung gutgeschriebenen Bonuspunkte: raus damit!

```
function EntlehnungsGebühr(ausleihe:TEntlehnung):Currency;
  with ausleihe do begin
     EntlehnungsGebühr:=0;
     CASE buch.art OF
       reguläres  :  begin  inc(EntlehnungsGebühr,2);
                            IF dauerinTagen>2 THEN inc(EntlehnungsGebühr, (dauerinTagen-2)*1.5); end;
       NeueAusgabe : inc(EntlehnungsGebühr,dauerInTagen*3);
       KinderBuch  : begin inc(EntlehnungsGebühr,1.5);
                           IF dauerinTagen>2 THEN inc(EntlehnungsGebühr, (dauerinTagen-2)*1.5); end;
     END;
  end;



function EntlehnungsBonus(ausleihe:TEntlehnung):Byte;
  begin
    IF buch.art=NeueAusgabe AND dauerInTagen>1 THEN EntlehnungsBonus:=2
                                               ELSE EntlehnungsBonus:=1;
  end;
```

Nun haben wir aus der ursprünglichen KundenEntlehnungsFunktion einiges herausgezogen, was dort bestimmt nicht hingehört. Machen wir uns also an die Erweiterung! Eigentlich muß man jetzt nur noch die verbleibende Routine kopieren, abändern und einer übergeordneten Funktion das gewünschte Ausgabeformat (unformatierter Text oder Html) übergeben. Was dabei herauskommt kann ich gleich zeigen:

```
function KundenEntlehnungen(IN k:TKunde;fmt:TextFormat) : String;
  var txt         : String;
      insgesamt        : Double;   //Preis für Entlehnung
      StammKundenBonus : Word;

  procedure HtmlTxt;
    begin
    txt := ' <h1>Ausleihen von <em>'+k.name+'</em></h1> '+#10;
    FOR i:=low(k.ausleihen) TO high(k.ausleihen) DO WITH ausleihe=k.ausleihen[i] DO begin
      Concat(txt,' '+buch.name+':'+#9+EntlehnungsGebühr(ausleihe)+'€ <br>');
      inc(insgesamt,EntlehnungsGebühr(ausleihe));
      inc(StammKundenBonus,EntlehnungsBonus(ausleihe));
    end;
    Concat(txt,'<p>geschuldeter Betrag : <em>'+insgesamt+'€</em> '+#10);
    Concat(txt,'<p>Sie haben sich <em>'+StammKundenBonus+'</em> Bonuspunkte erworben</p>');
    end;

  procedure KlarTxt;
    begin
```

```
    txt := ' Ausleihen von '+k.name+': '+#10;
    FOR i:=low(k.ausleihen) TO high(k.ausleihen) DO WITH ausleihe=k.ausleihen[i] DO begin
      Concat(txt,' '+buch.name+':'+#9+EntlehnungsGebühr(ausleihe)+'€');
      inc(insgesamt,EntlehnungsGebühr(ausleihe));
      inc(StammKundenBonus,EntlehnungsBonus(ausleihe));
    end;
    Concat(txt,'geschuldeter Betrag : '+insgesamt+'€ '+#10);
    Concat(txt,'Sie haben sich '+StammKundenBonus+' Bonuspunkte erworben');
    end;

  const ErzeugeText : Array [fmt] Of Procedure = (HtmlTxt,KlarTxt);
  begin
    insgesamt:=0; StammKundenBonus:=0;
    KundenEntlehnung := ErzeugeText(fmt);
  end;
```

Statt zwei Prozeduren HtmlTxt und KlarTxt anzulegen wäre es vielleicht besser nur eine Routine zu haben und diese mit einer Liste von Formatierungen für die einzelnen Textbereiche wie Überschrift, Textkörper und Rechnungsbetrag zu parametrisieren, wobei Klartext eine Liste von Leerstrings übergeben bekommt, weil ja hier keine Formatierungsanweisungen in den Text eingestreut werden sollen. Dieserfalls müßten künftige Änderungen im Wortlaut des Textes nur ein einziges mal durchgeführt werden. Vielleicht wäre es eine praktikable Lösung für jede Sprache ein eigenes Unterprogramm und für jede Formatvorlage einen geeigneten Parameter an das Unterprogramm zu übergeben. Bestimmte Ausgabeformate verlangen es aber vielleicht die Formatanweisungen vom Klartext zu trennen, weshalb ich es hier bei den beiden Prozeduren HtmlTxt und KlarTxt bewenden lassen will. Eine weitere Flexibilisierung hätte gewiß ihren Preis und kann an den Bedürfnissen der als nächstes durchzuführenden Änderung sehr leicht vorbeigehen. Stattdessen würde ich lieber in einem weiteren Schritt Details der Berechnung von der eigentlichen Ausgabe trennen. Das Aufsetzen der Schleife (for) sowie den schleifenspezifischen Code über der Gesamtheit aller Ausleihen (inc) kann von einer gemeinsamen Routine übernommen werden.

```
function KundenEntlehnungen(IN k:TKunde;fmt:TextFormat) : String;
  var GesamtGebühr     : Double;
      StammKundenBonus : Word;
  type AusleihenProc = procedure(var ausleihe:TEntlehnung);

  procedure FürAlleAusleihen(BearbeiteAusleihe:AusleihenProc);
    begin
      FOR i:=low(k.ausleihen) TO high(k.ausleihen) DO WITH ausleihe=k.ausleihen[i] DO begin
        BearbeiteAusleihe(ausleihe);
        inc(GesamtGebühr,EntlehnungsGebühr(ausleihe));
        inc(StammKundenBonus,EntlehnungsBonus(ausleihe));
    end end;

  // ***

  var txt    : String;

  procedure PlainTxt;
    procedure ZeigeEntlehnung(var ausleihe:TEntlehnung) begin
      Concat(txt,' '+buch.name+':'+#9+EntlehnungsGebühr(ausleihe)+'€'); end;
    begin
      txt := ' Ausleihen von '+k.name+': '+#10;
      FürAlleAusleihen(ZeigeEntlehnung);
      Concat(txt,'geschuldeter Betrag : '+GesamtGebühr+'€ '+#10);
      Concat(txt,'Sie haben sich '+StammKundenBonus+' Bonuspunkte erworben');
    end;

  procedure HtmlTxt;
    procedure FormatiereEntlehnungAlsHtml(var ausleihe:TEntlehnung) begin
      Concat(txt,' '+buch.name+':'+#9+EntlehnungsGebühr(ausleihe)+'€ <br>') end;
    begin
      txt := ' <h1>Ausleihen von <em>'+k.name+'</em></h1> '+#10;
```

```
    FürAlleAusleihen(FormatiereEntlehnungAlsHtml);
    Concat(txt,'<p>geschuldeter Betrag : <em>'+GesamtGebühr+'€</em> '+#10);
    Concat(txt,'<p>Sie haben sich <em>'+StammKundenBonus+'</em> Bonuspunkte erworben</p>');
  end;

const ErzeugeText : Array [fmt] Of Procedure = (HtmlTxt,PlainTxt);
begin
  GesamtGebühr:=0; StammKundenBonus:=0;
  ErzeugeText[fmt];
  KundenEntlehnung := txt;
end;
```

Jener Code der vor dem Aufruf an FürAlleAusleihen() steht umfaßt die Kopfzeile jener danach die Fußzeilen. Die Auflistung aller Entlehnungen muß von eigens benannten Subroutinen (ZeigeEntlehnung, FormatiereEntlehnungAlsHtml) übernommen werden, da Pascal keine Lambda-Ausdrücke verarbeiten kann. Mittels Lambdakalül, welches derzeit leider nur von Sprachen des funktionalen Paradigmas (Lisp,…) unterstützt wird, könnte der Code von ZeigeEntlehnung direkt als Parameter an FürAlleAusleihen übergeben werden.

Soweit so gut. Nehmen wir einmal an, daß der Code, den wir gerade bearbeitet haben, sich schon seit 20 Jahren im Einsatz befindet. Nun soll das bestehende lokale Entlehnungssystem mit allen österreichischen Bibliotheken vernetzt oder an das übergeordnete Betriebsinformationssystem angebunden werden. Dieses ist dem Zeitgeist entsprechend objektorientiert programmiert; möglicherweise in unterschiedlichen Sprachen: C++, Delphi und Java. Die einzelnen Programme können aber über Windows, Corba oder welche Interfaces auch immer miteinander kommunizieren. Um auch unser Entlehnugssystem einzubinden muß es dem Paradigma der Objektorientierung unterworfen werden. Um uns keine unnotwendige Arbeit zu machen konvertieren wir unsere Anwendung in Object Pascal, einer Sprache, die den Vorteil hat, daß jedes Pascal-Programm ohne jedwede Änderung compiliert.

Die ohenhin überfällige Umwandlung in objektorientiertes Design wäre hier auch so, galube ich, keine schlechte Idee. Ersetzen wir das Schlüsselwort „record“ durch „class“ und fügen wir die Signaturen aller Prozeduren mit TEntlehnung als Parameter(also EntlehnungsGebühr und EntlehnungsBonus) der Entlehnunsklasse hinzu und machen wir aus KundenEntlehnungen eine Methode von Kunde. Fertig! Das Programm wird fehlerlos übersetzt. Ich möchte es trotzdem nicht vorzeigen. Für einen Außenstehenden verhaltet sich jetzt alles so, als ob er es mit einem Objekt zu tun hätte, im Inneren jedoch regiert nach wie vor prozedurales Design.

Sicheres Indiz für prozedurales Design ist immer eine Case-Anweisung auf eine Typvariable (BuchArt). In der Objektorientierung löst man dieses Problem gewöhnlich durch die Bildung von Subklassen und dem Aufruf virtueller Methoden: Jede Klasse ist für ihre Fähigkeiten ganz alleine verantwortlich und kapselt den Code des entsprechendeen Zweiges der Case-Anweisungen in einer eigenen Methode. Daß eine Superklasse das Verhalten einer Subklasse im Vorhinein festlegt, ist in der Objektorientierung nicht zulässig. Stattdessen wird das Verhalten geerbt und dann angepaßt oder überschrieben. Das Ableiten von Subklassen soll auch dann möglich sein wenn die Superklasse nur in compilierter Form vorliegt; also ohne Änderung an der Superklasse.

Viele Programmierer während jetzt vielleicht geneigt je nach Art der Gebührenverrechnung entsprechende Subklassen von Büchern für Bestseller, normale Büchern und Kinderbücher abzuleiten; dies ist jedoch keine gute Lösung. Die Gebührenverrechnung sollte als eigene aggregierte Klasse gekapselt werden, denn vielleicht

will man später neben Büchern auch noch Zeitschriften, Kassetten und Platten zur Entlehnung freigeben ohne daß dabei die Vererbungshirarchien für die entlehnbaren Artikeln und jene für die Gebührenverrechnung kollidieren. Es gibt jedoch noch einen anderen triftigen Grund für eine eigene Gebührenverrechnungsklasse: In Object Pascal ist es zwar möglich den Typ eines Objekts während seiner Lebenszeit mit „Buch:=Regulaeres(Buch);“ zu ändern, man erhält aber dabei nicht ohne Grund eine Warnung vom Compiler! Nehmen wir an das Buch hatte vorhin den Typ `NeueAusgabe`; würde `Regulaeres` zusätzlich zu den in der Superklasse definierten Datenobjekten neue Felder definieren, so wäre deren Wert nach der forcierten Typumwandlung undefiniert; umfaßten reguläre Bücher mehr Datenfelder als neue Ausgaben, so endete der nächste Zugriffsversuch auf die neuen typspezifischen Felder erwartungsgemäß mit einer Zugriffsverletzung. Deshalb sind Typumwandlungen während der Objektlebenszeit in Java generell verboten, sodaß sich diese Frage erst gar nicht stellt (die einzige Möglichkeit wäre es ein neues Objekt zu erstellen und die Datenwerte zu kopieren).

Schauen wir uns jetzt gleich einmal an, was bei der Umwandlung in Objekte herausgekommen ist:

```
UNIT Bibliothek;

INTERFACE

     Currency = Word;


TYPE
  TBuchArt = class
               class function Gebühr(dauer:Word):Currency;virtual;abstract;
               class function Bonus(dauer:Word):Byte;virtual;
             end;



  TBuch = class
            type TBuchArt = class
               class function Gebühr(dauer:Word):Currency;virtual;abstract;
               class function Bonus(dauer:Word):Byte;virtual;
             end;
            var
             titel  :String;
             Gebührenverrechnung :BuchArt;
          end;

  BuchArt = class of TBuch.TBuchArt;

  Kinderbuch  = class(TBuchArt) class function Gebühr(dauer:Word):Currency;override;end;
  Regulaeres  = class(TBuchArt) class function Gebühr(dauer:Word):Currency;override;end;
  NeueAusgabe = class(TBuchArt) class function Gebühr(dauer:Word):Currency;override;
                                class function Bonus(dauer:Word):Byte;override;end;



  TEntlehnung = class
                  buch         : TBuch;
                  dauerinTagen : Word;
                public
                  constructor Create(buch:TBuch;dauer:Word);
                  function Gebühr:Currency;
                  function Bonus:Byte;
                end;

  AusleihenProc = procedure(var ausleihe:TEntlehnung);

  TKunde = class
             name      : String;
             ausleihen  : Array Of TEntlehnung;
           protected
             procedure FürAlleAusleihen(BearbeiteAusleihe:AusleihenProc);
             function GesamtGebühr:Currency;
             function StammKundenBonus:Byte;
           public
```

```
            constructor Create(name:String);
            function HtmlAuszug:String;
            function TextAuszug:String;
          end;


// ------------------------------------------------------------------------------------------- //

IMPLEMENTATION



constructor TEntlehnung.Create(buch:TBuch;dauer:Word);
  begin self.buch:=buch; self.DauerInTagen:=dauer end;

{function TEntlehnung.Gebühr:Currency;
  var Gebühr:Currency;
  begin
    Gebühr:=0;
    CASE buch.art OF
      regulaeres  :  begin  inc(Gebühr,2);
                            end;
      NeueAusgabe : inc(Gebühr,dauerInTagen*3);
      KinderBuch  : begin inc(Gebühr,15);
                          IF dauerinTagen>2 THEN inc(Gebühr, (dauerinTagen-2)*15) end;
    END;
  end;}

class function KinderBuch.Gebühr(dauer:Word):Currency;
  var Gebühr:Currency;begin Gebühr:=2; IF dauer>2 THEN inc(Gebühr, (dauer-2)*15) end;

class function Regulaeres.Gebühr(dauer:Word):Currency;
  begin Gebühr:=dauer*2 end;

class function NeueAusgabe.Gebühr(dauer:Word):Currency;
  begin Gebühr:=ifthen(dauer>2, 15, 15*(dauer-1)) end;


class function TBuchArt.Bonus(dauer:Word):Byte; begin Bonus:=1 end;
class function NeueAusgabe.Bonus(dauer:Word):Byte; begin Bonus:=ifthen(dauer>1, 2,1) end;

// ------------------------------------------------------------------------------------------- //

function TEntlehnung.Gebühr:Currency; begin buch.GebührenVerrechnung.Gebühr(dauerinTagen) end;
function TEntlehnung.Bonus:Byte;      begin buch.GebührenVerrechnung.Bonus(dauerinTagen) end;


// ------------------------------------------------------------------------------------------- //


constructor TKunde.Create(name:String);
  begin self.name:=name end;


  procedure TKunde.FürAlleAusleihen(BearbeiteAusleihe:AusleihenProc);
    begin FOR i:=low(k.ausleihen) TO high(k.ausleihen) DO BearbeiteAusleihe(ausleihen[i]);end;

  function TKunde.GesamtGebühr:Currency;
    procedure SumGebühr;begin inc(GesamtGebühr,ausleihe.Gebühr);end;
    begin
      GesamtGebühr:=0;
      FürAlleAusleihen(SumGebühr);
    end;

  function TKunde.StammKundenBonus:Byte;
    procedure SumBoni;begin inc(StammKundenBonus,ausleihe.Bonus);end;
    begin
      StammKundenBonus:=0;
      FürAlleAusleihen(SumBoni);
    end;

// ***


function TKunde.TextAuszug:String;
  var txt:String;
  procedure ZeigeEntlehnung(var ausleihe:TEntlehnung) begin
```

```
    Concat(txt,' '+buch.name+':'+#9+EntlehnungsGebühr(ausleihe)+'€'); end;
  begin
    txt := ' Ausleihen von '+name+': '+#10;
    FürAlleAusleihen(ZeigeEntlehnung);
    Concat(txt,'geschuldeter Betrag : '+GesamtGebühr+'€ '+#10);
    Concat(txt,'Sie haben sich '+StammKundenBonus+' Bonuspunkte erworben');
  end;

function TKunde.HtmlAuszug:String;
  var txt:String;
  procedure FormatiereEntlehnungAlsHtml(var ausleihe:TEntlehnung) begin
    Concat(txt,' '+buch.name+':'+#9+EntlehnungsGebühr(ausleihe)+'€ <br>') end;
  begin
    txt := ' <h1>Ausleihen von <em>'+name+'</em></h1> '+#10;
    FürAlleAusleihen(FormatiereEntlehnungAlsHtml);
    Concat(txt,'<p>geschuldeter Betrag : <em>'+GesamtGebühr+'€</em> '+#10);
    Concat(txt,'<p>Sie haben sich <em>'+StammKundenBonus+'</em> Bonuspunkte erworben</p>');
    HtmlAuszug:=txt;
  end;
```

Kinderbuch, Reguläres und NeueAusgabe sind Subklassen von BuchArt und definieren jeweils ihre eigenen EntlehnungsGebühr, wohingegen standardmäßig für jede Ausleihe gleich viele Bonuspunkte anhand der Superklassenmethode BuchArt gutgeschrieben werden außer bei Büchern der Klasse NeueAusgabe. Wollte jetzt eine der Routinen TextAuszug oder HtmlAuszug die Gebühr einer Entlehnung ermitteln, so müßte es hierzu folgende Nachrichtenkette verwenden: `ausleihe.buch.GebührenVerrechnung.Gebühr(ausleihe.dauerinTagen)`. Eine derartige Delegation ist nicht nur abenteuerlich lang, sondern auch sehr unübersichtlich und schlecht gekapselt. Deshalb gibt es die Methode `ausleihe.Gebühr`.

Betrachten wir schlußendlich noch was aus FürAlleAusleihen geworden ist: Diese Prozedur hat eigentlich drei verschiedene Dinge auf einmal erledigt: den gutgeschriebenen StammKundenBonus und die gesamten Entlehnungsgebühr berechnet sowie die Auflistung jedes ausgeborgten Buches anhand der übergebenen Routine. Im Zeichen der Objektorientierung wurden folglich die Methoden Gesamtgebühr und StammkundenBonus von FürAlleAusleihen abgespalten, was zur Folge hat, daß dieselbe Schleife nunmehr dreimal durchlaufen werden muß. Alle drei Methoden sind als protected gekennzeichnet, sodaß sie zwar von abgeleitete Klassen nicht aber von außen aufgerufen werden dürfen, zumal sie derzeit nur intern von HtmlAuszug und TextAuszug gebraucht werden.

Der objektorientierte Code ist nicht unwesentlich länger als der gleichwertige prozedurale Code; währenddessen mögen die langen Methodensignaturen der Subklassen von TBuchArt, die auch gleich ein zweites mal bei der Klassendeklaration aufgeführt werden müssen und dann nur eine einzige kleine Anweisung enthalten sowie die Aufspaltung der Entlehnungsschleife in drei Teile den prozeduralen Programmierer befremden, leidet doch die Effizienz unter den vorgenommenen Änderungen. In der Tat verweilt der Rechner jedoch bei fast allen Anwendungen in über 90% der Ausführungszeit in nur einer einzigen kleinen Funktion! Es lohnt sich diese aggressiv zu optimieren und den Rest des Programms lieber nach Gesichtspunkten wie Verantwortlichkeit, Aufgabe und Funktionalität zu gliedern.

Ich habe anfangs versprochen dasselbe Beispiel auch in Java zu bringen. Dieses findet sich im Anhang. Hier sind nur der erste und der letzte Auszug enthalten, weil die Schachtelung von Unterprogrammen sowie die Verwendung von Prozedurvariablen in Java nicht möglich sind. Außerdem muß die Klasse Gebührenverrechnung instanziert werden, weil Java keine Klassenreferenzen zur Verfügung stellt. Statt einfach buch.GebührenVerrechnung:=KinderBuch schreiben zu können muß in Java ein leeres Objekt vom Typ KinderBuch erzeugt werden: buch.GebührenVerrechnung=new KinderBuch(). Eine

Klassenreferenz ist eine Variable in welcher ein Typ gespeichert werden kann. Eine Typvariable vom Typ class of TBuchArt kann jeder beliebige Subtyp von Buchart, also TBuchArt selbst oder NeuesBuch, Reguläres oder KinderBuch zugewiesen werden. Klassenreferenzen werden oft Klassenmethoden übergeben, welche dann bei Bedarf eine Instanz des angegebenen Typs erzeugen. Ein funktional äquivalentes Pendant zur Klassenmethode mit Klassenreferenzparameter stellen sogenannte Factory-Methoden in Java dar, welche mittels switch/case-Anweisung die gewünschte Instanz erzeugen. In diesem Beispiel wurden Typvariablen eingesetzt, um Klassen mit nur einer einzigen Instanz zu erzeugen, indem alle Variablen als „class“ (Java: „static“) deklariert worden sind.

Die Java-Version des Programms kommt ohne FürAlleAusleihen-Methode aus. Die Möglichkeit die der Anleihenliste zugrundeliegende Datenstruktur zu ändern ist dort ohnehin durch die Nutzung des Iterator-Interfaces gegeben. Um in Object Pascal das Iterator-Interface nutzen zu können, müßte das Array in einer Klasse gekapselt werden. Pascal bietet allerdings (wie neuerdings auch C#) darüberhinaus noch die Möglichkeit den Zugriff auf eine Arrayvariable mittels Eigenschaften auf Methodenaufrufe umzuleiten. Das Java Beispiel entspricht sehr genau dem Programm aus [XXX].

Ich habe TBuchArt im obigen Beispiel als aggregierte Teilklasse von TBuch dargestellt; diese Möglichkeit bietet aber leider keine derzeit verfügbare Implementierung von Object Pascal; ebenso gestattet es leider nicht jeder Pascal Compiler lokale Unterroutinen als Variablenparameter zu übergeben, wie dies beim Aufrufen von FürAlleAusleihen ausgeschöpft wird. Im Anhang findet sich deshalb eine Variante des Programms, die sich in jeder Umgebung problemlos übersetzen lassen sollte.

Damit hätte ich ein oder im Grunde zwei Beispiele zur Refakturierung gebracht. Zuerst habe ich das Design von KundenEntlehnung für die Erweiterung zur Ausgabe von Berichten im Html-Format aufbereitet; daraufhin habe ich gezeigt wie es möglich ist ein prozedurales Programm objektzuorientieren. Die Umwandlung in objektorientierten Code bringt nicht nur syntaktische Änderungen mit sich, sondern bedeutet einen Paradigmenwechsel. Ein solches Refactoring kann bei größeren Anwendungen erheblichen Aufwand verursachen.

Ich habe bisher die Frage geklärt, was Refactoring eigentlich ist, wie es sich in die Softwareentwicklung integrieren läßt und habe jetzt auch ein paar Beispiele für Refakturierungen gebracht. Viel wichtiger als die tatsächliche Durchführung ist es aber zu erkennen, wann und wo eine Umgestaltung angebracht oder zielführend ist. Das nächste Kapitel soll dabei helfen Designmängel aufzuspüren, indem es einige der häufigsten Ungereimtheiten aufzeigt. Will man dann tatsächlich eine Änderung vornehmen, so kann man sich anhand der im darauffolgenden Kapitel vorgestellten Einzelschritte orientieren. Weil es eine ganze Menge elementarer Refactorings und auch nicht wenige Möglichkeiten gibt wie das Design in das Suboptimale abgleiten kann, habe ich beides auf einem Handzettel, welcher auf ein A4-Blatt paßt, zusammengefaßt. Diesen kann man bei der täglichen Programmierarbeit hervorkramen, um nach etwas Passendem zu suchen.

## *Designmängel*

Zur Frage wann geändert werden soll meint Kent Beck [XXX]: „If it stinks, change it“.

Unter Gerüchen kann man sich eben mehr vorstellen als unter vager Ästhetik. Die meisten Menschen entwickeln mit zunehmender Erfahrung ein ziemlich gutes Gefühl für gut oder schlecht strukturierten Code. Übler Spaghetticode kann auf den Leser ebenso penetrant wie

intensiver Gestank einwirken. Vom Gefühl, daß hier etwas nicht stimmt bis zur konkreten Einsicht ist der Weg aber nicht immer offensichtlich. Folgende Punkte sollen dabei helfen festzustellen, worum es sich gerade handelt. Die einzelnen „Gerüche“ habe ich weitgehend übernommen. Wirklich neu ist nur die Einteilung in 5 Gruppen, welche ein wenig Übersicht in die unzählbar vielen Aromen bringen soll.

1. doppelter Code (Duplicate Code)

Nummer eins in der Stinkparade ist doppelter Code. Er ist es auf jedenfall Wert eliminiert zu werden. Die offensichtlichste Methode dies zu tun, ist eine Routine zu extrahieren (I.1). Meist unterscheiden sich die Stellen nicht in allen Details, weshalb vor der Ersetzung geeignete Parameter festgelegt werden müssen. Ist der Code Teil einer Schleife, so besteht die Möglichkeit die Schleife wie im Beispiel (FürAlleAusleihen) als ganzes gleich mitzudestillieren und den verbleibenden Rückstand später als (optionalen) Prozedurparameter wiedereinzuführen. Anfangs befindet sich das neue Unterprogramm im lokalen Gültigkeitsbereich der Routine, aus der sie extrahiert worden ist. Durch „Gültigkeitsbereich erweitern“ (II.2) kann sie bis zur Solitärprozedur entkoppelt werden, um ihr daraufhin eine neue Heimat in einem neuen Kontext (Klassen- oder Schachtelungskontext) zuzuweisen. Wo das neue Unterprogramm hingehört, muß der Programmierer entscheiden; das hängt ganz von der zur Verfügung gestellten Funktionalität ab.

2. überlange Methode (Long Method)

Überlange Methoden oder Prozeduren sind ebenfalls Kandidaten für die Aufspaltung (I.1). Meistens implementieren diese innerhalb verschiedener Bereiche ganz unterschiedliche Problemstellungen. Besteht die Chance, daß die dabei gewonnen Teilprogramme später in einem weitreichenderen Kontext nocheinmal verwendet werden können nicht, so sollte man diesen Schritt trotzdem nicht scheuen. Die Bezeichner der aus dem Prozedurkörper herausdestillierten Fraktionen fungieren nämlich als implizite Kommentare. Was in der Hauptroutine zurückbleibt hat mehr Gewicht; diese, nunmehr kurz und bündig, sagt mehr darüber aus was getan werden soll ohne den Leser gleich mit den Details der Durchführung zu konfrontieren.

3. konfuse Klasse

Im Laufe der Zeit sammeln sich in den verwendeten Klassen immer mehr und mehr Methoden an. Will nun jemand, der nicht mit der Implementierung dieser Klasse betraut worden ist, eine Nachricht an eine Instanz dieser Klasse schicken, so wird er schnell von der Vielzahl geerbter und neu definierter Fähigkeiten überwältigt. Oftmals sind nur die Dokumentationen der einzelnen in alphabetischer Reihenfolge aufgelisteten Methoden verfügbar, welche dann auch noch kopierten Text enthalten. Stattdessen sollte man lieber eine ganzheitlichere Sicht auf die Klasse und ihre häufigsten Verwendungsmuster bieten.

Konfuse Klassen sind genauer auf ungenutztes Erbe (iii.4), den Gültigkeitsbereich und die Sichtbarkeit (v.1, V.2) ihrer Eigenschaften und Fähigkeiten zu prüfen.

4. ausladende Klasse (Large Class)

Ist das Interface verworren so steht es meistens auch nicht besonders gut um die interne Struktur. Muß eine solche Klasse viele Datenelemente verwalten, sollte man die Aufteilung in mehrere Objekttypen erwägen. Dies geschieht, indem man erst die gewünschten Datenelemente, dann die entsprechenden Methoden in ein neu geschaffenes Objekt übersiedelt (II.3). Zur Verbesserung der Datenorganisation lohnt sich ein Blick auf die sich bietenden Strukturierungsmöglichkeiten (ii). Werden in einem Schritt viele neue Methoden und Datenelemente eingeführt, so kann ein Teil davon in eine Sub- oder Superklasse gezogen werden (II.3). Die interne Struktur sollte aber in der Objektorientierung niemals Selbstzweck sein, sondern sich stets nach den äußeren Gegebenheiten richten. Im prozeduralen Paradigma mag sich die Programmstruktur an der Berechnung orientieren, in der Objektorientierung hingegen zählen die Anforderungen mehr. Deswegen ist es keine schlechte Idee von der Verwendung auszugehen, sich an den von der Klasse zur Verfügung gestellten und den wirklich intensiv verwendeten Interfaces zu orientieren, diese gegebenenfalls anzupassen (V.2) und erst dann über das weitere Vorgehen bei der internen Restrukturierung zu entscheiden. Manchmal kann es sogar sinnvoll sein eine Klasse so aufzuspalten, daß Datenelemente dupliziert werden müssen und, was aber einen zusätzlichen Synchronisationsaufwand mit sich bringt (III.4).

5. ausufernde Parameterliste (Long Parameter List)

Lange Parameterlisten sind unangenehm in der Anwendung und verschlechtern die Lesbarkeit. Bevor der Verwender einen Aufruf machen kann, muß er sich der Aufgabe aller Parameter bewußt werden und überlegen, welche Werte übergeben werden sollen, auch dann wenn er nur etwas ganz einfaches bezwecken will. Für den Leser stellt sich die Aktualparameterliste als langes Tupel mit vielen Werten vor, deren Bedeutung in keiner Weise augenscheinlich ist.

Überlange Paramterlisten sind oft ein Zeichen eines mißglückten prozeduralen Designs. In Objekten hingegen haben Methoden Zugriff auf eine Reihe von Eigenschaften und Instanzvariablen. Steht ein benötigter Datenwert nicht direkt lokal zur Verfügung, so kann immer noch ein anderes Objekt darum gefragt werden.

Handelt es sich bei den Parametern um eine ganze Menge elementarer Datenwerte, so kann als Alternative die Übergabe eines ganzen Objektes statt vieler Einzelwerte angestrebt werden (IV.6: Objektprameter). Daneben besteht auch die Möglichkeit, die Methode in ihr eigenes Objekt zu übersiedeln (II.4) und vor dem eigentlichen Aufruf die benötigte Information mit Nachrichten an das neu geschaffene Objekt einzustellen. Eine andere Möglichkeit ist es, dieselbe Routine mit verschiedenen Signaturen oder mit optionalen Parametern (IV.2) zu deklarieren. Will die Umgestaltung der Parameterliste nicht so recht gelingen, sollte man sich vergegenwärtigen, ob nicht auch andere Designmängel wie Feature Envy (Fremdkoppelung) vorliegen und die Methode möglicherweise nicht in einer anderen Klasse besser aufgehoben wäre.

6. mehrgleisige Änderung (Divergent Change)

Äußerst unangenehm ist es für eine gewünschte Verhaltensänderung des Programms, mehrere Methoden umschreiben zu müssen, da man nur allzuleicht eine der vielen Anpassungen vergißt und sich das Programm daraufhin inkonsistent und merkwürdig zu verhalten beginnt. Ein offensichtlicher Mangel der zu mehrgleisigen oder verstreuten Änderungen führt, ist doppelter Code. Leider ist die Sache nicht immer so einfach. Eine

vom Nutzer oder Entwickler gewünschte Umstellung hat oft viele Facetten. Sind alle Änderungen innerhalb einer Klasse durchzuführen, so spricht man von mehrgleisiger Änderung (i.6); tangieren sie mehrere Klassen so handelt es sich um eine verstreute Änderung (i.7). Eine Möglichkeit mehrgleisige Änderungen in den Griff zu bekommen ist die Aufteilung der Verantwortlichkeit auf mehrere Klassen wie anhand des Model-View Konzepts später noch dargelegt.

7. verstreute Änderung (Shotgun Surgery)

Ist das Pendant zur mehrgleisigen Änderung. In vielen Fällen ist es schlimmer mehrere Klassen, welche über das ganze Programm verstreut sein könnnen, ändern zu müssen als viele Methoden in nur einer einzigen Klasse umzustellen.

***Strukturierung***

1. befreundete Datenobjekte (Data Clumps)

Gelegentlich finden sich mehrere Datenobjekte in einer Klasse, welche immer wieder zusammen verwendet werden. Solche Gruppen von Instanzvariablen können in ihrem eigenen Objekt beheimatet werden.

2. fehlende oder objektlose Feinstruktur (Primitive Obsession)

Offensichtlich zusammengehörige Information wie der Anfang und das Ende einer Zeitspanne liegen oft als getrennte Datenelemente vor. Man sollte sich nicht scheuen eigene Klassen für diese Aufgaben anzulegen.

3. Datensatzklasse (Data Class)

Klassen, die nur Datenelemente sowie entsprechende set- und get-Methoden in sich bergen könnten sind wie Records prozedurale Konstrukte. Es lohnt sich fast immer dieser Klasse mehr Verantwortung aus jenen Programmteilen, welche faktischen Gebrauch von der Datensatzklasse machen, zu übertragen.

4. unnütze Klasse (Lazy Class)

Klassen, welche im Laufe der Zeit durch das Übersiedeln von Eigenschaften und Fähigkeiten ihre Funktionalität fast vollständig eingebüßt haben, können wieder entfernt werden.

5. Spaghettilogik

Eine unübersichtliche Ablauflogik kann die Lesbarkeit des Programmes stark beeinträchtigen. (XXX Refactorings-Liste?)

***Vererbung***

1. Fremdkopplung (Feature Envy)

Die Grundidee der Objektorientierung ist es zusammengehörige Daten und Routinen gemeinsam zu kapseln. Einige Routinen benötigen aber Daten aus verschiedensten Objekten. Ist eine Methode mehr an einem Parameterobjekt interessiert als an ihrem eigenen, ist es Zeit sie zu übersiedeln.

2. wiederkehrende Case-Anweisung (Switch Statement) //auch ProcVariablen

Eine Case oder Switch Kontrollstrukturen sind im Grunde nichts schlechtes, verbessern sie doch gegenüber vielen einzelnen IF-THEN-Anweisungen die Übersicht im Programm. Kommt jedoch ein Datenobjekt bei seiner Verwendung immer nur im Kopf von Case-Kontrollanweisungen vor, sollte man sich überlegen, ob es sich nicht um ein Verhalten handelt, welches besser im Datenobjekt selbst zu kapseln ist. Dies trifft im besonderen auf Variablen, welche Typinformationen tragen, zu. In der prozeduralen Programmierung ist auch die Übergabe eines Prozedurparamters möglich.

3. ungenutztes Erbe (Refused Bequest)

Für den Nutznießer einer Klasse kann es aufreibend sein, wenn diese eine Unmenge an Fähigkeiten erbt, welche dann in Zusammenhang mit ihrem schlußendlichen Verwendungszweck einfach nicht mehr zu gebrauchen sind. Soetwas kommt manchmal vor, wenn zugunsten der Einfachheit in der Vererbungshierarchie entschieden wird, ist aber eine ungute Sache. Sprachen wie C++ bieten die Möglichkeit später nicht mehr genutzte Fähigkeiten zu verstecken(V.2), wer sich aber mit oberflächlicher Kosmetik nicht zufriedengeben will, der sollte in der Superklasse aufräumen (V.4).

4. analoge Subklassen (Parallel Inheritance Hierarchies)

Leiten zwei verwandte Klassen gleiche oder analoge Subklassen ab, so kann etwas in der Hierarchie nicht stimmen. Verwenden die einander entsprechenden Subklassen einen Satz gemeinsamer Funktionen, so hat man noch Glück im Unglück, weisen sie jedoch doppelten Code auf, so muß man erst damit beginnen Struktur in das vorherrschende Durcheinander zu bringen. Ein besserer Designansatz als zweimal parallel aber an unterschiedlichen Orten dieselbe Hierarchie aufzuziehen ist es, die gemeinsame Superklasse in Funktionsbereiche aufzugliedern und diese in eingeschachtelten Klassen mit eigener Vererbungshierarchie unterzubringen, wodurch das Problem auch ohne Multiple-Inheritance lösbar wird. Ein Beispiel dazu habe ich bereits in Kapitel 2 XXX gebracht. Dort wäre es ungeschickt gewesen die Hierarchien der artikelbezogenen Preisgestaltung und des entlehnbaren Artikels selbst durcheinander zu bringen.

5. ungenutzte Möglichkeit (Gegenmittel: Extract Hierarchy)

Findet sich eine komplexe hoch parameterisierte Klasse mit einer Vielfalt in der Funktionalität, die einem schweizer Taschenmesser um nichts nachsteht, so kann die Bildung von Subklassen für einzelne Teilaufgaben mehr Klarheit ins Design bringen, weil die einzelnen abgeleiteten Klassen nur ihren eigenen Spezialfall behandeln müssen und daher weniger abstrakt sind.

***Koppelung und Schnittstellen***

1. unerwünschte Seiteneffekte

Die Veränderung globaler Variablen kann unerwartete Effekte mit sich bringen. Wo möglich sollten Funktionen, welche einen Wert zurückgeben, ganz auf Seiteneffekte, also der Veränderung von Zuständen ihres Objekts oder eines anderen darüberliegenden Gültigkeitsbereiches, verzichten. Funktionen, welche beim Auftreten eines Fehlers eine Statusnummer zurückgeben und die Details der Fehlermeldung in globale Variablen schreiben, sollten stattdessen lieber eine Exception erzeugen. Sollen mehrere Werte zurückgeliefert werden, so kann dies entweder durch mehrere Variablenparamter oder die Rückgabe eines ganzen Objektes als Funktionswert erfolgen. Die Tatsache, daß Funktionen einen Wert zurückgeben, suggeriert dem Benutzer nämlich, daß die neu errechnete Information ausschließlich durch den Funktionswert nach außen gelangt; Seiteneffekte sind daher bei Funktionen zu entfernen. Die Aufgabe vieler Prozeduren und Methoden kann aber mittels direktem Zugriff sehr effizient gelöst werden (z.B. Einstellen der Zeichenfarbe für Leinwand, Flag ob Text seit letztem Speichern geändert worden ist). Seiteneffekte über globalen Variablen sind im Gegensatz zu den das eigene Objekt betreffenden Änderungen sehr sehr kritisch zu beurteilen, weil diese kein zweites mal instanziert werden können, was aber schnell, beispielsweise durch die Einführung von Threads in der parallelen Programmierung, zum Problem wird.

Jene Instanzvariablen, die von einem Seiteneffekten betroffen sind, müssen als private oder protected deklariert werden, sodaß sie von außen nicht unkontrolliert manipuliert werden können. Die Möglichkeit, daß eine Routine mit Seiteneffekt auf Variable x diese ein zweites mal als Variablenparameter überreicht bekommt, muß in jedem Fall vermieden werden, weil dies zu Inkonsistenzen führen kann.

Wenn sie richtig gehandhabt werden, können Seiteneffekte Parameterlisten entlasten und die Effizienz steigern.

var y:=1, r:=0;

procedure(var x) = begin x:=3; r:=y+x; end;

lokale Behandlung; x ist als Zeiger realisiert: x = 3, r = 6

RemoteProcedureCall; x wird hin und zurück kopiert: x = 3, r = 4

*Übergebener Var-Param wird geändert; schlägt Änderung sofort zu y durch?*

2. umständliche Nachrichtenkette (Message Chains)

Gibt eine Methode ein Objekt zurück an welches eine weitere Nachricht gesendet wird, so liegt eine Nachrichtenkette vor. Nachrichtenketten können sehr lange werden. Beispielsweise kann die Terminreservierung beim Chef an die Sekretärin oder ein entsprechendes Kalenderobjekt delegiert werden. Der Vorgesetzte eines Mitarbeiters kann z.B. nur über die Abteilung, in der er arbeitet und welche eine Referenz auf den Abteilungsleiter bereitstellt, erreicht werden. Diese Details der Weiterreichung kann durch eigene Methoden im Ausgangsobjekt gekapselt werden. Eine überlange Nachrichtenkette

habe ich bereits im Bibliotheks-Beipiel vorgestellt: ausleihe.buch.gebührenverrechnung.EntlehnGebühr(ausleihe.dauerinTagen). Diese konnte gemäß (IV.1) vereinfacht werden, indem im Entlehnungsobjekt eine eigene Methode zur Kostenermittlung eingeführt worden ist, sodaß ausleihe.Gebühr den gewünschte Effekt erbringt.

3. rigide Delegation (Middle Man)

Ein Zuviel an Delegation kann das Interface einer Klasse künstlich aufblasen. Jede Klasse sollte einen Mindestanteil der an sie gerichteten Nachrichten in Eigenverantwortung lösen können. Der Overhead zur Kapselung von Nachrichtenketten kann in Abhängigkeit von der verwendeten Sprache erheblich sein und legt damit implizit den Mindestanteil zur Auflösung fest.

4. stark gekoppelte Klassen (Inappropriate Intimacy)

Klassen welche stark miteinander kommunizieren müssen, sollten in irgendeiner Form zusammengebracht werden. Die Vielzahl an Kommunikationsbeziehungen löst nämlich die Kapselung und damit die Möglichkeit eine Klasse unabhängig von der anderen verändern zu können auf und verursacht gleichzeitig auch noch einen Overhead, welcher sich negativ auf die Ausführungszeit auswirkt.

5. Teilfunktionalität aus Bibliothek übernehmen (Incomplete Library Class)

Die Funktionalität einer aus einer Bibliothek übernommenen Klasse kann durch Ableitung einer Unterklasse beliebig erweitert werden. Polymorphe (d.h. virtuelle) Methoden aus der Superklasse welche Objekte der Oberklasse zurückgeben erfordern allerdings einen expliziten Downcast, wenn bekannt ist, daß sie jetzt mit dem Objekt einer Subklasse arbeiten. Beispielsweise könnte ein Canvas-Objekt, welches als Zeichenfläche in einem Fenster fungiert, um neue Zeichenoperationen wie der rotierten Ausgabe von Ellipsensektoren erweitert werden.

***Kontext und Metainformationen***

1. unangemessener Sichtbarkeitsbereich

Der Sichtbarkeitsbereich einer Methode oder Variablen sollte nicht größer als notwendig sein. Es ist möglich alle Verwendungen einer Fähigkeit oder Eigenschaft aufzusuchen und ihr den dabei ermittelten minimalen Sichtbarkeitsbereich zuzuweisen. Der Gültigkeitsbereich einer Zählervariable in einer Schleife sollte nicht über diese hinausreichen. Engere Gültigkeitsbereiche vermindern die Koppelung zwischen einzelnen Programmteilen, was die Wartbarkeit und Erweiterbarkeit verbessert. Letztendlich muß aber anhand logischer Gesichtspunkte entschieden werden wo eine Einheit hingehört.

2. temporär genutzte Felder (Temporary Field)

Die Aufgabe temporär genutzter Instanzvariablen ist meistens nur schwer zu ergründen. Oft wurden sie eingeführt, um Spezialfälle in einem Algorithmus zu behandeln. Wenn dies möglich ist, sollte man eigene Klasse zur Abdeckung der Sonderfälle anlegen. Um die fortwährende Abfrage, ob ein Objekt instanziert worden ist und einen gültigen Wert enthält, vermeiden zu können, kann ein eigenes Nil- oder Guardobjekt eingeführt werden. Nachrichten an Nilobjekte verpuffen gewöhnlich ohne eigens abzufangender Fehlermeldung im Bitnirwana; Guardobjekte kennzeichnen zumeist das Ende einer Liste und bereinigen das Suchen, Einfügen und Entfernen von Listenelementen um lästige Spezialfälle.

3. nicht genutzte Flexibilität (Speculative Generality)

Ein zuviel an Flexibilität ist nicht optimal; sie erschwert die Lesbarkeit und verlangsamt das Programm. Mittels Refactoring, vor allem dann wenn eine ausreichende Unterstützung durch Werkzeuge gegeben ist, kann man Indirektionen und andere Mittel zur Steigerung der Flexibilität dynamisch einführen oder entfernen, so diese benötigt werden.

4. übereifrige Kommentierung (Comments & Direktwerte)

Kommentare sollten wichtige Informationen für den Leser enthalten, welche Auskunft darüber geben, was gemacht werden soll. Ist ein Code ohne ausführliche Kommentierung nicht lesbar, so drängt sich der Verdacht schlechter Strukurierung auf. Im Idealfall ist der Quellcode selbsterklärend. Die Namen von Konstanten anstelle von Direktwerten sowie die Benennung von Unterroutinen sollten bereits einen groben Eindruck von dem vermitteln, was hier vorgeht.

5. nicht mnemonische oder inkonsistente Namensgebung (misnomer, Alternative Classes with Different Interfaces)

Unzutreffend oder gar willkürlich gewählte Bezeichner erschweren das Verständnis eines Programmes, wenn sie es nicht sogar unmöglich machen. Alle Bezeichnungen sollten sorgsam gewählt sein und den tatsächlichen Zweck oder die tatsächlichen Bedeutung des Objekts oder der Aufgabe, welche sie benennen, wiedergeben. Mnemonische Bezeichner sind einprägsam und entsprechen ihrer wahren Bedeutung. Im Ernstfall darf man die Umbenennung nicht scheuen. Methoden in unterschiedlichen Klassen, welche aber dieselbe Operation durchführen, sollten gleich benannt sein, auch dann wenn im Kontext der jeweiligen Klasse eine andere Benennung naheliegender wäre.

Ich hoffe in diesem Kapitel einen Einblick in die Vielgestaltigkeit der Designschwächen und Verbesserungsmöglichkeiten gegeben zu haben. Obwohl man der Liste wahrscheinlich noch einiges hinzufügen könnte, geben bereits die ersten sechs Aromen einen guten Ausgangspunkt, um über die weitere Vorgehensweise entscheiden zu können. Doppelten Code sowie lange und überladene Methoden, Klassen und Parameterlisten kann man sogar oft schon durch einen kurzen Blick auf den Quellcode ohne genaueres Nachlesen erkennen. Die unter ii-v aufgeführten Strukturierungsgesichtspunkte sind großteils nur Spezialfälle oder stehen in direktem oder indirektem Zusammenhang mit den zu Beginn erwähnten

Designmerkmalen. Bestimmte Ungereimtheiten wie verstreute oder mehrgleisige Änderungen fallen hingegen unter Umständen erst beim Durchführen einer Änderung oder dem Anstehen einer Erweiterung auf.

## *Umgestaltungsmöglichkeiten*

Einige der besprochenen Designänderungen bedeuten große Umwälzungen für das Gesamtsystem. Um so wichtiger ist es den gesicherten Weg kleiner Schritte nicht zu verlassen. Es gibt recht verschiedene Ansätze zur Einteilung elementarer Refakturierungen. Die folgende Aufstellung ist eine von mir gewählte Einteilung, welche in einigen nicht ganz unwesentlichen Punkten von der in [XXX] abweicht. Beispielsweise habe ich einige Änderungen wie Pull Up Field, Pull Up Method, Push Down Field, Push Down Method und Move Method/Field zusammen mit der Schachtelung von Routinen im Punkt Instanzvariablen und Methoden übersiedeln zusammengefaßt, welcher eigentlich nur zwei aufeinanderfolgende Änderungen des Gültigkeitsbereiches vorsieht. Im Gegensatz zu den im letzten Kapitel besprochenen Designmerkmalen richten sich die hier besprochenen Umwandlungen nicht nach Zweck oder Semantik sondern alleine nach der technischen Durchführung. Andere Restrukturierungen wie die Umwandlung von Rekursion in Iteration und vice versa habe ich selbst hinzugeführt, um eine ausschließlich puristisch objektorientierte Sichtweise zu vermeiden und auch anderen Programmentwicklungsparadigmen wie dem funktionalen oder blockstrukturiertem Rechnung zu tragen. Damit es leichter ist, sich unter den einzelnen Punkten etwas vorzustellen, habe ich teilweise Kommentare in Klammern eingefügt. Zu jeder Umgestaltung auch Beispiele zu bringen würde jedoch den Rahmen dieser Arbeit sprengen, weshalb ich den interessierten Leser auf Martin Fowlers Buch verweisen will.

Grundlegendes

6. Methoden extrahieren oder einsetzen
7. Temporärvariable ersetzen (parallele Zuweisung nur Lisp)
8. Zuweisung an Parameter entfernen
9. Umbenennen, Temporärvariablen separieren, unbenutzte Variablen entfernen (Kosmetik)
10. Rekursion versus Iteration
11. Algorithmus substituieren
12. Ablauflogik vereinfachen
13. Nil/Guard-Objekt einführen

Kontextwechsel

14. Zugriffsbereiche schaffen/auflösen (With-Anwsg)
15. Gültigkeitsbereich erweitern,einschränken&wechseln
16. Instanzvariablen und Methoden übersiedeln
17. langes Unterprogramm einkapseln (Replace Method with Method Object)
18. Klassen und Objektkontext
19. Aggregieren/Einebnen (geschachtelte Klassen)
20. Sichtbarkeit ändern (private, public, protected, published)

Datenorganisation

21. Datensätze und Klassen extrahieren oder einsetzen
22. statische versus dynamische Datenobjekte (Change Value/Reference)

23. von Feldern, Listen, Vektoren und Mengen
24. einfache und doppelte Verkettung
25. von Aufzählungen, Objekttypen und Klassenreferenzen(virt. Konstruktoren, vol.Typ.)
26. eigenverantwortliche Allokation (Factory Method)

Parameterübergabe

27. Parameter hinzufügen/entfernen
28. Übergabeart festlegen
29. Seiteneffekte eliminieren
30. Funktionalitäten trennen (Farbe setzen und abfragen: für FN kontraintuitiv)
31. Methoden trennen/parameterisieren (Prozedurvariablen)
32. Objektparameter (Preserve Whole Object, Introduce Parameter Object)

Kapselung

33. Eigenschaften (Introduce/Remove Setting Method)
34. Schnittstelle anpassen (Hide Method, private&public, …)
35. Sichtbarkeit geerbter Eigenschaften ( :private classxy, Superklasse anlegen)
36. spezialisierende Typumwandlung kapseln (Encapsulate Downcast)
37. bestehende Klasse konsistent erweitern

Sprachelemente und Entwurfsmuster

38. Nachrichtenketten, Delegation und Vermittler
39. Delegation und Vererbung
40. Ausnahmen, Flags, explizites Testen
41. volatile Typisierung (Typänderung zur Laufzeit)
42. Entwurfsmuster einführen/wechseln (Design Patterns: MVC, Iterator, …)

## *Refakturierung in der Praxis*

Will man sein Programm nun tatsächlich reorganisieren, so sind noch einige Dinge zu beachten, die ich bisher nicht in ausreichendem Maße behandelt habe:

Wann soll refakturiert werden? – wenn Änderungen anstehen, welche nur schlecht in das bestehende System einzugliedern sind, wenn die Übersicht im Programm wiederhergestellt werden soll, wenn bei einem Review Verbesserungsvorschläge auftauchen um Fehler leichter aufzuspüren. Gründe gibt es genug. Vielleicht sollte man sich aber auch dessen bewußt sein, daß der Nutzen einer Restrukturierung nicht sofort in Erscheinung tritt, weshalb es sinnvoll sein kann direkt vor einem Meilenstein oder Abgabetermin vorübergehend darauf verzichten. Grundsätzlich ist es aber besser laufend zu reorganisieren, denn zu warten bis es anders nicht mehr geht verursacht nur unnotwendige Mehrarbeit.

Die Entscheidung für oder gegen ein Refactoring hängt aber nicht zuletzt vom damit verbundenen Aufwand ab. Deshalb ist die Unterstützung durch Werkzeuge essentiell. Die ersten dieser Werkzeuge waren für Smalltalk verfügbar, welches auch gleichzeitig eine umfangreiche Bibliothek und Fundgrube für Programmteile bot. Wer sich für den Einsatz eines bestehendes Rectoring-Tools entscheiden will, sollte auf die Integration in die Entwicklungsumgebung, die Benutzerfreundlichkeit sowie die unterstützten Sprachen und die Menge der verfügbaren Transformationen achten:

vollautomatische Tools

- Smalltalk VisualWorks (Smalltalk)
- Eclipse Platform (Java, C, C++, Perl); Open Source,
- Together ControlCenter v6
- IntelliJ IDEA (Java); kommerziell,

  auch komplexe Rf.: Nachrichtenkette<->Delegation, Konstruktor<->Factory-Methode
- RefactorIT (Java); komerziell, integrierbar in Jbuilder und NetBeans
- Borland JBuilder v7: eigentlich Entwicklungsumgebung für Java; unterstützt aber gleichzeitig elementare Refactorings wie die Änderung der Paketstruktur schon in v.6

halbautomatische Tools

- Refactoring Browser
- XRefactory
- jFactor
- Delphi: eigentlich nur erweiterte Entwicklungsumgebung für Object Pascal:

  Find in Files, automatisches Umbenennen, Hierarachy-Browser

Obgleich es eine Vielzahl an Tools gibt, möchte ich hier für keines Partei ergreifen sondern den Leser seine Entscheidung lieber selbst treffen lassen. Ob ein Werkzeug den Bedürfnissen entspricht, kann man am besten selbst feststellen, indem man es ausprobiert. Leider werden trotz der augenscheinlichen Vielzahl an Werkzeugen meistens nur elementare Refactorings und immer nur einige wenige Sprachen unterstützt.

Will man ein eigenes Tool schreiben, so ist es von großem Vorteil bereits über ein funktionierendes Front-End eines Compilers der Quell- und Zielsprache zu verfügen. Ist dies nicht der Fall, kann man auf sogenannte Scanner- oder Parsergeneratoren wie Flex, Yacc oder CoCo zurückgreifen, welche Parser und Scanner automatisch aus einer Grammatik generieren, wobei auf die Art der unterstützten Grammatiken (LL1, SLR, LALR, LR1, …) zu achten ist.

So gut man auch mit Tools ausgestattet ist und so gut man sich auch eingearbeitet hat, alle Restrukturierungen wird man nicht vollautomatisch durchführen können. Gute Entwicklungsumgebungen bieten auch hier eine Unterstützung durch erweiterte Such- und Ersetzungsfunktionen oder einer graphische Übersicht über die bestehende Klassenhierarchie an. Schlimmstenfalls erfolgt die Programmerstellung mit dem Systemeditor, was aber auch keine so große Katastrophe ist, wie es vielleicht scheinen mag. In diesem Fall sollte der Programmierer sein Vorhaben in möglichst kleine Schritte unterteilen und nach jeder elementaren Umformung gründlich testen. Es ist von Vorteil, wenn man seine Tests automatisieren kann. Das Testen während einer Restrukturierung eröffnet glücklicherweise gegenüber dem Testen neu erstellter Software einige Vorzüge. So ist es nicht nur möglich mithilfe von Assertions die Übereinstimmung zwischen Implementierung und Konsistenzbedingungen der Spezifikation zu überprüfen sondern direkt die Ausgabe des reorganisierten Programmes mit der ursprünglichen Ausgabe zu vergleichen. Wenn dies

möglich ist, ist es sehr effizient die Programmausgabe in eine Datei umzuleiten und von einem Programm auf Gleichheit prüfen zu lassen:

```
MeinProg  2>&1 1>ausgabe2.txt
FC ausgabe1.txt ausgabe2.txt
```

Unter Linux ist statt dem Befehl FileCompare diff zu verwenden. Händisches Refactoring im besonderen aber auch etwa halbautomatisches erforderten auf jeden Fall die wiederholte Übersetzung des Quellcodes. Softwareinterpretierte Sprachen bieten hier den unschätzbaren Vorteil, daß ein Programm erst gar nicht übersetzt werden muß, sondern direkt interpretiert werden kann, weshalb auch in Smalltalk schon so früh eine Refakturierung möglich war. Es können aber auch Compiler einiges bereitstellen um die Übersetzung zu beschleunigen: Reine Syntaxprüfung ohne Erzeugung von Objektcode, modularisierte Compilation, um nur die zuletzt geänderten Programmteile neu übersetzen zu müssen, sowie das Abschalten nicht benötigter aber zeitraubender Optimierungen. Besonders schnell und effizient sind Sprachen wie (Object) Pascal, die in nur einem Durchlauf übersetzt werden können. Java hingegen erlaubt zyklische Klassendefinitionen ohne Forward-Deklarationen, was mehrere Durchläufe erfordert und compiliert meiner eigenen Erfahrung nach nur sehr langsam, obwohl der Objektcode nur auf einer virtuellen Maschine ausführbar sein muß. Vielleicht sollte man der Übersetzungsgeschwindigkeit keinen zu großen Stellenwert einräumen, ist das Warten doch mehr ein subjektives Ärgernis als eine faktische Verzögerung und werden doch die Rechner immer schneller.

Abschließend möchte ich die Restrukturierung noch unter dem Gesichtspunkt der Motivation erörtern. Eine zu rigide Planung schlägt sich vor allem in der schlechten Motivation der Mitarbeiter nieder. Ist die Stimmung im Keller so geht jeder nur mehr den Weg des geringsten Wiederstandes, sodaß man nicht lange zu suchen braucht bis man auf Erscheinungen wie doppelten Code oder ähnliches trifft. Quelltext mittels Copy und Paste zu kopieren und abzuändern ist nämlich viel einfacher als sich über die Programmstruktur Gedanken zu machen.

Gelingt es hingegen den Entwicklern mehr Verantwortung zu übertragen, wirkt sich dies normalerweise auch positiv auf die Motivation aus. Treffen die Programmierer kleinere Designentscheidungen selbst, gibt ihnen dies das Gefühl sich selbst besser einbringen zu können. Dies ist am meisten in einem Team, welches aus lauter gleichwertigen Mitgliedern besteht, der Fall. Voraussetzung hierfür sind allerdings gut qualifizierte Programmentwickler.

Auch in Bereichen wie dem Software Reuse verbessert Refactoring nicht nur die technische Durchführbarkeit sondern erlaubt es auch dem Wiederverwerter den übernommenen Code nach seinem eigenen Geschmack umzugestalten, was ein nicht ganz unwesentliches Hemmnis für den Reuse beseitigt. Als negativ kann es jedoch empfunden werden, wenn nach getaner Arbeit regelmäßig jemand über den eigenen Quellcode herfällt und ihn derart zerzaust, daß ihn der Urheber selbst nicht wiedererkennen würde. Werden aufgrund einer zu konformistischen Gesinnungen all die guten Dinge, die sich der Entwickler überlegt hat, wieder ausgebaut, so ärgert dies natürlich. Refactoring dient grundsätzlich dazu Mängel im Design zu beheben, aber nicht dazu dem Quelltext anderer gewaltsam seinen eigenen Stempel aufzudrücken. Ist eine Designentscheidung nicht unmittelbar nachvollziehbar, so sollte im Zweifelsfall der Urheber gefragt werden.

## *Zusammenfassung und Ausblick*

Refactoring bringt mehr Flexibilität in den Softwareentwicklungsprozeß, indem es zusätzlich zur strikten Vorausplanung des TopDown Modells auch Elemente der BottomUp-Entwicklung zuläßt. Der Einsatz leichtgewichtiger Softwareentwicklungsprozesse gewinnt durch die Möglichkeit der Refakturierung an Attraktivität, erfordert jedoch gute Programmierer. Umgekehrt können die Ursprünge des Refactorings im Extreme Programming, in der Open-Source Bewegung und der objektorientierten Programmierung (Smalltalk) gesucht werden.

Restrukturierungen werden meistens vorgenommen um das Design bestehender Software zu verbessern, um die Entwicklung flexibler zu gestalten und um die Entwicklungsgeschwindigkeit vom Anfang bis zum Ende leichter durchhalten zu können. Dies ist sehr schwer, denn die im Laufe der Zeit zunehmende Komplexität der Software erschwert und verlangsamt die Arbeit ganz drastisch. Refactoring hat den Problemen in Erhaltung und Wartung als auch bei der Entwicklung einiges entgegenzusetzen, muß aber umsichtig eingesetzt werden. Die Möglichkeit später Refactorings durchführen zu können darf nicht dazu verleiten einfach darauf los zu programmieren und sich daraufhin durch den Entwicklungsprozeß zu hacken. Eine stabile Architektur und ein umsichtiges Design stellen nach wie vor die Grundlage für den Erfolg oder Mißerfolg eines Projektes. Das Refactoring ermöglicht es uns das Design eines Programms nicht mehr als starre Salzsäule, die irgendwann zerbröselt, sondern als dynamische (wenn auch manchmal etwas zähflüssige) Eigenschaft von Software zu sehen.

Refactoring ist auch ein interessante Forschungsdisziplin, welche vom Software-Engineering über die Graphentheorie bis hin zum Compilerbau viele unterschiedliche Gebiete in sich vereint. Hier finden sich auch einige interessante Ansätze wie Designmetriken oder die Nutzung sprachunabhängiger Modelle wie etwa OMG´s Model Driven Architecture, welche vielleicht in Zukunft eine größere Rolle spielen könnten. Während Compiler die Mikrostruktur eines Programms zur Leistungssteigerung transformieren betreffen die hier besprochenen Design-Refactorings mehr die Architektur und Gesamtstruktur einer Anwendung. Design Patterns(Entwurfsmuster) können dynamisch eingeführt und aufgelöst werden. Ich habe Änderungen am Design stets unter dem Gesichtspunkt der Lesbarkeit, Änderungsfreudigkeit und Erweiterbarkeit betrachtet, es steht jedoch außer Zweifel, daß auch die Architektur einen wesentlichen Einfluß auf die Leistung hat.

## *Literaturangaben:*

Die originalen Literaturangaben sind leider verloren gegangen. Ich bitte um Umsicht und Entschuldigung:

Folgende drei Bücher/Artikel habe ich meinen Zuhörern nach gehaltenem Vortrag empfohlen:

*1. Fowler, Martin; Refactoring: improving the Design of existing Code* **** wichtigste Quelle ****

*2. Serge Demeyer, Bart Du Bois, Hans Stenten, Piter Van Gorp; Refactoring: Current Research and Future Trends*

*3. Eric Steven Raymond:The Cathedral and the Bazaar*

*Die verwendete/verfügbare Literatur habe ich im selben Ordner, wie die Seminararbeit aufbewahrt und es können meines Wissens nach sicher keine weiteren Quellen verwendet worden sein:*


1. Gregor J. Rothfuss, Master Thesis: A Framework for Open Source Projects; 2002

2. Eric Steven Raymond: The Cathedral and the Bazaar; Thyrsus Enterprises *, 2000

3. Tom Mens: Refactoring: Current Research and Future Trends, Vrije Universiteit Brussel, Belgium, Universiteit Antwerpen, Belgium, LDTA´03 Preliminary Version

4. Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, Oscar Niertrasz: A Meta-model for Language-Independent Refactoring, Proceedings ISPSE 2000, IEEE, 2000, pp. 157-167

5. Prof. Jacques Pasquier-Rocha: An Introduction to Refactoring, Master Seminar, University of Fribourg, Switzerland ⋆

6. Nicola Fankhauser: The composite structure of tests in JUnit, April 17, 2003

7. Yoshio Kataoka, Michael D. Ernst, William G. Griswold, David Notkin: Automated Support for Program Refactoring using Invariants, University of Washington, of California, MIT Lab for Computer Science

8. Yoshio Higo, Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue, Yoshio Kataoka: On Refactoring for Open Source Java Program, Osaka University, PRESTO Corp, Japan, Toshiba Corp., Japan

9. Huiqing Li, Claus Reinke, Simon Thompson: Tool Support for Refactoring Functional Programs, University of Kent °

10. René Meier, Marc-Olivier Kilijian, Raymond Cunningham, Vinny Cahill: Towards Proximity Group Communication, Trinity College Dublin

11. Jan Philipps, Bernhard Rumpe: Roots Refactoring, University München

12. Arie van Deursen, Leon Moonen: The Video Store Revisited – Thoughts on Refactoring and Testing, CWI, the Netherlands

13. Ralf Reißing: Refactoring, Universität Stuttgart

14. Michael G. Wagner, J. Randall Randy Jue: The Open Source Myth; Why Executives Should Think Twice about Making Company Source Code Public

15. Yung-Pin Cheng: Refactoring Design Models for Inductive Verification, Taiwan


* http://www.tuxedo.org/~esr/, esr@thyrsus.com
⋆ http://diuf.unifr.ch/softeng/, nicola.frankhauser@unifr.ch
° http://www.cs.kent.ac.uk/projects/refactor-fp/