Seminar „Software in Komponenten"´

Sommersemester 2005

Delocalized Plans

Maximilian Schmidt
schmidtm@inf.fu-berlin.de
Advisor: Christopher Oezbek

14.08.2005

Zusammenfassung

"Delocalized Plans" - Modelle, Strategien, Methoden und Werkzeuge

Inhaltsverzeichnis

1 Einleitung

Es ist erwiesen, dass bei der Entwicklung von Software durchschnittlich mehr als 50% des Gesamtaufwandes für die spätere Wartung einkalkuliert werden müssen [Tie89, FM92, PPBH91]. Dieser Aufwand liegt u.a. in dem Problem von ”delocalized plans” begründet.

Das Modifizieren von Software lässt sich in 3 Phasen einteilen: Das Verstehen, das Modifizieren an sich, sowie das Validieren der Änderungen. Um eine Änderung vorzunehmen, muss der zu modifizierende Teil vorerst gefunden werden. Schon diese Aufgabe kann sich als schwierig erweisen, wenn eine entsprechende Funktion über mehrere Programmteile verteilt ist [RCM04]. Diese Streuung der Funktionalität erschwert aber nicht nur das Finden bestimmter Programmfunktionen, sondern ist auch eine reiche Fehlerquelle bei der Modifikation eines Programmes.

Ziel der vorliegenden Ausarbeitung ist es, das Problem der ”delocalized plans” durch Beispiele aus vorhandenen Texten zu belegen und die Anstrengungen, die dadurch entstehenden Fehler zu vermeiden, zusammenzufassen. Dazu werde ich im ersten Teil in die Terminologie einführen, näher auf die unterschiedlichen Problemarten eingehen und einige Beispiele zur Verdeutlichung aufzeigen. Im Hauptteil stelle ich dann Methoden vor, welche sich zur Aufgabe gesetzt haben, diese Probleme näher zu analysieren und/oder Lösungswege aufzuzeigen. Abschliessend werde ich vergleichend zur Lektüre die aktuelle Situation aufgreifen und eine Liste von frei verfügbaren Hilsmitteln aufstellen.

„Changes made to a program without an understanding of its behavior can ruin the integrity of the system." - „Änderungen an einem Programm vorzunehmen, ohne ein Verständnis für dessen Verhalten zu besitzen, kann die Integrität des umgebenden Systems zerstören." [PPBH91]

2 Grundlagen

Im Folgenden werde ich zuerst in das begriffliche Umfeld einführen und dann schließlich einige Beispiele nach Soloway/Letovsky [SPL+88, SL86] anführen, welche das Vorkommen von "delocalized plans" näher beschreiben und die resultierenden Probleme verdeutlichen sollen.

2.1 Modelle & Strategien

Der Begriff "delocalized plan" beschreibt eine allgemeine Funktion ("feature") eines Programms, deren zugehörige Teile physikalisch voneinander getrennt implementiert wurden [Tie89]. Man differenziert zwischen "plan" und "goal", wobei ein "plan" einer umfassenden funktionalen Einheit entspricht und "goal" nur das Resultat mehrerer "plans" zusammenfasst. Man könnte z.B. das Sortieren eines Datensatzes als "goal" bezeichnen, welches durch einen "Mergesort"-Algorithmus realisiert wird, welcher vier abstrakte Teilschritte ("plans") beinhaltet: Die Rekursion auf einem binären Baum, den Vorgang, eine Folge von Zahlen in zwei Teile zu spalten, zwei Zahlen aufsteigend zu sortieren und schließlich das Zusammensetzen von sortierten Teilfolgen [PPBH91]. Die grundlegene Funktionsweise eines Plans ist also, einen Teil des übergreifenden Ziels ("goal") in mehrere kleine Einheiten, sog. "subgoals" zu unterteilen, welche wiederum bestimmte Teile des eigentlichen Programmcodes repräsentieren [Tie89].

Um ein Programm zu verstehen, gibt es mehrere Herangehensweisen. Generell unterscheidet man zwischen "micro-strategies" und "macro strategies". "Macro strategies" bezeichnen generelle Herangehensweisen an ein vorhandenes Programm, wogegen "micro strategies" die Herangehensweisen im Kleinen beschreiben, d.h. wie ein gestelltes Problem innerhalb eines speziellen Programmkontextes verarbeitet wird. "Micro strategies" können sich z.B. dadurch unterscheiden, ob für die Modifikation eines bestehenden Programms ein vorher aufgestellter Plan systematisch verfolgt wird oder ob der Ansatz eher frei erfolgt und nur an der aktuellen Situation innerhalb des Quelltextes orientiert ist [RCM04].

Die Kategorie der in diesem Kontext entscheidenden "macro strategies" unterteilt sich in zwei spezielle Herangehensweisen: "systematic" und "as needed". Im ersten Fall, des systematischen Herangehens, wird das Programm in der Abfolge seiner Operationen komplett durchgesehen, wodurch ein globales Verständnis der Zusammenhänge entsteht. Gerade bei größeren Programmen scheint diese Herangehensweise jedoch sehr ineffizient zu sein. Tests bestätigen, dass das "Lesen" eines kompletten Programmes ("browsing"/"scrolling") extrem zeitaufwenig ist und mit steigender Komplexität des

Programms sogar zu Fehlern führen kann - im Gegensatz zu einer direkten Herangehensweise, z.B. durch Referenzensuche "cross reference searches" und Suche nach Schlüsselwörtern "keyword searches" [RCM04].

Daher wird meist die "as needed" Strategie verwendet, bei welcher direkt an dem gestellten Problem angesetzt wird und nur die daran geknüpften Verweise betrachtet werden. Dadurch kann es wiederum zu Fehlern kommen, da kein übergreifendes Verständnis vorhanden ist.
Soloway impliziert in dieser Hinsicht bei seiner Einteilung eines Programms in seine funktionalen Bestandteile durch "plans" eine "as needed" Methode des Verstehens, welche hier auch als "bottom-up"-Methode verstanden werden kann. Durch das Lesen eines Programmstückes können besagte "plans" herausgearbeitet werden, welche zusammen zu einem höheren "goal" zusammengesetzt werden können [PPBH91]. Erfahrene Programmierer haben ein gewisses Repertoire an "plans", welches ihnen beim Verständnis eines unbekannten Programmes helfen kann [Tie89].

2.2 Experimente

Soloway/Letovsky führten mehrere Versuchsreihen durch, welche allgemeine Verständnisfehler bei der Quelltextanalyse zu Tage fördern sollten. Die Versuche wurden an einem mittelgroßen Programm vorgenommen: der "personal database" (PDB) - geschrieben in Fortran. Die PDB beschreibt ein Programm zur Speicherung von personenbezogenen Daten, wobei verschiedene Operationen ausgeführt werden können, um z.B. einen Datensatz zu löschen, zu suchen oder neu einzufügen. Die Aufgabe bestand darin, den vorhandenen Quelltext um verschiedene Methoden zu erweitern.
Nachfolgend das erste von zwei Beispielen:

Die Aufgabe bestand darin eine Methode RESTORE zu schreiben, welche den letzten Löschvorgang durch die Operation DELETE rückgängig machen kann. Als sich die Versuchspersonen zu diesem Zweck die Operation DELETE ansahen, wurden einige von ihnen bei folgender Zeile stutzig:

```
NCHNGE = NCHNGE + 1
```

In der Dokumentation konnte dazu gelesen werden:

```
NCHNGE : contains the number of changes to the database made
         during the current session
```

Jetzt stellte sich die Frage ob die Wiederherstellung eines Eintrages eine Änderung darstellte. Alle Versuchspersonen schienen diese Frage für sich mit "ja" zu beantworten, denn die Zeile wurde in allen Fällen für die neue Methode RESTORE übernommen. Fakt war jedoch, dass diese Variable einen Zähler darstellte, mit welchem gemessen wurde, ob am Ende einer Sitzung die Datenbank neu geschrieben werden sollte. Da die Methode RESTORE im eigentlichen Sinne ihrer Anwendung jedoch eine Änderung

an der Datenbank (nämlich DELETE) rückgängig machen sollte (und somit der Zähler dekrementiert werden müsste), wurde in der modifizierten Datenbank bei sämtlichen Versuchspersonen der Datenbestand unnötig oft neu geschrieben.

Als zweites Beispiel lässt sich im selben Kontext ein Programmstück anführen, welches die Datenbank vor einem Speicherüberlauf bewahren sollte, indem eine obere Grenze MAXREC eingeführt wurde. Bei jedem neu eingefügten Eintrag sollte der aktuelle Wert von Einträgen NRECS erhöht werden, durfte jedoch nie MAXRECS erreichen. Hier die Codefragmente zum Einfügen ein neuen Eintrags, sowie zum Einlesen der Datenbank in den Speicher beim Start des Programms:

```
    SUBROUTINE create(...)
C  Create new record in Database
     IF (NRECS.EQ.MAXREC) THEN
      WRITE(6,*) "Database full."
      WRITE(6,*) "Cannot perform command."
      WRITE(6,*) "Contact system manager."
     RETURN
C  Else, really create record
      NRECS = NRECS + 1;
      ....
      RETURN
    END

    SUBROUTINE getDB(...)
C  Read the database into the array
     ...
10   READ(13,20,END = 100) DBASE(IREC)
     IREC = IREC + 1;
     IF (IREC.LT.MAXREC) GOTO 10
      WRITE(6,*) "Database full."
      WRITE(6,*) "Contact system manager."
      IERR = 1
     RETURN
      ....
     RETURN
    END
```

Da bei jedem Erzeugen eines neuen Eintrags überprüft wird, ob die Datenbank schon voll ist, können nie mehr als MAXREC Einträge in der Datenbank vorhanden sein und die Überprüfung beim Einlesen der Datenbank kann als redundant angesehen werden.

Auch wenn die Fehler dieser beiden Beispiele vorerst nur zu unschönen Nebeneffekten wie Performanceverlust oder Redundanz führen, so sind doch in komplexeren Systemen weitreichendere und kritische Fehler damit wortwörtlich vorprogrammiert.

Aus diesen Resultaten konnte geschlussfolgert werden, dass gewisse Referenzen in der Dokumentation eines Programms enthalten sein sollten, welche beschreiben, zu welchem Zweck ein Konstrukt vorhanden ist und welche Rolle es einnimmt, sowie, falls notwendig, welche anderen Teile von einer betrachteten Struktur abhängig sind [SPL+88, Tie89].
Zum ersten Beispiel könnte eine korrekte Dokumentation der Variablen NCHNGE demnach ungefähr so lauten:

```
NCHANGE:
 Role: contains the number of changes to the database so far
 Goal: used to test whether database file needs to be updated
       at the end of a session
```

Um im zweiten Beispiel das Konzept bzgl. des Speicherüberlaufs deutlich zu machen könnte man das entsprechende ”goal” zusammen mit einem oder mehreren ”plans” in die Dokumentation einfügen:

```
Goal: Prevent Overflow of Array
Plan: Whenever NREC might be increased, guard against overflow
 Places where NREC might be increased:
   (1) when reading in db in GETDB
   (2) when allocating new records in CREATE
```

Wie man also sieht ist das Vorhandensein von semantischen Beschreibungen der Referenzen notwendig, um festzustellen, WARUM etwas verwendet wurde, da sich nur das WIE direkt aus dem Programmtext ableiten lässt [Tie89].

3 Methoden

Es wurden im letzten Abschnitt eine kurze Einführung in die fachliche Terminologie gegeben und zur Verdeutlichung der Problematik einige Beispiele angeführt. Im weiteren Verlauf sollen nun drei Methoden gezeigt werden, welche einen Ansatz bieten, diese Problematik ansatzweise zu umgehen oder zumindest unterstützend einzugreifen.

3.1 Program slicing

Wie Soloway/Letovsky in ihren Experimenten bereits feststellten, ist die Modifikation eines bestehendes Programms gerade deswegen so fehlerfördernd, da durch dezentralisierte Strukturen bestimmte Teile des Codes missverstanden werden können oder gar unbeachtet bleiben. Das Prinzip des ”program slicing” bezeichnet in diesem Fall die Vorgehensweise, alle Teile eines Programms, welche sich auf einen gewissen Teil beziehen, zusammenfassend und ohne jegliche Redundanz zu extrahieren.

Auf diese Weise sollte es möglich sein, alle Folgen einer Änderung überblicken zu können. Diese Herangehensweise kann man sich sehr leicht an einem einfachen Programm verdeutlichen, in dem man eine globale Variable betrachtet und jegliche Zugriffe und Funktionen, die diese Variable verwenden extrahiert, ohne jedoch die Teile einzubeziehen, welche nicht darauf bezogen sind [GL91, HRB90]. Abbildung 1 zeigt "program slicing" auf einer Variablen **i** in einem simplen Beispiel (Wisconsin Program-Slicing Tool). Man sieht in den markierten Bereichen alle Programmstücke, welche mit dieser Variablen direkt oder indirekt verknüpft sind. In Zeiten von OOP (Objekt-orientierte Programmierung) ist dieses Verfahren ungleich schwieriger geworden und es müssen dynamische Verfahren eingesetzt werden um z.B. den Lebenszyklus eines Objekts zur Laufzeit feststellen zu können [LN97].

```
/* Return the sum of parameters a and b */
static int add(a,b)
int a,b;
{
  Count++;      /* count number of calls */
  return (a+b);
}

static void demo()
{
  int sum, i;

  /* Initialize */
  sum = 0;
  i = 0;

  /* Cumulative sum */
  while (i < MAX) {
    sum = add(sum,i);

#ifdef EVEN
    i = add(i,2);  /* Next even integer */
#else
    i = add(i,1);  /* Next integer */
#endif EVEN
  }

  /* Result */
  printf("Sum %d, i %d, Add %d\n",
          sum, i, Count);
}

main() {
  demo();
}
```

Abbildung 1: program slicing

3.2 Literate Programming

Unter "literate programming" versteht man das Verfassen eines Quelltextes unter Integration der dazugehörigen Dokumentation auf eine Art und Weise, die ermöglicht, das Programm (ähnlich einem literarischen Text) zu "lesen", statt Dokumentation und Code separat zu betrachten. Zu diesem Zweck werden innerhalb des Programmtextes zwischen die Befehle, Zuweisungen und Definitionen einer Sprache verschiedene Schlüsselwörter eingefügt, welche eine beschreibende Rolle einnehmen und auf diese Weise direkt mit dem zugrundeliegenden Code verwoben sind. Soll das entsprechende Programm dann übersetzt werden, werden Dokumentation und Programmcode getrennt und einem entsprechenden Übersetzer bzw. einem Textsetzsystem übergeben.
Auf diese Weise ist es möglich, den semantischen Kontext eines Programms in den eigentlichen Programmcode einzuflechten und so Missverständnisse bereits im Keim zu ersticken. Diese Methode hat sich aufgrund der Umständlichkeit der Integration in existierende Programmiersprachen bisher nicht durchgesetzt. Es können jedoch Teile die-

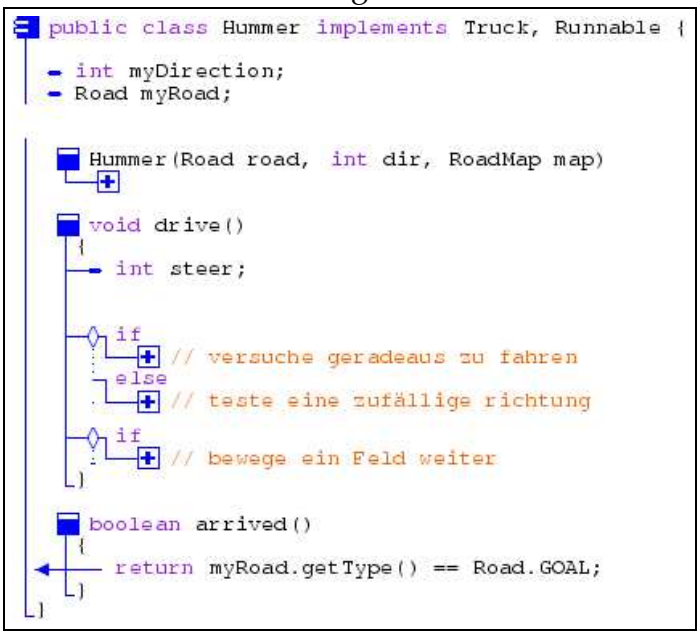
ser Idee in Sprachkonstrukten wie Java/JavaDoc wiedergefunden werden.

```
@2. procedures and functions of 8-queens@=code
   @2.1 procedure SetQueen@
   @2.2 procedure RemoveQueen@
   @2.3 procedure regress@

@2.1 procedure SetQueen@=code
  procedure SetQueen;
    begin
      EmptyRow[i]:=false;
      EmptyPositiveDiag[CurrentColumn+i]:=true;
      EmptyNegativeDiag[CurrentColumn-i]:=true;
    end;

@2.2 procedure ....
```

Abbildung 2: Das 8-Damen Problem [SC93]

3.3 CSDs

Abbildung 3: CSD



Um das Verständnis eines Programmes nicht unnötig zu verkomplizieren, können Techniken wie ”program slicing” eingesetzt werden, welche relevante Fragmente extrahieren. Seit es OOP gibt ist jedoch z.B. für das grundsätzliche Verständnis von Programmen ein ”top-down” Ansatz besser geeignet. Dabei werden vorerst die grundlegenden Funktionen (”goals”) des Programms betrachtet, um dann tiefer zu gehen und einzelne funktionale Teile (”plans”) zu betrachten. Um eine solche Herangehensweise zu unterstützen können Diagramme der Kontrollstrukturen ”control structure diagramms” (CSD) herangezogen werden, welche verschiedene Ebenen von Abstraktion darstellen können. Das

Prinzip eines CSDs besteht darin, den syntaktischen Aufbau eines Programms in einen Baum zu übertragen, so dass für das momentane Verständnis unwichtige Ebenen ausgeblendet werden können [HCM02]. CSDs sind heute rudimentär in üblichen Entwicklungsumgebungen integriert, indem es möglich ist unwichtige Programmteile durch "Faltung" auszublenden [CS88].

4 Tools

Wie leicht zu erkennen ist, sind die im letzten Abschnitt genannten Methoden heute durchaus nichts besonderes mehr, sondern teilweise in jedem besseren Quelltexteditor anzutreffen. Um jedoch alle Vorteile dieser Methoden nutzen zu können, muss man häufig zusätzliche Programme oder Module heranziehen. Anbei folgt nun eine kleine Liste der frei verfügbaren Plugins für Eclipse, einer kostenlosen IDE, welche hauptsächlich in und für Java entwickelt wurde, jedoch auch Unterstützung für Python und C/C++ bietet.

- Code Analysis Plugin (CAP)
  CAP is a plugin for the eclipse platform and analysis the dependencies of your Java project. It opens an own perspective and displays the results in an clear way using different diagrams.
  http://cap.xore.de

- Creole
  With Creole you can explore your Java code visually allowing you to see its structure and the links (calls, accesses, etc.) between its different pieces.
  http://www.thechiselgroup.org/creole

- JLinxs
  JLinxs provides an additional view to quickly search through packages, classes, methods, and fields. In the additional view, JLinxs provides the possibility to access other types of documents (PDF, HTML) as quickly as the Javadoc files.
  http://www.diligent-it.com/products/

- Eclipse Code Folding
  The Coffee-Bytes Eclipse Folding Plug-in enhances the default Eclipse code folding feature set with many additional options and plenty of new functionality.
  http://www.coffee-bytes.com/

- Protocols
  The Protocols plug-in for the Eclipse Java development environment lets you group related elements of class and interface definitions. Partitioning the source code into sections, using comments as headers, is a common way of improving code readability ...
  http://www.bergner.se/protocols

- RefactorIT
  Restructure Java source-code by its over 30 Refactoring Operations and Wizards. RefactorIt helps you to understand the project by Smart Code Searches and Graphical Dependency Analyzer.
  http://www.refactorit.com/

- JGrasp (standalone)
  jGRASP produces Control Structure Diagrams (CSDs) dynamically for Java, C, C++, Objective-C, Ada, and VHDL; CPG diagrams for Java and Ada; UML diagrams for Java; and has an integrated debugger and workbench for Java.
  http://www.jgrasp.org/

5 Zusammenfassung

Nachdem am Anfang dieser Ausarbeitung in das begriffliche Umfeld eingeführt wurde und einige Beispiele zur Verdeutlichung zitiert wurden, konnten im zweiten Teil einige Methoden genannt werden, welche helfen sollten, die Folgen von "delocalized plans" zu minimieren. Wie im vorigen Abschnitt gesehen werden konnte, sind die meisten Funktionen heute selbstverständlich für den Programmierer in Entwicklungsumgebungen ansatzweise verfügbar, wie z.B CSDs, im Sinne des Ausblendens von unwichtigen Quelltextabschnitten, sowie literate Programming im Sinne von Lokalität von Code und Dokumentation in Inventionen wie JavaDoc. Das Prinzip des Slicing wird für OOP (dynamisches Slicing) noch erforscht, es gibt jedoch viele einzelne Programme, welche dafür gute Ansätze liefern.

Schließlich ist es Dank OOP heutzutage durch Abstraktion möglich, Programme strukturiert direkt von einem konkreten Modell abzuleiten, so dass auch hier das Suchen von Funktionen innerhalb eines Programms erleichtert wird. Dadurch erhält auch die (integrierte) Dokumentation eine hierarchische Struktur, was ebenfalls entscheidend ist, damit gewisse Details verborgen werden können, sobald eine abstraktere Sichtweise auf das gegebene System notwendig ist[FM92, MV93].

Zusammenfassend lässt sich schlussfolgern, dass ein ansatzweise optimales System noch nicht entwickelt wurde. Neben der Dokumentation sollte der Programmierer optimalerweise in der Lage sein, gleichzeitig den Programmcode zu betrachten, während er sich eine Liste von Referenzen für ein gerade gewähltes Objekt ansehen kann [FM92]. Allein das Umschalten zwischen verschiedenen Perspektiven (z.B. Views in Eclipse) ist noch zu umständlich, jedoch unvermeidlich, da der visualisierte Bereich auf die Grösse des Bildschirms beschränkt ist.

Literatur

[CS88] J.H. Cross and S.V. Sheppard. The control structure diagram: an automated graphical representation for software. *System Sciences, Proceedings of*

the Twenty-First Annual Hawaii International Conference on, 2:446–454, January 1988.

[FM92] Robert M. Freeman and Malcolm Munro. Redocumentation for the maintenance of software. *Proceedings of the 30th annual Southeast regional conference*, April 1992.

[GL91] K.B. Gallagher and J.R. Lyle. Using program slicing in software maintenance. *Software Engineering, IEEE Transactions on*, 17(8):751–761, August 1991.

[HCM02] D. Hendrix, J.H. Cross, and S. Maghsoodloom. The effectiveness of control structure diagrams in source code comprehension activities. *Software Engineering, IEEE Transactions on*, 28(5):463–477, May 2002.

[HRB90] Susan Horowitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):36–46, January 1990.

[LN97] Danny B. Lange and Yuichi Nakamura. Object-oriented program tracing and visualization. *IEEE Computer*, 30(5):63–70, May 1997.

[MV93] A. Mayrhauser and A.M. Vans. From program comprehension to tool requirements for an industrial environment. *Program Comprehension, IEEE Second Workshop on*, pages 78–86, July 1993.

[PPBH91] S. Paul, A. Prakash, E. Buss, and J. Henshaw. Design technologies: Theories and techniques of program understanding. *Proceedings of the conference of the Centre for Advanced Studies on Collaborative research*, October 1991.

[RCM04] M.P. Robillard, W. Coelho, and G.C. Murphy. How effective developers investigate source code: an exploratory study. *Software Engineering, IEEE Transactions on*, 30(12), December 2004.

[SC93] Stephen Shum and Curtis Cook. Aops: An abstraction oriented programming system for literate programming. *Software Engineering. Journal*, 8(3):113–120, May 1993.

[SL86] Elliot Soloway and Stan Letovsky. Delocalized plans and program comprehension. *IEEE Software*, 3(3):41–48, May 1986.

[SPL+88] Elliot Soloway, Jeannine Pinto, Stan Letovsky, David Littman, and Robin Lampert. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31(11):1259–1267, November 1988.

[Tie89] Tim Tiemens. Cognitive models of program comprehension. *Software Engineering Research Center*, December 1989.