USAFA-TR-94-3

RE-ENGINEERING LEGACY COBOL PROGRAMS

LT COL J. K. JOINER
DEPT OF COMP SCI
USAF ACADEMY
CO 80840

W.T. TSAI
DEPT OF COMP SCI
UNIV OF MINNESOTA
MPLS, MN 55455

DECEMBER 1994

FINAL REPORT

prepared for

DEAN OF THE FACULTY
UNITED STATES AIR FORCE ACADEMY
COLORADO 80840

USAFA-TR-94-3

Technical Review by Lt Col Schweitzer
Department of Computer Science
USAFA Academy, Colorado 80840

Technical Review by Lt Col R. K. Liefer
Department of Astronautics
USAF Academy, Colorado 80840

Editorial Review by Lt Col Kent A. Esbenshade
Department of English
USAF Academy, Colorado 80840

This research report entitled "Re-Engineering Legacy Cobol Programs" is presented as a competent treatment of the subject, worthy of publication. The United States Air Force Academy vouches for the quality of the research, without necessarily endorsing the opinions and conclusions of the author.

This report has been cleared for open publication and public release by the appropriate Office of Information in accordance with AFM 190-1, AFR 12-30, and AFR 80-3. This report may have unlimited distribution.

[illegible]

JAMES D. BEASON, Lt Col, USAF
Director of Research

17 Jan 95
Dated

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
1994

3. REPORT TYPE AND DATES COVERED
FINAL REPORT

4. TITLE AND SUBTITLE
RE-ENGINEERING LEGACY COBOL PROGRAMS

5. FUNDING NUMBERS

6. AUTHOR(S)
J.K. JOINER W.T. TSAI

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
DEPT OF COMP SCI
USAF ACADEMY, CO
80840

DEPT OF COMP SCI
UNIV OF MINNESOTA
MPLS, MN 55455

8. PERFORMING ORGANIZATION REPORT NUMBER
USAFA-TR-94-3

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

Accesion For

NTIS CRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	

By
Distribution /

Availability Codes

Dist	Avail and / or Special
A-/	

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This paper proposes a semi-automatic two-step process to re-engineer legacy Cobol programs into OO programs including automatic identification of instance variables and methods from Cobol code followed by semi-automatic optimization of the OO design produced in the first step. We describe several issues related to automatic feature identification and report our experiences using two different automatic approaches on sample industrial code. Then we describe four ways to improve an OO design including moving methods from one class to another, merging methods, splitting methods, and merging classes. This is done by analyzing class interaction diagrams, call graphs, and code and presenting potential optimization points to programmers.

14. SUBJECT TERMS
OBJECT-ORIENTED DESIGN, SOFTWARE MAINTENANCE

15. NUMBER OF PAGES
20

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
UNCLASSIFIED

18. SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

19. SECURITY CLASSIFICATION OF ABSTRACT
UNCLASSIFIED

20. LIMITATION OF ABSTRACT

Re-Engineering Legacy Cobol Programs

J. K. Joiner

Department of Computer Science
US Air Force Academy
USAFA, CO 80840

W. T. Tsai

Department of Computer Science
University of Minnesota
Minneapolis, MN 55455

Abstract—This paper proposes a semi-automatic two-step process to re-engineer legacy Cobol programs into OO programs including automatic identification of instance variables and methods from Cobol code followed by semi-automatic optimization of the OO design produced in the first step. We describe several issues related to automatic feature identification and report our experiences using two different automatic approaches on sample industrial code. Then we describe four ways to improve an OO design including moving methods from one class to another, merging methods, splitting methods, and merging classes. This is done by analyzing class interaction diagrams, call graphs, and code and presenting potential optimization points to programmers.

1. Introduction

Legacy software systems are programs that are critical to the operation of companies, but that were developed years ago using early programming languages such as Cobol and Fortran. These programs have been maintained for many years by hundreds of programmers, and while many changes have been made to the software, the supporting documentation may not be current. These factors contribute to the staggering cost of maintaining legacy systems. Consequently, there is an urgent need to find ways to make these programs more maintainable without disrupting the operation of the company [2, 15, 3, 10]. One approach the Cobol software industry is considering is re-engineering their legacy systems into OO programs. In our contact with the Cobol software industry, we found the push to convert existing Cobol programs into OO programs coming both from within and outside the companies. Several software managers indicated to us that they receive frequent requests from their customers to convert legacy Cobol programs into OO programs.

Object-oriented (OO) design and programming are enjoying considerable popularity these days and much is being written about the benefits of using OO techniques on software development processes, including benefits for software maintenance [7, 5, 13, 16, 14, 11, 9]. While new software development can proceed from the ground up using the new techniques, legacy systems that were developed using an old style are more difficult to incorporate the new software development ideas. Comparing the characteristics of Cobol programs and OO programs, we find a wide gap between the two. The lack of local variables in Cobol means that the granularity of encapsulation is the entire program. In OO programs, variables can be encapsulated at the method and class level. The inability to pass parameters to called procedures in Cobol limits their generality and makes the entire program the unit of modularity. In OO programs methods and procedures are fully general and the class is the unit of modularity. Closing the gap between Cobol programs and OO programs is manageable if smaller steps are taken, therefore we propose a two-stage process here similar to one we used to identify instance variables and methods in legacy Fortran programs [15].

The first phase is automatic and involves the identification of instance variables and methods from the existing Cobol program using static program analyses such as data flow analysis [1], control flow analysis [8] and variable classification [3, 10]. Our approach to finding instance variables and methods in Cobol programs begins by constructing a data-centered dependence model of the program [10]. This model stores *define* and *use* information for each variable and *call* and *called-by* information for each paragraph. We then visit each paragraph and determine whether it should become a method, and if so, what class it should be attached to. This approach interleaves method and class identification, since new classes are created on the fly as methods are identified. When a paragraph is chosen to be a method, it is associated with a class at that time. If the class does not already exist, it is created with instance variables from the variables used in the method. The output of this step is a complete OO design consisting of classes with instance variables and methods. Next, we analyze the OO design in an attempt to improve and optimize it. We have identified four strategies for conducting this optimization step: (1) merging methods that cooperate in idioms, (2) removing cycles from the class interaction diagram, (3) merging smaller classes, and (4) splitting large methods into smaller ones. Finally, the Cobol code can be translated into an OO program based on the OO design developed so far. The complete re-engineering process is summarized in Fig. 1.

High-level process steps	Low-level process steps
1. Identify instance variables and methods from existing variables and paragraphs and compose into an OO design	a) Use control flow analysis, data flow analysis and variable classification to partition existing paragraphs as methods, b) allocate existing variables as instance variables
2. Optimize the generated OO design	a) Merge cooperating methods, b) eliminate cycles in the class interaction diagram, c) merge small classes, d) split large methods
3. Translate existing code assets into an object oriented program (in C++ for example)	a) Compute parameter lists for methods, b) translate legacy variables into OO instance variables, c) translate legacy code into OO methods

Fig. 1 Summary of a Cobol re-engineering process

The most crucial and difficult aspect of the process is instance variable and method identification because of the desire to create a good OO design. Although we can optimize an OO design to a degree after identifying an initial set of classes, the original Cobol program structure greatly influences the quality of the generated OO design because we identify instance variables and methods from the existing Cobol program. For this reason, we first studied the characteristics of sample legacy programs from industry and report the results in Section 2. In Section 3, we discuss some important issues related to semi-automatic re-engineering of legacy systems. In Section 4, we discuss feature identification algorithms and present the results of experiments using two different identification algorithms on sample industrial Cobol programs. In Section 5, we present our work on optimizing OO designs derived from Cobol programs using our algorithms.

2. Characteristics of legacy Cobol programs

Despite sizes of up to millions of lines of code, most legacy Cobol programs we encountered from industry were reasonably well structured with mature data divisions and numerous small cohesive paragraphs. To quantify these characteristics, we randomly selected portions of huge industrial programs that were being maintained by hundreds of programmers at their maintenance sites and studied them. Our software comprises ten programs totaling 13,522 lines of code, 6,489 variables and 355 paragraphs. To understand the data structures used in legacy programs, we counted the number of variables (V) and the number of top-level[1] variables (T) in our sample programs. We then

[1]We define a top-level variable to be a 01- or 77-level data item declared in the data division.

computed the average number of fields per top-level item (V / T). The data showed that the programs define an average of 649 variables each, but these are organized in an average of just 29 top-level variables with 22 fields each on average. Because of the number and average size of the top-level variables encountered in our sample programs, we conclude that top-level variables are reasonable candidates for becoming instance variables.

To understand the code structures used in legacy programs, we counted the number of source lines (L) and the number of paragraphs (P). We then computed the lines per paragraph (L / P). We found on average each program contains 36 paragraphs of approximately 14 lines each. The small average size of existing paragraphs hints that many of them may become methods of classes without further splitting or modification.

Next we were interested in how variables were used in the programs. We found that 42% of the paragraphs defined exactly one variable and 85% of the paragraphs defined three or less variables. This statistic is one more sign that existing paragraphs make good candidates for becoming methods because of the small number of variables being defined in them.

Finally, we found few 'messy' constructs such as fall-throughs (execution in one paragraph 'falls-through' into the next without a return), ALTER sentences (self-modifying code) or GOTO sentences used in the programs. Standard high-level database interface languages, such as SQL, are often used enhancing portability with the re-engineered programs. Based on our study of legacy Cobol programs, we made the following conclusions regarding re-engineering legacy programs into OO programs:

1. Automatic re-engineering tools are essential due to the size of the Cobol programs needing to be re-engineered.
2. Individual Cobol programs can be re-engineered one-at-a-time since Cobol applications are constructed of many 'small' programs.
3. Existing record variables make good candidates for instance variables since many represent the state of entities in the application domain.
4. Existing paragraphs make good candidates for methods since they are generally small and define few variables.
5. Minimal 'hand' re-engineering is needed because the programs use few messy constructs such as GOTOS, ALTERS or fall-throughs.

3. Issues in semi-automatic re-engineering

The most interesting and crucial aspect of this Cobol re-engineering approach involves the problem of identifying instance variables and methods in the existing Cobol program since these will be used to construct classes in the OO design. A well-designed class should contain a coherent set of instance variables and methods that carry out the operations on the state of the instance variables, and the quality of the OO design greatly influences the understandability and maintainability of the entire system. Since we are interested in re-engineering legacy systems that are in continuous use in their domains, our approach is to identify instance variables and methods from the existing assets in the Cobol program and then compose them into an OO design. We identified the following issues related to our approach:

- How do we classify and study the algorithms that are used to identify reusable features in the existing program?
- How do we evaluate the OO design generated by a given algorithm?
- How do we assure the equivalence of the original Cobol program and the new OO program?

3.1 Classifying feature identification algorithms

The algorithms used to examine paragraphs and determine their suitability as methods can be classified according to the types of analyses used and the weights put on each of the analyses. Three types of static program analyses can be used in the algorithms: (1) data flow analysis that reports the variables a paragraph defines and uses, (2) control flow analysis that reports the static calling hierarchy that exists in the code, and (3) variable classification that reports the categories of the variables in the program. Data flow analysis can be used to identify and rank the variables referenced in a paragraph according to the number of places it is referenced. Control flow analysis can be used to provide an order of visiting paragraphs and to make sure that paragraphs called by one paragraph are all assigned to the same class as the calling paragraph. Finally, variable classification can be used to pick only 'important' variables, such as domain or linkage variables, as instance variables of classes and not pick program variables such as scalars[2] to be the basis of classes. These three types of program analyses can be combined in a number of ways to guide the identification of instance variables and methods. This approach gives a taxonomy

[2]Scalars are individual, single-valued variables such as numbers and strings. They are contrasted with records and arrays which have multiple fields in a structured layout.

of feature identification algorithms consisting of single analysis approaches, two analysis approaches and three analysis approaches.

3.2 Evaluating OO designs

Next, the OO design generated by the automatic algorithm must be evaluated. The primary goal of the identification process is to create well-behaved classes that are useful for application designers and easy to maintain. One evaluation criteria for OO designs is Demeter's Law [12]. It advocates: (1) creating classes that do not rely on the internal structure of any other class, and (2) creating classes that do not send messages to a large number of other classes. Provision (1) can be satisfied by ensuring that methods only access instance variables of their own class. This includes the stipulation that a class not access the instance variables of its superclasses in an inheritance hierarchy. The solution for provision (2) as suggested by Lieberherr is to allow methods to send messages to classes of other instance variables or parameters of the method only. Following Demeter's Law does not guarantee a good OO design, but not following it probably makes an OO program harder to maintain. Enforcing Demeter's Law is mainly up to the programmer—the visibility of instance variables is an option in some OO programming languages and restricting messages to parameters is also under programmer control. We can recommend these details in an OO design, but the final decisions are made at implementation time.

Another way to evaluate an OO design is to compare the number of messages required to accomplish typical processing tasks. We cannot count or predict the actual number of messages sent at runtime, but we can compare the relative number of messages required by two designs for the same task. Consider a method that updates a counter. It must get the original value of the counter, add the increment, and store the new value. If the counter is an instance variable in the same class as the method, no messages need to be sent, but if it is in another class, two messages are required: one to get the original value and one to set the new value. In this example, we can say that one design saves two messages over the other. We can extend this analysis over the entire OO design and come up with a net number of messages saved between two designs.

Next, we can evaluate an OO design in terms of the class interactions involved. To facilitate this analysis, we can construct a class interaction diagram [See Fig. 2] such that an arrow from one class to another indicates that the latter class requests services from the former. In other words, the arrow depicts the 'flow' of services from a 'source' class to a 'sink' class. Using class interaction diagrams, we can compare two OO designs and compute statistics on the number and types of interactions among classes. Although Demeter's Law says to keep the number of class interactions low, it does not recommend a

Class interaction diagram without cycles. Information flows are easy to see from top to bottom of the diagram.

Interaction diagram containing same number of classes and interactions, but with cycles. Hard to trace information flows.

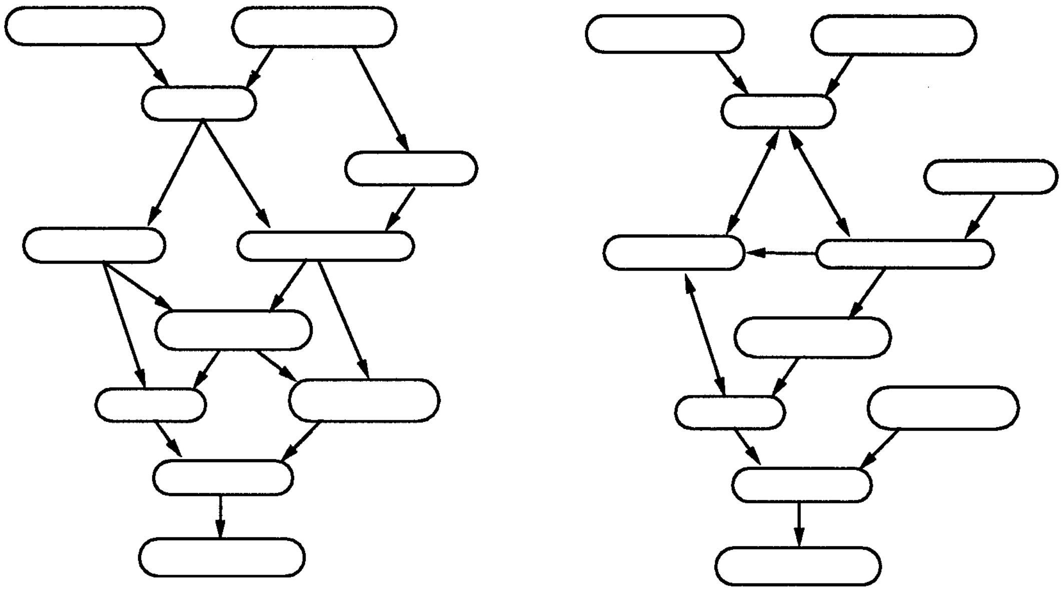


Fig. 2 Comparing class interaction diagrams and cycles

specific number. Recently, Chidamber and Kemerer [4] reported seeing median values of less than 10 interactions with other classes, plus they saw that 50% of a systems classes were self-contained and did not reference any other classes. We take these numbers as goals to achieve in an OO design.

Another statistic that can be derived from a class interaction diagram is the number of cycles among interacting classes. We believe that cycles in interaction diagrams complicate them and make understanding difficult, even though they are 'legal' in an OO design. Compare the class interaction diagrams in Fig. 2. They each contain the same number of classes and interactions, but the one on the left without cycles is easier to comprehend than the one on the right because the information flow and class dependence of the process depicted on the left is easier to understand.

Finally, OO designs can be evaluated based on the types of variables chosen for classes. Classes should be predominantly based on important variables from the application domain and methods should be chosen to provide a coherent set of required behaviors on the state of the instance variables. Our data-centered program understanding tools provide automatic variable classification into categories such as domain and program variables [3, 10]. The results of an automatic class identification algorithm can be compared with the automatic variable classification and conclusions drawn about the OO design. We believe that most domain variables should become classes while some program variables may become classes. Also, we need to be alert for domain variables that are missed during analysis of the Cobol program.

3.3 How to assure the equivalence of the original program and the re-engineered OO program

An extremely critical issue for organizations re-engineering legacy systems is how to assure that the re-engineered systems are equivalent to the systems they will be replacing. In the approach we propose for re-engineering legacy Cobol programs, we derive the OO design exclusively from the existing code. Due to this, we propose a systematic process of code inspections that compare the new OO code to the original Cobol code to assure the equivalence between the original and re-engineered programs. Our confidence in code inspection was recently confirmed by Davis, who claims inspection can find as many as 82 percent of all errors [6]. The inspection process we propose consists of three steps. The first step is to trace every variable in the original Cobol program to the OO program. Because we base our OO design on the existing variables, we can find every existing variable somewhere in the new program. Some will be instance variables in classes, some will be local variables in methods or procedures, and some will be global variables. Next we can trace every operation in the original Cobol program to the OO program. We implement the OO methods and procedures from the original Cobol code, so we can find every operation somewhere in the new OO code. Some operations will be in methods, some will be in procedures, some will be incorporated in new selection or iteration statements that have been hand optimized to take advantage of new programming language features. Finally, we can trace every procedure call from the original Cobol program to the OO program. Again, because we produce the OO implementation directly from the Cobol implementation, we can find every transfer of control in the new system. Some will be implemented as messages, some will be implemented as procedure calls, and some will be implemented as inline code in methods or procedures. One way to check this is to compare the original Cobol call graph with the new OO message sequence graph. These two graphs should match indicating that all of the transfers of control are accounted for. By verifying these three features through inspection, we can be reasonably sure that the functionality of the original Cobol program is present in the new OO code. A prudent final step would involve running the test plan developed for the original code on the new one, but as Davis points out, code inspections find a significant number of errors, greatly reducing the cost of testing.

4. Automatic feature identification approaches

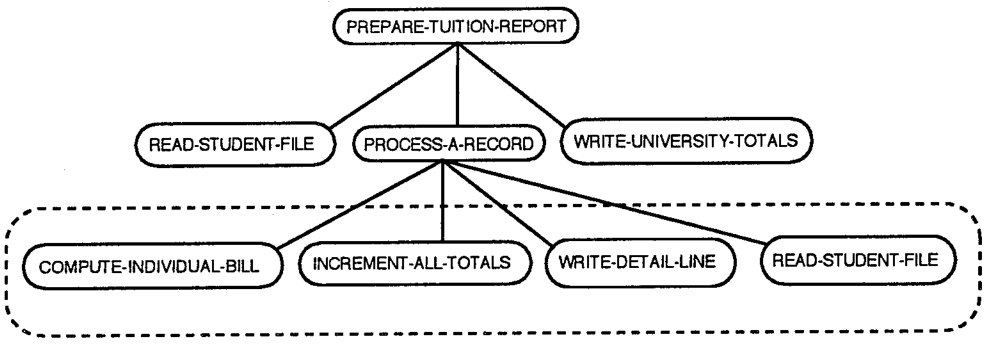
We designed and implemented two partitioning algorithms and experimented with them on our sample industrial programs. The first algorithm uses control flow analysis and data flow analysis to identify instance variables and methods. The rationale for this algorithm came from studying the call graphs of Cobol programs and observing several instances where small numbers of paragraphs cooperated in the processing tasks. These cooperating paragraphs showed up in clusters in the call graph where one paragraph was called by one or more 'clients' and the rest were always called in the same sequence by the same small number of paragraphs within the cluster. We reasoned that these clusters of paragraphs should probably be assigned to the same class because they were cooperating in the processing tasks. A good way to make this happen is to use the call graph relationships to assign methods to classes.

After studying the OO designs produced by the first algorithm, we saw some undesirable features such as basing classes on individual scalar variables and extra messages needed to set instance variable values. To correct these deficiencies, we designed a second algorithm using variable classification and data flow analysis. Variable classification is used to filter scalars and only consider records to become the basis of a class. Then data flow analysis is used to pick the record variable that is referenced the most in the paragraph. The rationale is that messaging will be reduced if we make a paragraph a method of the variables it references the most since the instance variables referenced by a method are directly available without requiring extra messages.

Based on our experiments, we believe that a combination of data flow analysis, control flow analysis and variable classification should be used in any algorithm, and that some optimization will still be required after identifying the initial OO design using any algorithm. We now describe both of the algorithms in detail followed by an evaluation of the OO designs produced by both.

4.1 A control flow based algorithm

The objective of the control flow algorithm is to assign paragraphs clustered in the call graph to the same class. The approach is to visit the paragraphs in breadth-first top-down order and assign a paragraph to the same class as its parent in the call graph if that paragraph references the same variables. If the paragraph does not reference the same variables as its parent, then we try to find a common variable referenced by its siblings and create a new class from that variable.



NOTE: All the methods inside the dashed border are assigned to the same class.

Fig. 3 Call graph for tuition program

To evaluate this algorithm, we used a textbook Cobol program that computes college tuition amounts [See the call graph in Fig. 3]. We focus on the four paragraphs called by PROCESS-A-RECORD. Because PROCESS-A-RECORD does not use or define any variables, it is not assigned to a class, so the algorithm looks for a common variable referenced by the four siblings called by it. It finds STUDENT-RECORD and assigns the four paragraphs to that class. The resulting class diagram in Fig. 4 shows that several methods in class STUDENT-RECORD must use 'get-' and 'set-' methods from classes INDIVIDUAL-CALCULATIONS and UNIVERSITY-TOTALS because they reference instance variables in those classes. In Fig. 4, an arrow from one method to another means the first method sends a message to the second method to request its services. Also, method names beginning with 'get-' and 'set-' provide access to private instance variables of the classes. We guessed we might be able to reduce the number of messages using a different OO design and developed a second algorithm for that purpose.

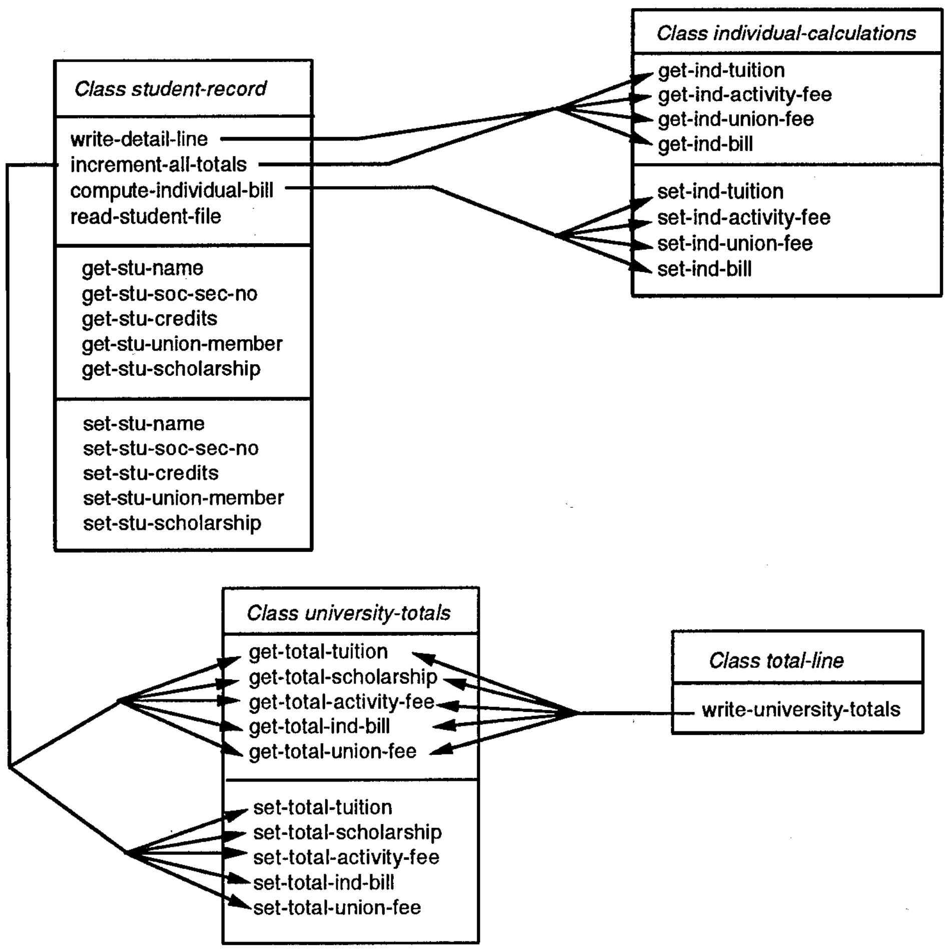


Fig. 4 Tuition class diagram using control flow algorithm

4.2 A data flow based algorithm

We made two observations concerning the OO designs produced by the control flow based algorithm: (1) some classes were based on single scalar variables as instance variables instead of record variables, and (2) extra messages were needed to access instance variables because methods referenced instance variables other than those of their own classes. To fix the problem with item (1), we used variable classification to consider only record variables as instance variables of classes. To fix the problem with item (2), we used data flow analysis to assign a paragraph to the record variable it referenced the most. This approach could reduce the number of messages in the program and create classes with more instance variables. Fig. 5 shows the updated class diagram for the tuition program using the second algorithm. We see all methods that define instance variables have been assigned to the classes of those instance variables. For example method INCREMENT-ALL-TOTALS is assigned to class UNIVERSITY-TOTALS. This resulted in no requirement to use 'set-'

methods saving 21 messages (in a static analysis) when compared to the class diagram for the tuition program in Fig. 4 using the control flow algorithm.

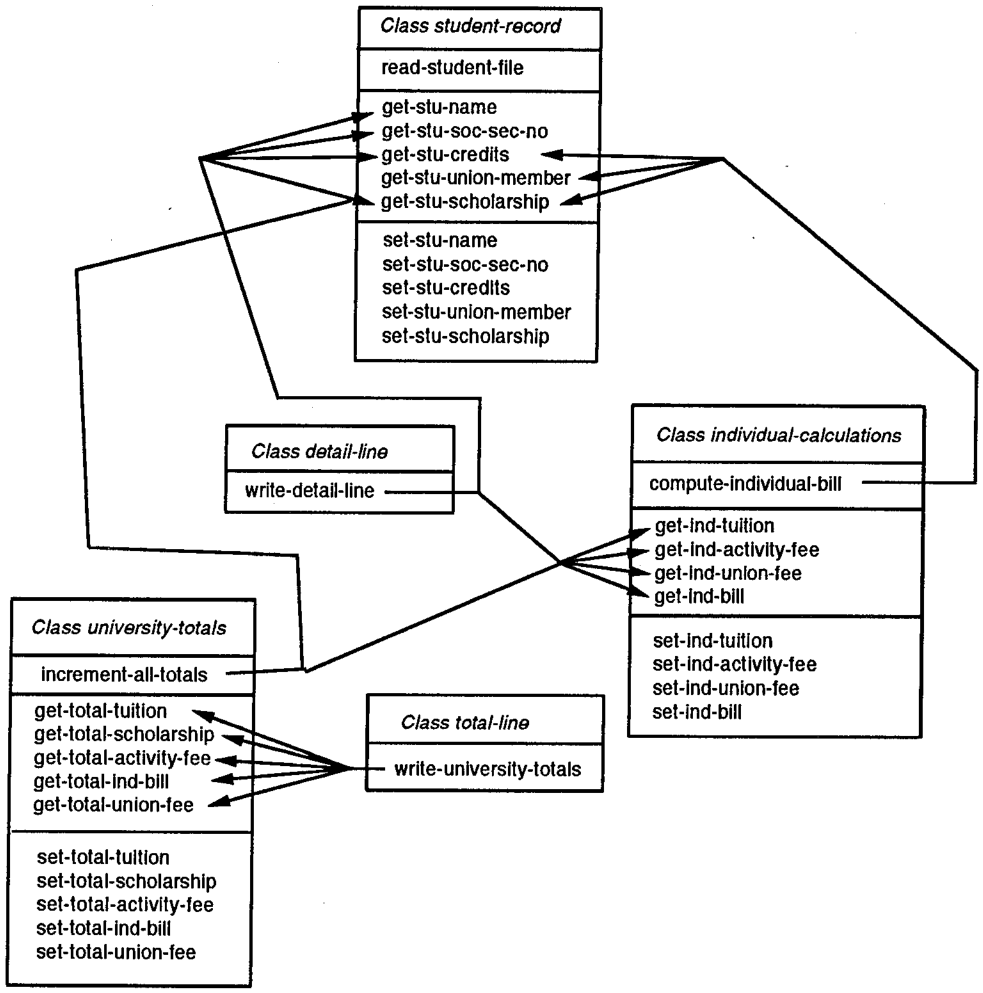


Fig. 5 Tuition class diagram using data flow algorithm

4.3 Evaluation using industrial programs

We also ran both algorithms and produced OO designs for the ten industrial programs described in Section 2. Both algorithms produced OO designs following the guidelines of Demeter's Law but we found differences between the algorithms in terms of the number of classes and methods identified and the number of class interactions involved. The control flow algorithm picked fewer classes than the data flow algorithm (80 versus 97) because it favors assigning new methods to existing classes rather than creating new classes for them. We next evaluated the average number of class interactions in the two OO designs and found that the classes in the control flow based design interacted with approximately 3 ± 2 additional classes on average while the classes in the data flow based algorithm interacted with only 2 ± 2 classes. Additionally, we found that 7 of 26 classes (27%) in the control flow design needed interactions with more than 3 other classes, while in the data flow design only 4 of 27 classes (15%) needed that many additional interactions. Thus, the OO design produced by the data-flow algorithm reduced the coupling between classes compared to the design produced by the control-flow algorithm.

5. Optimizing an OO design

After identifying instance variables and methods using an automatic algorithm, the OO design should be evaluated and possible optimizations explored. We have identified three areas to look for possible improvement: improving the messaging behavior, improving the class interaction graph, and reducing the number of classes, instance variables and methods. Improving messaging involves reducing the number of messages needed to accomplish program tasks. This can be accomplished by moving a method from one class to another to take advantage of direct access to instance variables needed by the method, splitting large methods to reduce the number of classes involved, or possibly merging classes to bring instance variables into the class where they are used. Improving the class interaction graph involves eliminating cycles in class interactions and reducing the number of classes in the graph. This can be accomplished by moving a method from one class to another or by merging classes. Reducing the number of classes, methods, and instance variables reduces the complexity of the OO design and simplifies its understandability and maintenance.

The optimization of an OO design involves making decisions and tradeoffs, and a good deal of experience is needed to do it well. We cannot give an algorithm, but many of the optimizations depend on the following four basic modifications to the OO design: (1) merging two or more methods into one, (2) moving a method from one class to another, (3) splitting large or complex methods into two or more smaller methods, and (4) merging two or more classes into one. These techniques can be used by themselves or in combination to improve an OO design. We now examine each of these techniques and give some examples to help build experience in using them.

5.1 Merging methods

Merging methods can reduce the complexity of an OO design and improve runtime performance by reducing the number of messages required. Many looping activities in Cobol require the use of multiple paragraphs because the language does not contain sequencing constructs such as FOR, WHILE, or REPEAT. Idioms developed to handle these common sequencing tasks are constructed of multiple paragraphs using the PERFORM–

UNTIL statement. For example, to iterate through a one-dimensional array, two paragraphs are needed: the first paragraph initializes the array index and then performs the second paragraph a fixed number of times. The second paragraph increments the index and processes one array element at a time. This idiom is illustrated in Fig. 6.

```
P1.
   MOVE 0 TO INDEX-1.
   PERFORM P2 100 TIMES.
P2.
   ADD 1 TO INDEX-1.
   MOVE ZERO TO MY-ARRAY(INDEX-1).
```

Fig. 6 A Cobol looping idiom

A method identification algorithm based on identifying existing Cobol paragraphs will create two methods from this type of idiomatic construct. We developed an algorithm that finds these idiomatic clusters in programs and shows them to a programmer for optimization. The algorithm [See Fig. 7] uses control-flow analysis to find paragraphs that potentially cooperate in sequencing activities.

1. Find a paragraph that calls exactly zero paragraphs and is called by exactly one paragraph. This is the potential 'end point' in a cluster.
2. Repeatedly go up the called-by chain for this paragraph until finding a paragraph that is called by more than one other paragraph or calls more than one other paragraph. This paragraph is the potential 'entry point' into the cluster and all intermediate paragraphs are members of the cluster.

Fig. 7 A cluster identification algorithm

In addition to identifying paragraphs participating in sequencing idioms, we provide a list of shared variables, such as INDEX-1 in Fig. 6, that may participate in the idiom. This can help the programmer identify possible temporary or local variables in the optimized method. Depending on the method identification algorithm used, the paragraphs in idiomatic clusters may be in separate classes. In this case, they should all be moved to the same class before attempting to merge them [We discuss heuristics for moving methods in Section 5.3]. To evaluate the impact of merging methods in an OO design, we experimented on the automatically generated OO design for a typical batch data processing application consisting of 83 paragraphs. Results of the automatic cluster analysis on this industrial Cobol program identified 21 potential clusters involving 46 of the 83 paragraphs.

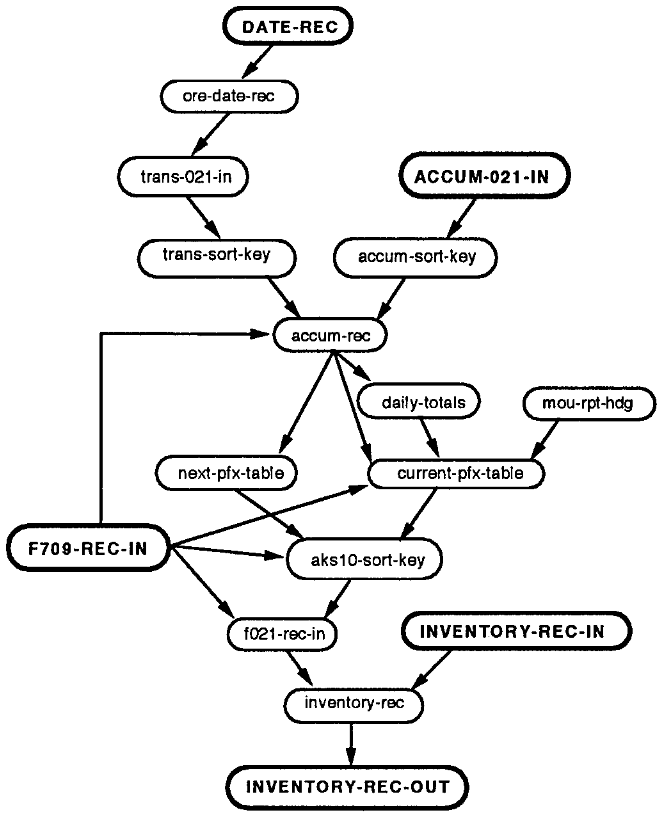


Fig. 8 An excerpt from a class interaction diagram

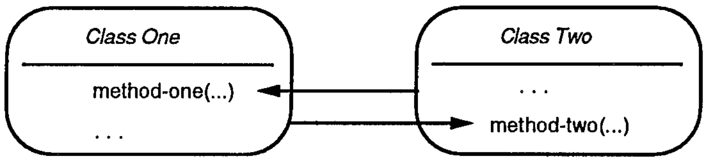
After examining each potential cluster by hand, only 11 of the 21 clusters were suitable for merging into single methods. Some reasons why others were not suitable included paragraphs that were performed just once, paragraphs that were called from multiple locations in a paragraph, and cases where there were no shared variables between the paragraphs. By merging the 11 clusters, we were able to reduce the paragraph count in the program by 17 to 66.

5.2 Analysis of the class interaction diagram

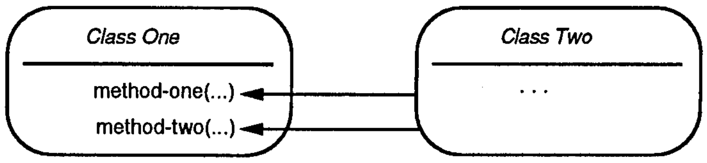
To further assess the OO design in the same sample industrial Cobol program, we constructed a class interaction diagram drawing an edge when a method from one class called a method from another class. Fig. 8 contains an excerpt of this class interaction diagram focusing on the classes involved in producing values of INVENTORY-REC-OUT. Classes in bold such as DATE-REC, ACCUM-021-IN, F709-REC-IN, and INVENTORY-REC-IN can be considered *input* classes because they supply services to other classes without using any themselves, and classes such as INVENTORY-REC-OUT can be considered *output* classes because they receive services from other classes but do not provide any.

On average each of the classes in the program interacted with 2 ± 2 other classes, which is good. However, we noticed that there were 10 instances when there was a direct cycle between two classes. In these cases one method in each of two classes called another method in the opposite class creating a direct cycle. Cycles are legal, but complicate the understanding of OO programs because class relationships are harder to visualize when there are cycles [See Section 3.2]. If classes can be arranged in a diagram with all interaction arrows flowing consistently in one direction from program input classes to program output classes, it is easy to visualize the roles of the classes as well as the high-level sequencing of the processing in the program. For example, there are no direct cycles in Fig. 8 and it is easy to trace the flow of information from the input classes to the output class in this diagram. The ability to visualize the relationships of major entities in a program is helpful in understanding it and maintaining it. Therefore we are interested in breaking cycles in the class interaction diagram.

To break these cycles, we developed two strategies: (1) try to move one or more methods involved in the cycle from one class to the other [See Fig. 9], or (2) try to merge the classes involved in the cycle into a single class. We first find the methods involved in a cycle and then use additional analysis to decide if a method should be moved to another class or whether the classes should be merged.



Before restructuring, these two classes share a cycle of interaction: method-one needs the services of class two and method-two needs the services of class one. In addition, each uses services of its own class.



After restructuring, the cycle is broken.

Fig. 9 Breaking a class dependence cycle by moving a method

5.3 Moving a method from one class to another

In our study, we discovered two heuristics that can be used to decide whether to move a method from one class to another and if so, which one to move. Both heuristics are based on control flow analysis and attempt to move methods to or keep methods in the same class as other methods they call or are called by. The first heuristic addresses the issue of which method to move and says, "move the method that calls or is called by other methods in the class being moved to." The reason for this is further restructuring of the methods in a class is easier when methods from the same class call each other than call methods in other classes. This heuristic can be implemented by checking the other class methods called by both methods in the cycle and moving the one that calls other methods from the class involved in the cycle.

The second heuristic addresses the issue of which method not to move and says, "keep a method in a class if it calls or is called by other methods in that class." Again, the call graph can be used to show this, especially if the call graph is annotated with the class a paragraph is assigned to. Furthermore, a quick scan of the paragraph code can confirm its role with other paragraphs in the class. By looking at call graphs and paragraph code, a programmer can decide which of two methods should be moved to the opposite class in a cycle to break it.

5.4 Merging two classes into one

Another strategy we identified in trying to break cycles in the class interaction graph was to merge the classes involved in the cycle into one class. This breaks the cycle by bringing all the instance variables and methods into the same class, eliminating the need to use messages to access the instance variables. To be suitable for merging, a class should have a small number of instance variables and methods (less than 10), and it should not have interactions with a large number of other classes (more than 2 or 3). In case there is a difference in sizes, merge the smaller class into the larger one. In the industrial program we have been studying, two of the original ten cycles were eliminated by merging the four classes involved.

Later we discovered several cases where we could merge classes that were not involved in cycles simply to reduce the number of classes in the design making it easier to understand and maintain. We used the same size criteria to find candidate classes for merging and looked for cases where one small class had an interaction with only one other class. In the sample industrial program we tested, we found an additional ten classes that could be merged, reducing the original 27 classes in the OO design to only 13.

5.5 Splitting large methods

A final class optimization strategy we discovered involves looking for 'large' methods either in terms of lines of code or in terms of the number of other class variables used. Our criteria for these conditions is: greater than sixty lines of code (one page worth of listing) or greater than two-three class variables involved in addition to the parent class. After finding a candidate method for splitting, an analysis of the code is required to determine if the method really should be split. In the sample application we examined, we found two candidates for splitting based on the number of interacting classes criteria—each of the methods interacted with 4 or more other classes—even though the lines of code were less than the one page limit. In examining the code, however, we only found one of the methods that really could benefit from splitting. This method consisted of 38 lines of code and interacted with 4 other classes. After splitting, one new method had 16 lines of code and interacted with 1 other class, and the other had 22 lines of code and interacted with 3 other classes. A code inspection will readily find the appropriate place to split a method or will show that the method should not be split. The point of a split will be in a straight-line section of the code and will not be in the middle of a loop or other logical block of code. A method that contains complex or cascaded selection statements (IFs) will probably not be amenable to splitting.

6. Conclusion

In exploring algorithms that identify instance variables and methods in existing code, we discovered that variable classification and data flow analysis should play the major role to keep the relative number of messages small. Without an input from data flow analysis, a paragraph may be assigned to a class that requires it to send many messages to other classes, and we showed how 21 messages were saved in two designs of a small program for computing tuition amounts using a data-flow analysis approach. Variable classification can limit the pool of existing variables in the program from which classes are created and force the algorithm to create classes only from 'important' variables. No matter how sophisticated an algorithm is, though, it is unlikely that a completely desirable OO design will be produced automatically. Some of the undesirable characteristics that need to be corrected include: classes with too few instance variables and methods, classes with too many interactions with other classes, classes with cycles of interactions with other classes, and too many small methods. Many of these characteristics are the result of Cobol idioms developed to work around deficiencies in Cobol itself. These can be optimized in an OO design if a programmer knows what to look for. We identified four strategies for improving an OO design and developed algorithms to analyze OO designs and recognize some of these symptoms.

7. Acknowledgment

This research is partially supported by U S WEST Advanced Technologies and Fujitsu Laboratories.

8. References

1. Aho, A.V., Sethi, R., and Ullman, J.D. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

2. Chen, S., Tsai, W.T., and Chen, X.P. SAMEA: An Object-Oriented Environment for Supporting Assembly Programs. *Journal of Software Engineering and Knowledge Engineering* 2, 2 (Feb. 92), 197–226.

3. Chen, X.P., Tsai, W.T., Joiner, J.K., Gandamaneni, H., and Sun, J. Automatic Variable Classification for Cobol Programs. To appear in *Proceedings of IEEE COMPSAC*, 1994.

4. Chidamber, S.R., and Kemerér, C.F. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20, 6 (June 1994), 476–493.

5. Cox, B.J. *Object Oriented Programming*, Addison-Wesley, 1986.

6. Davis, A.M. Fifteen Principles of Software Engineering. *IEEE Software*, 11, 6 (Nov. 1994), 94–97.

7. Goldberg, A. and Robson, D. *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA, 1983.

8. Hecht, M.S. *Flow Analysis of Computer Programs*, North Holland, New York, NY, 1977.

9. Henderson-Sellers, B., and Edwards, J.M. The Object-Oriented Systems Life Cycle. *Communications of the ACM*, 33, 9 (Sept. 1990), 142–159.

10. Joiner, J.K., Tsai, W.T., Chen, X.P., Subramanian, S., Boddu, C., and Sun, J. Data-Centered Program Understanding. In *Proceedings of International Conference on Software Maintenance* (Sept. 19–23, Victoria, B.C.) IEEE, 1994, pp. 272–282.

11. Korson, T. and McGregor, J.D. Understanding Object-oriented: A Unifying Paradigm. *Communications of the ACM* 33, 9 (Sept. 1990), 40–60.

12. Lieberherr, K.J. and Holland, I. M. Tools for Preventative Software Maintenance. In *Proceedings of International Conference on Software Maintenance*, IEEE, 1989, pp. 2–13.

13. Meyer, B. *Object-Oriented Software Construction*, Prentice Hall, Englewood Cliffs, N.J., 1988.

14. Nierstrasz, O. A Survey of Object-Oriented Concepts", in *Object-Oriented Concepts, Databases, and Applications*, edited by Kim, W. and Lochovsky, F., ACM Press/Addison-Wesley, 1989.

15. Ong, C.L. and Tsai, W.T. Class and Object Extraction From Imperative Code. *Journal of Object-Oriented Programming*, (March/April 1993), 58–68.

16. Stroustrup, B. What is Object-Oriented Programming? *IEEE Software*, 5, 3 (May 1988), 10–20.