REPORT DOCUMENTATION PAGE

1. AGENCY USE ONLY (Leave blank)

2. REPORT DATE
10/23/98

3. REPORT TYPE AND DATES COVERED
Final Progress Report 8/94-8/98

4. TITLE AND SUBTITLE
Identifying objects in Legacy Systems for Reuse and Reengineering

5. FUNDING NUMBERS
DAAH04-94-G-0334

6. AUTHOR(S)
Arun Lakhotia

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(ES)
The Center for Advanced Computer Studies
The University of Southwestern Louisiana
PO Box 44330
Lafayette, LA 70504

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AR033634.12-RT-DP5

11. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy or decision, unless so designated by other documentation.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This project investigated the problem of migrating legacy software systems into object-oriented systems. We have successfully developed a technique for automatically refactoring legacy programs to make them object-oriented, without changing their external behavior. The technique consists of three parts. First, all non-recursive functions of a program are inlined to create one big function. This function is then broken into smaller set of functions using certain rules of cohesion. The new set of functions are partitioned using cluster analysis such that each set of function represents a set of methods in a class. Our approach offers significant improvement over previous approaches. Since the program is factored into a new set of functions, our approach identifies objects even in poorly written programs, where other approaches fail. We are now experimenting with automatically identifying poorly written functions so that we can perform selective inlining. Our results provide an important milestone in automatic approaches for overhauling legacy software systems.

14. SUBJECT TERMS
Software Reengineering; Legacy Systems; object-oriented design; Software Maintenance

15. NUMBER OF PAGES
7 (seven)

16. PRICE CODE

17. SECURITY CLASSIFICATION OR REPORT
UNCLASSIFIED

18. SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

19. SECURITY CLASSIFICATION OF ABSTRACT
UNCLASSIFIED

20. LIMITATION OF ABSTRACT
UL

NSN 7540-01-280-5500

Enclosure 1
Identifying Objects in Legacy Systems for Reuse and Reengineering

Final Progress Report

Arun Lakhotia

October 23, 1998

U.S. ARMY RESEARCH OFFICE

Grant number: DAAH04–94–G-0334

UNIVERSITY OF SOUTHWESTERN LOUISIANA

APPROVED FOR PUBLIC RELEASE; UNLIMITED DISTRIBUTION

THE VIEWS, OPINIONS, AND/OR FINDINGS CONTAINED IN THIS REPORT ARE THOSE OF THE AUTHOR(S) AND SHOULD BE CONSTRUED AS AN OFFICIAL DEPARTMENT OF ARMY POSITION, POLICY, OR DECISION, UNLESS SO DESIGNATED BY OTHER DOCUMENTATION.
EXECUTIVE SUMMARY

This project investigated the problem of migrating legacy software systems into object-oriented systems. We have successfully developed a technique for automatically refactoring legacy programs to make them object-oriented, without changing their external behavior. The technique consists of three parts. First, all non-recursive functions of a program are inlined to create one big function. This function is then broken into smaller set of functions using certain rules of cohesion. The new set of functions are partitioned using cluster analysis such that each set of function represents a set of methods in a class. Our approach offers significant improvement over previous approaches. Since the program is factored into a new set of functions, our approach identifies objects even in poorly written programs, where other approaches fail. We are now experimenting with automatically identifying poorly written functions so that we can perform selective inlining. Our results provide an important milestone in automatic approaches for overhauling legacy software systems.

PROBLEM STUDIED

This project investigated the problem of migrating legacy software systems into object-oriented systems. The goal of the project was to develop techniques for identifying potential objects in legacy software systems.

SUMMARY OF IMPORTANT RESULTS

We have successfully developed an approach for identifying objects—a collection of functions and global variables—in legacy software systems, such as those written in FORTRAN and earlier languages. Our approach is significantly different from that we had proposed in the original proposal. The pitfalls of our original approach and our new approach have been reported in various annual technical reports. They are summarized below.

In addition, the last section summarizes some results from working on problems identified in the course of the primary research direction. These are considered as "orthogonal" in that they have implications independent of the central problem being studied.

Pitfalls of the original approach. In our original proposal we had separated the problem of reengineering legacy code into two steps:

1. Partition the functions (procedures, modules) of the system such that each partition represents the methods of an object class.
2. Reorganize the code such that its file/directory organization reflects these partitions. (Thus each file defines a single object class.)

Our project proposal addressed the first step: to recover objects in legacy code. We were motivated by the assumption that this was the most critical step when reengineering a software system. Based on that same assumption, other researchers working independently on the problem of reengineering legacy systems have also focused their attention on the subproblem of recovering objects [Lakhotia, 1997].

We performed a prototype study of our approach by restricting the problem to the simpler problem of finding "enumerated types" in programs using symbolic constants without any group...
[Lakhotia & Gravley, 1995; Gravley & Lakhotia, 1996]. The enumerated types identified by
our method were evaluated by comparing against enumerated types identified by a programmer.
Our results were not very satisfactory. The enumerated types created by the various heuristics
we used had too many positive and negative failures.

Besides recovering objects by cluster analysis, we also experimented with identifying objects
by pattern recognition. While we developed a “declarative” method for encoding and recognizing
program patterns [Bhatnagar, 1996], we soon ran into the limitations faced by other knowledge
based techniques. Our repository of program patterns was too small to be useful. And the effort
required to scale the repository was too monumental to pursue within the scope of this effort.
Also at question was the effectiveness of the pattern recognition based approach. In our initial
experiments the approach failed to recognize instances that were slight deviation of the original
pattern. Hence, we discontinued that effort too.

The results of other researchers investigating methods for recovering objects, parallel to
our work, have also produced negative results [Lindig and Snelting, 1997]. (A survey of these
efforts may be found in Lakhotia, 1997.) None of the techniques investigated for partitioning
functions (or similar syntactic units) into object classes have produced what may be considered
good object classes.

The negative results—on hindsight—can be explained by a basic principle of computing:
garbage in garbage out. If the initial factoring of a legacy system into functions is convoluted
one cannot just reorganize its components into a clean object-oriented model.

**Refactoring legacy code.** Our original approach for object recovery failed because the prim-
itive building blocks—typically, functions and procedures—of a legacy system are usually not
well factored. Legacy systems, as a result of repeated modifications over their life-span, contain
functions and procedures that are large and perform several different, orthogonal computations.
Such functions cannot be placed in any single object class because they modify various types of
data. As a result, object recovery techniques that simply group such functions fail.

The fundamental problem in making a system object-oriented, then, is to develop methods
to refactor the functions and procedures of the system. For instance, a function that performs
several disparate computations may be refactored by decomposing it into several functions,
each performing a single computation. Sometimes, refactoring may involve combining the
computations of two functions into a single function.

We have developed a set of formal transformations—Wedge, Split, Fold, and Tuck—for
refactoring large functions by decomposing them into small functions [Lakhotia and Deprez,
1998; Lakhotia, 1998]. Starting with some seed statements—an initial set of statements identified
by the programmer—the transformations first create a wedge that consists of all the statements
that affect the computations at the seed statements within a single-entry, single-exit region. The
flow graph of the function is then split such that all the statements in the wedge are placed
contiguously. This contiguous piece of code, which also forms a single-entry, single-exit region,
is then folded into a function. The whole process of wedge, split, and fold is called tuck.

**Identifying objects in legacy code.** In an earlier work we had developed an algorithm to
identify functions that were non-cohesive. This algorithm computes the cohesion between “pairs
of variables" by using the data and control dependences [Lakhotia, 1993; Nandigam, 1995; Nandigam, Lakhotia, & Cech, 1998]. The cohesion between variables can be used to partition the set of variables. Each group in the partition is an indicator of code that can be moved to a separate function.

The tuck transformation along with our algorithms for computing cohesion gives the necessary tools for refactoring a system. We have developed a prototype system that refactors a single function into several small functions. The results from our initial experiments are very promising. We are able to accurately identify functions that are not cohesive and to decompose such functions into smaller, meaningful functions.

Our approach for identifying objects may be described as follows:

1. Inline all the non-recursive functions of the program to create one big function.
2. Refactor this big function into smaller, cohesive functions [Deprez, 1997; Lakhotia & Deprez, 1997]
3. Use cluster analysis to create groups of these functions, such that each group represents methods of a class [Lakhotia, 1997].

We have experimented with Steps 2 and 3 separately. We are now developing a prototype that combines all the steps together.

Orthogonal research results As is common in most research efforts, besides focussing on our primary research agenda, we also produced some results on problems identified in the course of the research. These problems may be summarized into three groups.

1. Precise slicing of unstructured programs
2. A precision model for dataflow analysis algorithms
3. Debugging failures discovered by large data sets

An overview of our results for these problems follows.

Precise slicing of unstructured programs A slice of a program P at statement s is the set of statements that might affect the behavior of the program observed at s [Tip 1995, Weiser, 1984]. Our refactoring transformation performs a specialized form of program slicing. We have developed a slicing algorithm ideally suited for legacy programs. Our algorithm creates precise slices for any procedural programs, which includes programs containing any type of goto statements [Lakhotia & Deprez, 1997]. Previous algorithms compensated for the existence of goto statements by creating less precise slices [Agrawal, 1994; Ball and Horwitz, 1993; Choi and Ferrante, 1994]. Thus, our present algorithm significantly improves upon the algorithms proposed by others.

Program flow analysis We have developed a new theoretical model of a class of flow analysis problems [Lakhotia and Kortright, 1996a,b]. Our model generalizes a previous model of such problems [Kam and Ullman, 1977]. A significant advantage of our model is that it also provides a scale for measuring the precision of flow analysis algorithms. In the absence of such a scale, algorithms are compared by using benchmarks especially crafted to exhibit differences in the
result of algorithms. Our model also provides a basis for developing algorithms parameterized to improve their precision, albeit at an extra cost.

Flow analysis is used during object recovery to analyze dependencies between program components. In the absence of the dependency information the algorithms have to make conservative estimates—err on the side of assuming that certain components are related where in actuality they may not be. Thus, imprecision in such dependence analysis impacts the precision of object recovery algorithms, and hence the extent to which this task can be automated. An algorithm parameterized for cost-benefit trade-off would offer an engineer the opportunity to make similar trade-off decisions in distributing the tasks between man and machine.

**Debugging failures discovered by large data sets** Early in the project when experimenting with different cluster analysis techniques we encountered an interesting debugging problem. Our program failed on a very large data set derived from some real world program. We found it extremely hard—rather impossible—to use the failure causing data for debugging. The primary reason being that the data was very large and printing the intermediate state of the computation produced enormous amount of data.

This problem led to Chan’s Ph.D. research on debugging program failures discovered by large data sets [Chan, 94; Chan & Lakhotia, 98]. The essence of the work is a catalogue of techniques that help in creating smaller data sets that replicate the failure.

We are the first to identify this debugging problem. An analysis of a collection of debugging experiences surveyed by a group in Oxford indicates that such debugging situations, when they occur, can be “near fatal” for a software project. Our techniques, we hope will one day save some dying software project.

**LIST OF MANUSCRIPTS**


Grant # DAAH04-94-G-0334

**SCIENTIFIC PERSONNEL**

1. **Anurag Bhatnagar.** Received Ph.D. (Computer Science) in Spring 1996.
2. **Niranjan Bopardikar.** Expected to complete M.S. (Computer Science) in Spring 1998
3. **Varma Chanderraju.** Received M.S. (Computer Science) Spring 97.
4. **Al Ciallella.** Received M.S. (Computer Science) in Fall 96.
5. **Glen Davis.** Completed B.S. (Computer Science)
6. **Rodney DeSoto.** Enrolled in B.S. (Computer Science)
7. **Jean-Christophe Deprez.** Received M.S. (Computer Science) in Fall 1997. Is currently working towards a Ph.D.
8. **John M. Gravley.** Enrolled in Ph.D. program.
9. **Sharat Jenigiri.** Received M.S. (Computer Science), Spring 98.
10. **Shreyash S. Kame.** Expected to complete M.S. (Computer Science) in Spring 1999.
11. **Gaurav Khillan.** Received M.S. (Computer Science) Spring 97
12. **Enrique Kortright.** Associate Professor with Nicholls State University.
13. **Arun Lakhotia.** Principal Investigator.
14. **Durand E. LeBlanc.** Received B.S. (Computer Science)
15. **Daniel Liguas.** Moved on to other projects. Currently enrolled in Ph.D. (Computer Science).
16. **Pablo Mejia.** Did not complete B.S.
17. **Daryl Spillman.** Received M.S. (Computer Science)

**REPORT OF INVENTIONS**

1. Algorithm for classifying subsystems of a program
2. A measure of congruence between software architectures
3. A set of transformations for refactoring non-cohesive functions of a program.
4. Precise slicing of unstructured programs.
5. A precision model for data flow analysis frameworks
6. A relational approach to specifying and recognizing patterns
7. Algorithm for identifying enumerated types in legacy code
BIBLIOGRAPHY

Documents mentioned in the List of Manuscripts, above, are also part of the bibliography. They are not repeated here to reduce redundancy.


