An Empirical Study of a Repeatable Method for Reengineering Procedural Software Systems to Object-Oriented Systems

William B. Frakes and Gregory Kulczycki
Computer Science Department, Virginia Tech, Falls Church, VA, 22043

Abstract. This paper describes a repeatable method for reengineering a procedural system to an object-oriented system. The method uses coupling metrics to assist a domain expert in identifying candidate objects. An application of the method to a simple program is given, and the effectiveness of the various coupling metrics are discussed. We perform a detailed comparison of our repeatable method with an ad hoc, manual reengineering effort based on the same procedural program. The repeatable method was found to be effective for identifying objects. It produced code that was much smaller, more efficient, and passed more regression tests than the ad hoc method. Analysis of object-oriented metrics indicated both simpler code and less variability among classes for the repeatable method.

Keywords. Reengineering, coupling metrics

1 Introduction

The purpose of our research is to define a method and tools to assist software developers in the process of converting code in a procedural language, such a C, to an object-oriented language such as C++ or Java. Our hypothesis is that it is possible to support this process with a simple repeatable method that yields higher quality code than an ad hoc reengineering effort.

The history of programming and software engineering demonstrates the continual evolution towards larger grained programming constructs and more human focused languages (Frakes and Kang 2005). One aspect of this evolution is the development of more reusable systems based on object-oriented design and programming. One way of achieving this is by reengineering existing procedure-based systems to object-oriented systems. Companies sometimes use the migration from C to C++ or Java, for example, as opportunities for better reuse (Dunn and Knight 1991; Pole 1991; Lanza 2003). In addition to reengineering, identifying objects in procedural programs can enhance understanding of the system design and domain constructs, make debugging easier, and simplify program maintenance (Liu and Wilde 1990; McFall and Sleith 1993; Livadas and Johnson 1994).

Many companies have large inventories of legacy code written in procedural languages. When these companies migrate to new object-oriented architectures, they do not want to start from scratch. Therefore, a need exists for a methodology that can analyze existing procedural code and identify related functions and data that can be encapsulated into reusable objects in the application domain. There have been many studies of the procedural to object-oriented reengineering process (Pole 1991; Newcomb and Kotik 1995; Achee and Carver 1997; Fanta and Rajlich 1998; Valasareddi and Carver 1998;

Cimitile, Lucia et al. 1999; Siff and Reps 1999; Abd-El-Hafiz 2000; Canfora, Cimitile et al. 2001; Gui 2003; Lanza 2003).

Our methodology differs from others both in its simplicity, and its emphasis on empirical analysis and evaluation. Our method analyzes the procedural code and aids the programmer in determining how to create classes from groups of functions (Frakes, Kulczycki et al. 2006; Frakes, Kulczycki et al. 2008). The method uses various coupling metrics to determine how strongly any two functions are related, and therefore, whether they belong in the same class in an object-oriented design. The method uses the premise that program elements that exhibit certain kinds of coupling can be grouped together to form classes. We evaluated eight different coupling metrics of varying degrees of complexity. We also compared our method with an ad hoc reengineering effort. Both the repeatable and ad hoc methods were used to reengineer the ccount metrics tool, written in C, to object-oriented programs in C++. For the ad hoc method, a professional programmer manually reengineered the procedural code (Suryanarayanan and Frakes 2003). The programmer inspected the C code and designed the object-oriented code based on principles that he considered appropriate.

Section 2 of this paper describes the reengineering methodology that our study is based on, and Section 3 details the coupling metrics we used to help identify objects. We present an application of our method in Section 4 to a simple procedural program, and we use it to analyze the effectiveness of the various coupling metrics given in section 3. In Section 5 we compare the results of our method with an ad hoc reengineering effort.

2 Reengineering methodology

The method evaluated in this study proposes steps to be taken in reengineering a procedural system to an object-oriented system. The method delivers reusable objects from existing legacy code. It is based on the premise that program elements that exhibit certain kinds of coupling can be grouped together to form objects. The original steps to be taken in the reengineering process are borrowed from a method suggested by Pole (Pole 1991). They are as follows:

1. The domain expert creates a function stop list. A stop list contains functions identified by the domain expert as utility functions that do not perform tasks specific to the domain.
2. A call graph is generated. A tool or manual scanning of the code base is used to generate a call graph that shows the flow of control in the legacy code.
3. Dependency and context lists are created. A dependency list identifies all the functions invoked from a given function. A context list does the reverse—it identifies the functions that invoke or use a given function.
4. Objects are identified. In this step the metrics are calculated and the potential objects are identified. This step turned out to be the most involved step in the process. We discuss later the details of this process.
5. Domain expert chooses objects. The domain expert examines candidate objects and determines whether they are reasonable. Variables common to two or more functions are examined for their appropriateness as object attributes. Leftover

functions including the functions in the stop list can be converted into individual objects or packaged as utility objects.

In the course of this research, we found that the 4th step (identifying objects) took a disproportionately longer time than the others. We address this in the section on future work and give a revised list of steps that is more in line with what we experienced.

2.1 Comparison with other methodologies

There have been many methods proposed for identifying objects in procedural programs. A brief but good summary can be found in (Canfora, Cimitile et al. 2001). Table 1 presents a comparison of some selected methods. It compares the approach each method takes to object identification, how each method is evaluated, and the extent to which each method is automated.

Newcomb and Kotik proposed a transformation programming system based on the Software Refinery tool (Newcomb and Kotik 1995). The tool took code written in a procedural language and transformed it to an abstract syntax tree. Based on an analysis of the AST, functionally equivalent code was produced in an object-oriented language. The system went beyond simple object identification and attempted to construct class inheritance hierarchies. The tool was designed to look for opportunities where code could be compressed and/or reused. In a successful transformation, the object-oriented code was smaller and the number of total operations was reduced. The process was entirely automated, though human assistance could be used to improve the results.

Siff and Reps proposed a technique for identifying modules in procedural programs using concept analysis (Siff and Reps 1999). When reengineering a C program to a C++ program using their method, "the C program's struct types are the starting point for the C++ program's classes." (p. 750) They demonstrated their technique on a procedural program that included both stack and queue data structures. The groupings suggested by the concept analysis performed well in identifying the functions that belonged to the two data structures. The authors emphasize that a software developer can control the level of granularity of the groupings given by the concept analysis.

Abd-El-Hafiz also focused on object identification in his research, but used cluster analysis for the purpose (Abd-El-Hafiz 2000). He demonstrated his work on two programs, one of them the same ccount program that we use for our example later in the paper. He evaluated the effectiveness of his method by considering whether the proposed groupings corresponded to obvious data structures, and how well they corresponded to the modules in the procedural system.

Canfora et al. reengineered a complex system consisting of around 200,000 lines of code (Canfora, Cimitile et al. 2001). They looked at various object identification methodologies that were available at the time, including concept analysis and cluster analysis. They found that no one technique worked best in every situation. Their reengineering effort ended up using a variety of techniques, all overseen by a domain expert software engineer who chose the grouping tools and made the final decisions about how to group objects.

Approach	Basis for object identification	Evaluation	Automation
Newcomb and Kotik	Software Refinery (TM) transformation programming system	Functional correctness; increase in level of reuse	Achieves a "very high level of automation" in transforming code
Siff and Reps	Use the data in procedural programs, and apply concept analysis to find objects	Appropriate partitioning of functions in multiple data structures	Partitioning process is automated; granularity of results can be dynamically modified
Abd-El-Hafiz	Use a correlation matrix between functions and attributes, and apply clustering neural networks to find objects	Correct identification of data structures; Conformance of clusters to modules in original system	Partitioning process is automated; results should be good enough to avoid the need for a domain expert
Canfora et al.	A domain expert uses a variety of existing approaches to object identification, and tailors the results based on his or her knowledge of the system	In a large reengineering effort, the authors found that no single approach was best in all cases	Object identification was automated using various systems; domain expert made the final decision regarding groupings
Frakes and Kulczycki	Function pairs that are highly coupled are candidates for being methods in the same object	Coupling metrics are evaluated by domain engineers; a reengineered program using the method is compared to a baseline (ad hoc) reengineering effort	Partitioning process can be automated, but simpler metrics can be computed by hand; domain expert makes the final design decisions

Table 1. A comparison of selected approaches to object identification

3 Coupling Metrics

This section describes the metrics that we used in our methodology. Each metric describes a distinct relationship between any two functions in the legacy system. We call them *coupling* metrics because they are based on the various forms of module coupling, such as those given in (Frakes, Fox et al. 1991), and because they indicate the dependency and the amount of communication that takes place between functions.

The metrics can be divided into three broad categories based on the kind of coupling that motivated them.

1. **Invocation metrics.** These metrics are based on routine call coupling as described in (Pressman 2005). They rank functions based on how often one function invokes another.
2. **Shared parameter metrics.** This category currently contains only one metric—the shared parameter metric. It is based on data element coupling as described in (Frakes, Fox et al. 1991), which exists when data is passed from one function to another through a disciplined interface such as a parameter list.
3. **Shared variable metrics.** These metrics are based on data definition coupling as defined in (Frakes, Fox et al. 1991). Data definition coupling occurs when functions manipulate data of the same type.

Our goal is to use these metrics to determine if any two functions in the legacy system belong together in the same class when we move to an object-oriented system. We looked at many metrics because we did not know which ones would be the most effective in identifying objects. We discuss the effectiveness of the metrics we used and the prospect of finding additional metrics in Section 4 of this paper.

The following subsections present eight different metrics—three invocation metrics, one shared parameter metric, and four shared variable metrics. Table 2 gives the definitions of several functions that are used in the definitions of these metrics. With the exception of the source function, these helper functions are self-explanatory. The source function gives the set of variable declarations associated with a particular variable, tracing back through calls if the variable is a formal parameter. We discuss the source function in further detail when we look at the shared variable metric.

Function	**Definition**
invocs(f_1, f_2)	Number of times that function f_2 is invoked in the body of f_1
params(f_0)	{ $v_{t,n}$ | $v_{t,n}$ is a variable of type *t* with name *n* that appears in the parameter list of f_0 }
vars(f_0)	{ $v_{t,n}$ | variable $v_{t,n}$ of type *t* with name *n* appears in the body of f_0 }
source(v, f_0)	{ v_{dec} | variable v appears in f_0 and v_{dec} is a declaration of variable v, or v is a formal parameter in f_0, and $v_{dec} \in$ source(v_1, f_1) where f_1 invokes f_0, and v_{dec} is the actual parameter in that invocation that corresponds to v }
count(v, f_0)	Number of times that variable v appears in the body of f_0

Table 2. Functions used in the definitions of the eight coupling metrics.

3.1 Invocation metrics

When a function f_1 calls another function f_2, it indicates that they perform related tasks and suggests that those functions should be considered for inclusion in the same object. When a method from one class invokes a method from another class, those classes are related by routine call coupling (Pressman 2005). As the name implies, this form of coupling is routine in object-oriented programs. Nevertheless, when a function f_1 calls another function f_2 in a procedural program, it may indicate that f_2 can translate to a private method in same class that contains f_1. Therefore, these metrics may be considered helpful in identifying objects.

Direct invocation metric. This metric identifies the number of times that a function f_1 calls another function f_2. The metric is defined simply as

$$N(f_1, f_2) = \text{invocs}(f_1, f_2).$$

Indirect invocation metric. This metric identifies the number of times that a function f_1 indirectly calls a function f_2 by way of a third function f_{mid}. It is simply the sum of the direct invocation metrics for f_1 and f_{mid}, and f_{mid} and f_2. However, if either of the direct invocation metrics is zero, then no indirect invocation takes place, so the value of the indirect invocation metric is zero. The metric is defined in terms of the direct invocation metric as

$$N_{ind}(f_1, f_2) = N(f_1, f_{mid}) + N(f_{mid}, f_2) \text{ where } N(f_1, f_{mid}) > 0 \text{ and } N(f_2, f_{mid}) > 0$$

Recursive invocation metric. This metric identifies the number of times a function f_1 calls function f_2 and f_2 calls back to f_1. The value of the metric is the sum of the direct invocations from f_1 to f_2 and f_2 to f_1. Like the indirect invocation metric, the value of this metric is zero if no recursion exists. The metric is defined as

$$N_{rec}(f_1, f_2) = N(f_1, f_2) + N(f_2, f_1) \text{ where } N(f_1, f_2) > 0 \text{ and } N(f_2, f_1) > 0$$

3.2 Shared parameter metrics

Data element coupling occurs when modules access shared data that is passed in through a parameter list. If a client passes the same stack to functions in modules M_1 and M_2, then those modules exhibit data element coupling.

Shared parameter metric. This metric identifies the formal parameters that are common between two functions. It does this by counting the number of formal parameters that have the same type and same name. The metric is defined as

$$P(f_1, f_2) = | \text{params}(f_1) \cap \text{params}(f_2) |$$

3.3 Shared variable metrics

Shared variable metrics look at all variables—including parameters, global variables, and local variables—that are shared by functions. These metrics are based on data definition coupling (Frakes, Fox et al. 1991). Data definition coupling occurs when modules manipulate data of the same type. For example, if two modules modify a data structure of type stack, they exhibit data definition coupling.

There are two different kinds of shared variable metrics. The first, more sophisticated, metric considers variables to be shared only if they can be traced to a common declaration. For example, suppose variable *x* is declared in function f_0, which passes it to f_1 and f_2. Furthermore, suppose f_2 obtains *x* through a formal parameter *y*, which it then passes to f_3. Then functions f_0, f_1, f_2, and f_3 are all related, because they all use or manipulate a value that originated with a variable declared in f_0 (see Figure 1).

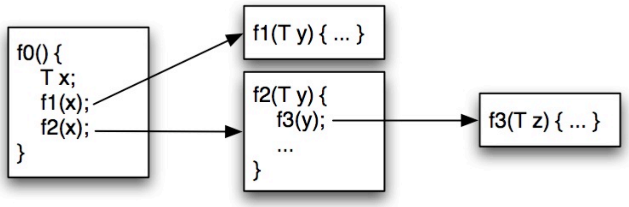

Figure 1. The functions all use a variable that can be traced to the same source.

Shared variable metric. This metric identifies variables in two functions that share a common source. The metric is defined as

$$V(f_1, f_2) = |\ \{ v \mid source(v, f_1) \cap source(v, f_2) \mid \neq \emptyset \}\ |$$

The function source(v, f_1) gives the set of sources (variable declarations) for variable *v* in f_1. If *v* is not a formal parameter in f_1, then *v* will have a unique source. However, if *v* is a formal parameter, then *v*'s source set includes the elements in the source sets of all corresponding actual parameters. Therefore, the size of *v*'s source set may be greater than one.

The simpler version of the shared variable metric considers functions to be related if they share variables with the same type and the same name.

Shared type-name variable metric. This metric identifies all variables in two functions that have a common type and name. The metric is defined as

$$V'(f_1, f_2) = |\ vars(f_1) \cap vars(f_2)\ |$$

We also include a variation for each of these metrics in our analysis. The metrics above count declarations of variables rather than uses. For example, if the only variable shared by two functions was the global stack *s*, the shared variable metric for those functions would be one. Even if *s* appears three times in the body of the first function and four times in the body of the second function, the value of the metric is still one. The metrics below count the static occurrences (the *tokens* rather than *types*) of common variables.

Shared variable tokens metric. This metric counts the static occurrences of all variables in two functions that share a common source. The metric is defined in terms of the shared variable metric as

$$V_{tokens}(f_1, f_2) = \sum count(v, f_1) + count(v, f_2) \text{ where } v \in V(f_1, f_2)$$

Shared type-name variable tokens metric. This metric counts the static occurrences of all formal parameters, global variables, and local variables that are common between two functions. The metric is defined as

$$V'_{tokens}(f_1, f_2) = \sum count(v, f_1) + count(v, f_2) \text{ where } v \in V'(f_1, f_2)$$

Table 3 summarizes the metrics and their definitions.

Name	Notation	Definition
Direct Invocation Metric	$N(f_1, f_2)$	$invocs(f_1, f_2)$
Recursive Invocation Metric	$N_{rec}(f_1, f_2)$	$N(f_1, f_2) + N(f_2, f_1)$ where $\| N(f_1, f_{mid}) \| > 0$ and $\| N(f_2, f_{mid}) \| > 0$
Indirect Invocation Metric	$N_{ind}(f_1, f_2)$	$N(f_1, f_{mid}) + N(f_{mid}, f_2)$ where $\| N(f_1, f_{mid}) \| > 0$ and $\| N(f_2, f_{mid}) \| > 0$
Shared Parameter Metric	$P(f_1, f_2)$	$\| params(f_1) \cap params(f_2) \|$
Shared Variable Metric	$V(f_1, f_2)$	$\| \{ v \mid history(v, f_1) \cap history(v, f_2) \| \neq \varnothing \} \|$
Shared Variable Tokens Metric	$V_{tokens}(f_1, f_2)$	$\sum_{v \in V(f_1, f_2)} count(v, f_1) + count(v, f_2)$
Shared Type-Name Variable Metric	$V'(f_1, f_2)$	$\| vars(f_1) \cap vars(f_2) \|$
Shared Type-Name Variable Tokens Metric	$V'_{tokens}(f_1, f_2)$	$\sum_{v \in V'(f_1, f_2)} count(v, f_1) + count(v, f_2)$

Table 3. Notations and definitions for the eight coupling metrics used in the study.

4 Reengineering *ccount*

The procedural system analyzed in the study was *ccount*, a metrics tools implemented in C that reports counts of commentary and non-commentary source lines and comment-to-code ratios (Frakes, Fox et al. 1991). The ccount tool was initially written in K & R C and later converted to ANSI C. For the purpose of this study the ANSI C version was used.

The statistics collected for the ccount tool including the main function are:

- Number of non-commentary lines of code: 749
- Number of files: 7
- Number of functions: 17

The ccount metric tool was used because it is tractable for a small case study, but non-trivial, so that the case study is still relevant.

This section shows how the method was applied to ccount. Throughout the process of evaluating the proposed method, the following metrics were captured:

- The time taken at each step of the process.
- The number of domain specific objects and utility objects created.
- The number of functions and lines of codes in the legacy system.

The authors acted as the domain experts.

4.1 Domain expert creates function stop list.

For ccount the functions identified to be in the stop list were string manipulation and file manipulation functions that are provided by the standard C libraries. Since the system was relatively small, rather than providing the list as a starting point, we analyzed the output from the next step to help us come up with the functions to be placed in the stop list. The time taken for this step was 1 hour.

4.2 A call graph is generated

The cflow tool was used to identify the flow of control (call structure) of ccount. The output from cflow is in a text format, which we then converted to the graphical representation given in Figure 2. The cflow tool provides options to generate output in both a top-down and bottom-up manner. The graphical representation of the bottom-up output would simply be the call graph in Figure 2 with the arrows reversed. The time taken for this step was 2 hours.

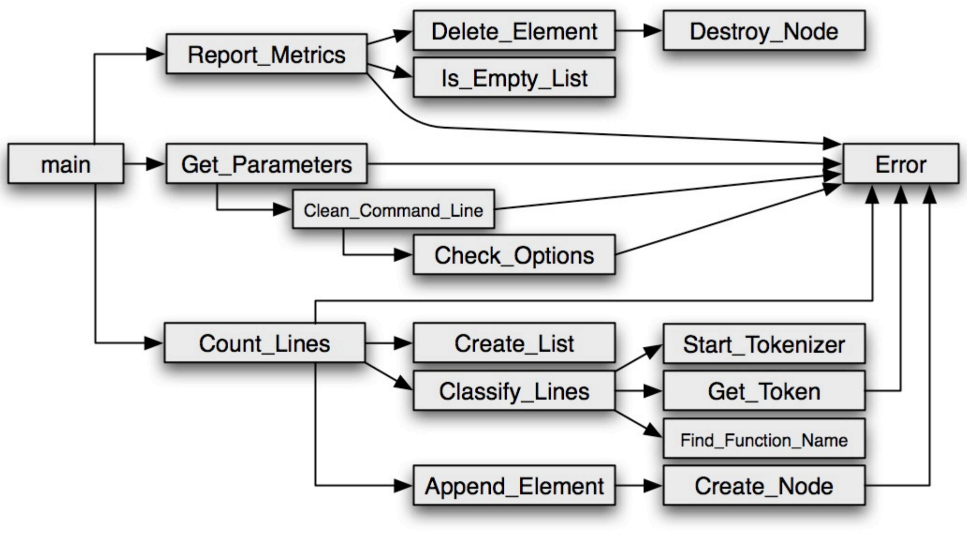

Figure 2. Call graph for ccount tool.

4.3 Dependency and context lists are created.

Using the call graph created in the previous step, the dependency list and the context list were created. The dependency list indicates the functions that are invoked by a given function. For example, the function Classify_Lines uses functions Start_Tokenizer, Get_Token, and Find_Function_Name. The context list indicates the functions that invoke a given function. For example, Create_Node is used by Append_Element. In this example, the only function invoked by multiple functions is Error, which is used by seven other functions. The time taken for this step was 2 hours.

4.4 Objects are identified

This step was by far the most involved and the most time-consuming. Therefore, to make the presentation clearer we have divided it into three sub-steps: collection of summary data, calculation of metrics, and identification of candidate objects. The total time taken for this step was 48 hours.

4.4.1 Summary data is collected

To determine the various metrics, we first identified the variables and functions accessed by each individual function. The collection of the required data for each function was done manually. Lack of a tool for collecting the data made the process time consuming.

For each function the following data was collected.

- The parameters passed to it
- The local variables defined and accessed
- The global variables accessed
- The functions invoked along with the parameters passed to those functions
- The data type returned by the function

For each variable (parameters, local variables, and global variables) the following was captured.

- Its name
- Its data type
- Its scope
- The number of static accesses made to it

Shared variables were identified by looking at each file to determine global variables or local variables manipulated by a function. Ccount did not have any global variables, but it did have variables with file scope that were manipulated by more than one function in that file.

Summary Data Collection Table		
Check_Options (params.c)		
Parameters	char *options	2
	char *optionargs	1
Global variables		0
Local variables	char *ch_ptr	8

Functions invoked	Error	2
Clean_Command_Line (params.c)		
Parameters	char *options	2
	char *optionargs	2
	char **argv[]	9
	int *argc	6
Global variables		0
Local variables	char **new_argv	24
	char **files	9
	char *ch_ptr	11
	int new_argc	18
	int num_files	6
	int arg_index	16
	int file_index	4
Functions invoked	Check_Options(options, optionargs)	1
	Error(…)	8

Table 4. Summary data for functions Check_Options and Clean_Command_Line.

An example of the information collected in this step is given in Table 4. From the table we can tell, for example, that the function *Check_Options* has two parameters, *options* and *optionargs*. The parameter *options* is accessed twice in the body of the function, and *optionargs* is accessed once. The function also accesses the locally defined variable *ch_ptr* eight times, and it invokes the function *Error* twice.

4.4.2 Metrics are calculated

Once the summary data for each function was collected, the coupling metrics were calculated for each pair of functions, provided that neither function is in the stop list. For example, Table 5 gives the data invocation metric calculated for the ccount functions. Function pairs that had a metric value of zero were not included in the table.

First function (f1)	Second function (f2)	N(f1, f2)
Main	Get_Parameters	1
Main	Count_Lines	1
Main	Report_Metrics	1
Get_Parameters	Clean_Command_Line	1
Get_Parameters	Error	1
Count_Lines	Create_List	1
Count_Lines	Error	1
Count_Lines	Classify_Line	1
Count_Lines	Append_Element	3
Report_Metrics	Error	2
Report_Metrics	Is_Empty_List	1
Report_Metrics	Delete_Element	1
Clean_Command_Line	Check_Options	1
Clean_Command_Line	Error	8
Classify_Line	Start_Tokenizer	1
Classify_Line	Get_Token	1
Classify_Line	Find_Function_Name	1
Append_Element	Create_Node	2
Delete_Element	Destroy_Node	1
Check_Options	Error	2
Get_Token	Error	1
Create_Node	Error	2

Table 5. Non-zero direct invocation metrics for ccount.

Most of the metrics can be calculated simply by inspecting the summary data for the two functions involved in the metric. The exceptions are the indirect invocation metric, the shared variables metric, and the shared variable tokens metric. Table 6 shows each of the eight metrics in which the first function (f1) is *Clean_Command_Line* and the second function (f2) is *Check_Options*. The following paragraphs indicate how to calculate each of these metrics.

$N(f_1, f_2)$	1
$N_{rec}(f_1, f_2)$	0
$N_{ind}(f_1, f_2)$	0
$P(f_1, f_2)$	2
$V(f_1, f_2)$	2
$V_{tokens}(f_1, f_2)$	7
$V'(f_1, f_2)$	3
$V'_{tokens}(f_1, f_2)$	26

Table 6. Metrics for f1 = Clean_Command_Line and f2 = Check_Options.

Invocation metrics. From Table 3 we see that Clean_Command_Line calls Check_Options once, yielding a direct invocation metric of one. Since Check_Options never calls Clean_Command_Line back, the recursive invocation metric is zero. In fact, in this particular study all of the recursive invocation metrics turned out to be zero. The *indirect* invocation metric requires slightly more work. Looking at Table 3, we see that the only other function besides Check_Options that is called by Cleam_Command_Line is the Error function, which is called eight times. If the Error function (whose record is not shown in Table 3) had called Check_Options *n* times, then the indirect invocation metric would have been $8 + n$. Since the Error function never actually invokes Check_Options, the indirect invocation metric is zero. Note that the direct and indirect invocation metrics are not necessarily symmetric. For example, we do not—in general—have $N_{ind}(f_1, f_2) = N_{ind}(f_2, f_1)$. However, the recursive invocation metric is symmetric.

Shared parameter metrics. We can also tell directly from Table 3 that functions Check_Options and Clean_Command_Line both have a parameter named *options* of type *char* and a parameter named *optionargs* of type *char*. For this reason, the value of the shared parameter metric is two. Note that, in this study, we ignored pointers when determining types—so variables declared with *char*, *char***, and *char*[] were all considered to have the same type.

Shared variable metrics. To calculate the shared variables metric we must determine which variables in Clean_Command_Line and Check_Options can potentially originate from the same source. From Table 3 we see that there are only three variables in Check_Options, so there are three candidates. The variable *ch_ptr* is declared in the body of Check_Options, so the only way that Clean_Command_Line can share this variable is if it is passed to Clean_Command_Line through some sequence of function calls. However, a quick look at the flow graph (Figure 2) tells us that although Clean_Command_Line calls Check_Options, there is no call path from Check_Options to Clean_Command_Line. Therefore, even though Clean_Command_Line also has a variable named *ch_ptr* of type *char*, they are not considered shared for the purposes of this metric. On the other hand, both *options* and *optionargs* are formal parameters in Check_Options, and since Clean_Command_Line calls Check_Options, we know that

Check_Options must share its formal parameters with the actual parameters passed to it by Clean_Command_Line. The fact that the actual parameters passed by Clean_Command_Line also happen to be named options and optionargs is unrelated to the calculation of this metric; the relevant fact is that the variables come from the same source. Thus, the value for the shared variable metric is two, and the value for the shared variable tokens metric is the sum of static occurrences for these variables in each function: 3 in Check_Options + 4 in Clean_Command_Line = 7.

The shared type-name variables metric is significantly easier to calculate. Both functions have variables with type-name combinations *char*/*options*, *char*/*optionargs*, and *char*/*ch_ptr*. Therefore the value of this metric is three, and the value of the shared type-name variable tokens metric is: 11 occurrences of these variables in Check_Options + 15 occurrences in Clean_Command_Line = 26.

4.4.3 Candidate objects are identified

Once the individual metrics have been are calculated, a threshold is determined for each metric, and each metric is individually evaluated to come up with candidate objects. In this study, the following guidelines were taken into consideration.

- In C++ the function main is not part of any object, therefore the coupling metrics in relation to that function were not used.
- If the coupling metric for two functions was above or equal to the threshold value, both were placed in the same object.
- If a function f_1 has the same coupling metric with multiple functions in different objects, then this is used as an indication that f_1 should be placed in a separate object as it might be a utility function.

The decision of which threshold to use was empirical to ensure that functions don’t cluster in one object. In the case of the direct invocation metric, the vast majority of function pairs had a metric value of zero, several functions had a value of one, and a few functions had a value greater than one (see Figure 3). A threshold value of one was chosen—a value of anything greater than one would have meant that too many functions would be in classes by themselves.

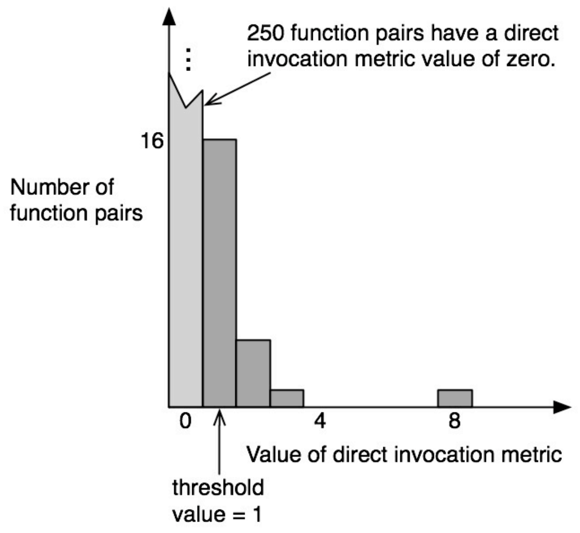

Figure 3. Distribution of values for the direct invocation metric.

Using the guidelines outlined above, the main function was placed in a class by itself, and the Error function was identified as a utility function, so it was also placed in a separate class. This led to the following partitioning of the functions into objects.

Object 1	Get_Parameters, Clean_Command_Line, Check_Options
Object 2	Count_Lines, Classify_Line, Start_Tokenizer, Get_Token, Find_Function_Name, Create_List, Append_Element, Create_Node
Object 3	Report_Metrics, Is_Empty_List, Delete_Element, Delete_Node
Object 4	Error
Object 5	Main

The process of determining a threshold and finding candidate objects was repeated for all of the metrics, yielding the partitioning of functions in Table 7. The recursive invocation metric is not included because recursive calls did not occur in the application.

Metric	Candidate Objects
Direct invocation	*Get_Parameters, Clean_Command_Line, Check_Options*
	Count_Lines, Classify_Line, Start_Tokenizer, Get_Token, Find_Function_Name, Create_List, Append_Element, Create_Node
	Report_Metrics, Is_Empty_List, Delete_Element, Delete_Node
	Error
Indirect invocation	*Get_Parameters, Clean_Command_Line, Check_Options*
	Count_Lines, Classify_Line, Start_Tokenizer, Get_Token, Find_Function_Name, Append_Element, Create_Node
	Report_Metrics, Delete_Element, Delete_Node
	Error, Create_List, Is_Empty_List

Shared parameters	*Get_Parameters, Clean_Command_Line, Check_Options*
	Count_Lines, Classify_Line, Start_Tokenizer, Report_Metrics, Error
	Delete_Element, Append_Element, Create_Node, Is_Empty_List, Create_List
	Destroy_Node, Find_Function_Name, Get_Token
Shared variables	*Get_Parameters, Clean_Command_Line, Check_Options*
	Count_Lines, Classify_Line, Start_Tokenizer, Get_Token, Find_Function_Name, Append_Element, Create_Node
	Report_Metrics, Delete_Element
	Error, Create_List, Is_Empty_List, Destroy_Node
Shared variable tokens	*Get_Parameters, Clean_Command_Line, Check_Options*
	Count_Lines, Classify_Line, Start_Tokenizer, Get_Token, Find_Function_Name, Append_Element, Create_Node
	Report_Metrics, Delete_Element, Destroy_Node
	Error, Create_List, Is_Empty_List
Shared type-name variables	*Get_Parameters, Clean_Command_Line, Check_Options*
	Count_Lines, Classify_Line, Start_Tokenizer, Get_Token, Find_Function_Name
	Report_Metrics, Create_Node, Append_Element, Delete_Element
	Error, Create_List, Is_Empty_List, Destroy_Node
Shared type-name variable tokens	*Get_Parameters, Clean_Command_Line, Check_Options*
	Count_Lines, Classify_Line, Start_Tokenizer, Get_Token, Find_Function_Name
	Report_Metrics, Create_Node, Append_Element, Delete_Element
	Error, Create_List, Is_Empty_List, Destroy_Node

Table 7. Candidate objects for each of the coupling metrics.

4.5 Domain expert chooses objects

In this step the domain expert analyzed the objects for reasonableness. Each metric was analyzed individually, and the results of this analysis are presented here. One of the criteria used in the analysis was whether the partitions corresponded to the modules in the C program, which exhibited good modular design in the first place. In particular, we were always interested to see if the candidate objects for a given coupling metric successfully identified the list data type. The time taken for this step was 16 hours.

The direct invocation metric provides a good breakup of the objects, but was unable to satisfactorily identify the list data type. It groups the functions that relate to extracting parameters since those functions invoke each other. However, the list functions do not necessarily invoke each other. The indirect invocation metric provides a breakup of objects very similar to the direct invocation metric. And similarly, it is not able to identify the list data type. This may indicate that these metrics will give similar results in general. If so, then the direct invocation metric should be used since it is easier to calculate.

The shared parameters metric is able to identify the list data type as it clusters all but one function in the same object. It places the functions Error and Report_Metrics in the same object as functions which classify lines. Since this metric only considers the parameter list of functions it does not always separate functions that have separate responsibilities.

The calculation of the shared variable metrics in general took up a substantial amount of time, but their results were not very different to the direct invocation metric. None of the shared variable metrics were able to identify the list data type; they all tended to have the functions related to the abstract data type either in the utility object or grouped with the Report_Metrics function.

Most coupling metrics placed the function Report_Metrics in a separate object. The task of reporting metrics (in ccount) follows that of counting and classifying lines, and hence it is best to use different classes for these to separate responsibilities.

If the list data type were already identified, the direct invocation metric would be the fastest and easiest to use to help determine objects. The shared parameters metric provides the best breakup of the objects; it comes closer than any other metric in identifying the list data type.

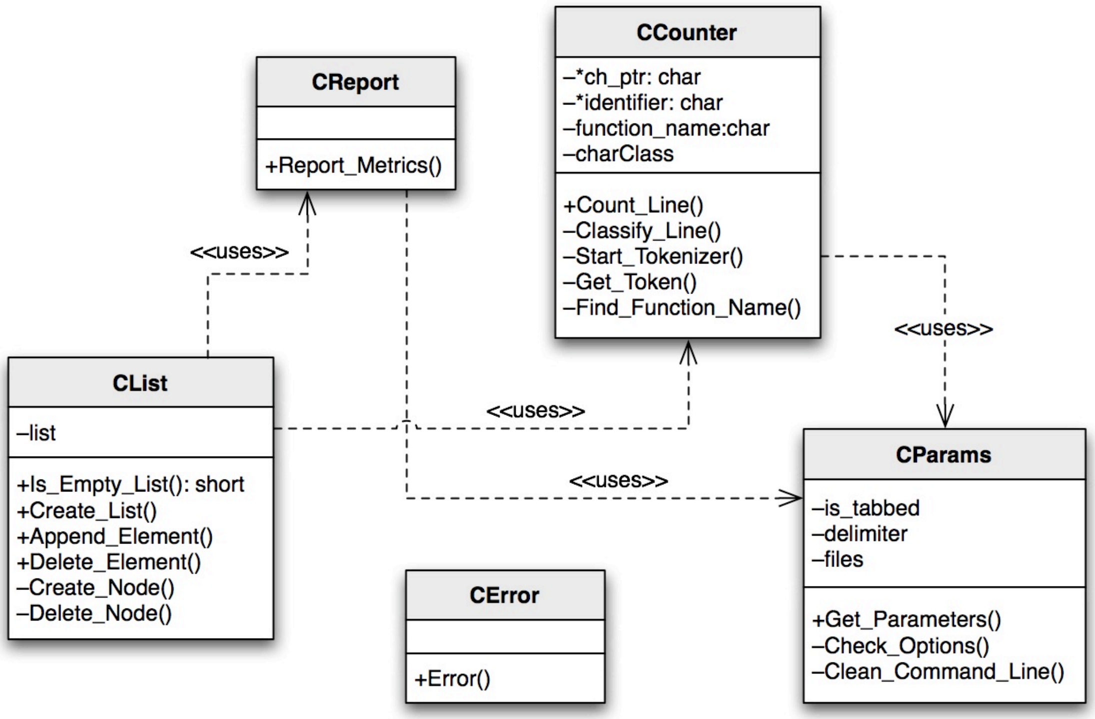

Figure 4. Class diagram for object-oriented ccount application.

Based on the above observations and using the candidate objects as references, we chose the following classes for coding the object oriented version of ccount. The list data type is identified and encapsulated in its own class. The functions main, Error, and Report_Metrics were each placed in their own class. Figure 4 gives a class diagram of the application.

```
Class::CError
   Error()
Class::CCount
   main()
Class::CReport
   Report_Metric()
Class::CCounter
   Count_Lines()
   private:
      char *ch_ptr
      char identifier[MAX_IDENT+1]
      char function_name[MAX_IDENT+1]
      char_class charClass[128]
      Classify_Line()
      Start_Tokenizer()
      Get_Token()
      Find_Function_Name()
Class::CParams
   Get_Parameters()
   private:
      short is_tabbed
      char *delimiter
      char **files
      Check_Options()
      Clean_Command_Line()
Class::CList
   Is_Empty_List()
   Create_List()
   Append_Element()
   Delete_Element()
   private:
      CElement *list
      Create_Node()
      Destroy_Node()
```

4.6 Coding

For coding in C++ the following guidelines were followed.

- Rather than using malloc and realloc functions to allocate memory, new was used.
- Rather than using #define, const was used.

- Some variables had to be renamed to adhere to C++ naming convention.

Otherwise, an effort was made to keep the function names the same and the algorithms the same. Due to the similar structure and syntax of the C and C++ languages, it was possible at times use the C functions with few changes.

The parameters extracted from the command line were placed as private variables in the class CParams and were accessed using public access get methods. The list was made a private variable in the CList class; only the methods in the CList class modified the list.

The global (file scope) variables accessed by the functions Get_Token, Start_Tokenizer, and Find_Function_Name were made private variables of the class CCounter.

To ensure that the code developed in C++ gave the same result as the C version, the 19 regression tests developed for C code in (Frakes, Fox et al. 1991) were utilized. Abnormal inputs were provided to check if the code is able to handle them. And the output generated for the statistics of a valid C file was verified to ensure that it was accurate. The C++ version was found to perform satisfactorily.

Time taken for the coding of ccount in C++ was 24 hours.

4.7 Process variables captured

The times taken for each step are shown in Table 8. The total time taken for the process was 93 hours. Though we did not record the times it took to calculate each metric in identify objects step, we estimate that we did not spend more than six hours calculating the direct invocation metric and the shared parameter metric—the two metrics that seemed to give the best results.

Step	Time taken
Create stop list	1 hour
Create flow graph	1 hour
Dependency list	2 hours
Identify objects	48 hours
Domain expert analysis	16 hours
Coding	23 hours
Total	93 hours

Table 8. Process Variables

The following data was captured for the ANSI C version and C++ version of ccount.

Statistics for the C version:

- Number of non-commentary lines of code : 749
- Number of files : 7
- Numbers of Functions : 17

Statistics for the C++ version:

- Number of objects : 6
- Number of real objects : 4
- Number of utility objects : 2
- Real objects with one function : 1
- Number of non-commentary lines of code : 679

5 Comparison with Ad Hoc Reengineering effort

This section compares our method with an ad hoc reengineering effort conducted on the same program by a professional programmer.

5.1 Ad hoc reengineering effort

In the ad hoc method, ccount was reengineered to C++ using standard reusable components and a singleton design pattern to capture utility classes.

When trying to determine how to reengineering the ccount program, the programmer considered several possibilities. A very simple approach would have been to take the modules or functions in the existing language and wrap them in modules or functions from the other language. This ensures that the resulting product is in the target language while not changing the functionality and the results by much. This approach is not optimal since, even though the conversion is complete, the new product does not use all the benefits and features of the new language. This is especially true when the source language is C and the target language is C++. Since C++ is backwards compatible to C, a very simple conversion would be to change the extensions on the files to .cpp and change printf's to cout's and be done. But, the resulting product would still be C in C++ clothing.

Another option is to start from first principles. This involves looking at the problem statement, identifying the objects that stand out in the problem, and designing and developing the product from the ground up. The process involves defining attributes and methods for the various objects and creating classes for these objects. This option produces better code with most utilization of the features of the destination language than the first option. Also, since this design is from the ground up, one can take advantage of various optimizations from the beginning and support quality and maintainability from the start. This approach, however, is poor reuse because it requires a from-scratch development effort. This approach takes much longer for the conversion.

In the ad hoc method, the programmer started from first principles in identifying objects and, once the objects were identified, the existing functions were remapped into methods that were appropriate for those objects. By doing so, he eliminated some functions, added new ones, and replaced existing ones with those from the standard C++ libraries. Sometime he replaced the functions with simpler ones that took advantage of the progress made in software platforms, and the portability of code that comes with using ANSI standards.

Since the programmer decided to use the existing functionality and not rewrite from the ground up, he was left with some functions that did not belong to any of the objects he identified. Some of these needed to be global since they maintained state information within the function between calls. These functions were packaged into a utility class using the singleton design pattern (Gamma, Helm et al. 1994) to achieve a single instance of the object. In addition, due to time constraints, he left the parsing algorithm used for the classification of a line the same as it was in the C version.

From the statement of the problem and first principles, the programmer identified three distinct objects:

- File – An object that needs to be analyzed, and one in which CSL (comment source lines), NCSL (non-comment source lines), and the ratio of CSL to NCSL must be determined. At least one file must be analyzed during any invocation.
- Func – An object at the lowest granularity that needs to be analyzed and whose metric must be reported. Every function belongs to one file, and a file can contain one or more functions. Code external to a C function is treated as belonging to a function named external.
- Line – An object that needs to be classified as either external or belonging to a function. It may be a comment, non-comment, neither, or both. Every line belongs to only one function and a function has one or more lines.

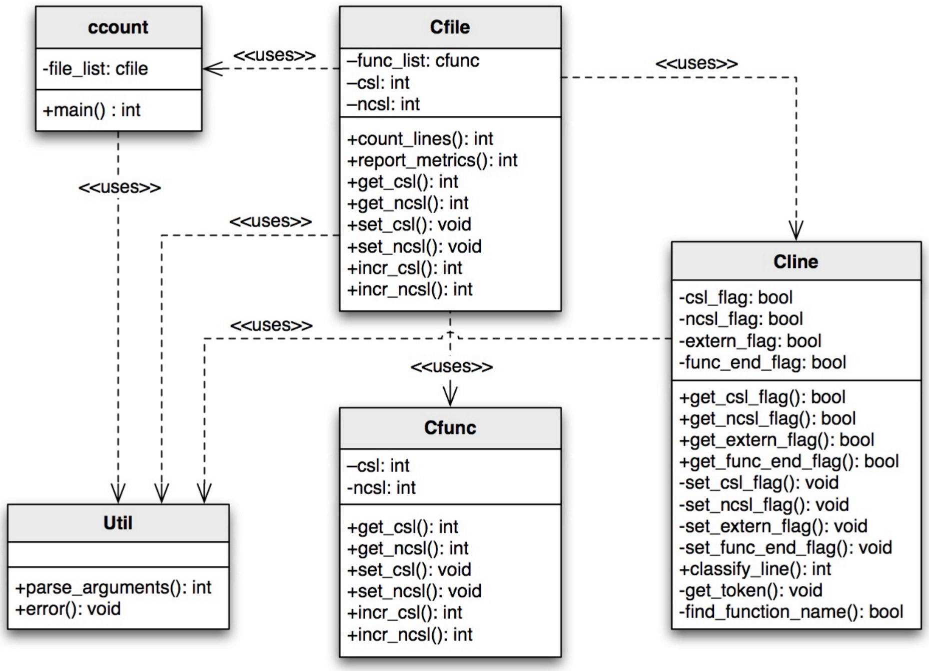

Figure 5. A UML class diagram of the code produced by the ad hoc method

These objects have certain attributes, and the programmer found a very good match of C functions with methods of these objects, though with a few changes. In addition, there are other functions and modules such as error checking and reporting, and command-line parsing that are either external to these objects or are not confined to one object. Finally, the sturdier, more generic and more optimal list container from the C++ Standard

Template Library can safely replace the C code for linked-list generation, maintenance, and deletion.

Figure 5 gives a UML class diagram of the reengineered code. The diagram represents all of the classes in the resulting code, but it omits some of the methods due to space constraints. Since the reengineering in this study was done by a professional programmer, this serves as a baseline for the repeatable method.

5.2 Comparison of Results

In this section we compare the ad hoc and repeatable method on several metrics. First we compared the two methods in terms of regression testing. As can be seen in Figure 6, the repeatable method passed more of the ANSI C version regression tests, 11 tests passed, than did the ad hoc method, which passed eight. One key difference between the two methods is in total code size (commentary code + source code). As summarized in Figure 7, the ad hoc method produced 2,481 lines of code, an increase of 61% over the C version. The repeatable method, on the other hand, was virtually the same as the C version producing 1,547 lines of code. As can be seen in Figure 8, the execution speed follows a similar pattern, with a small increase in execution speed for the repeatable method, and a much larger one for the ad hoc method. The differences in execution speed may be partially caused by the increase in code size.

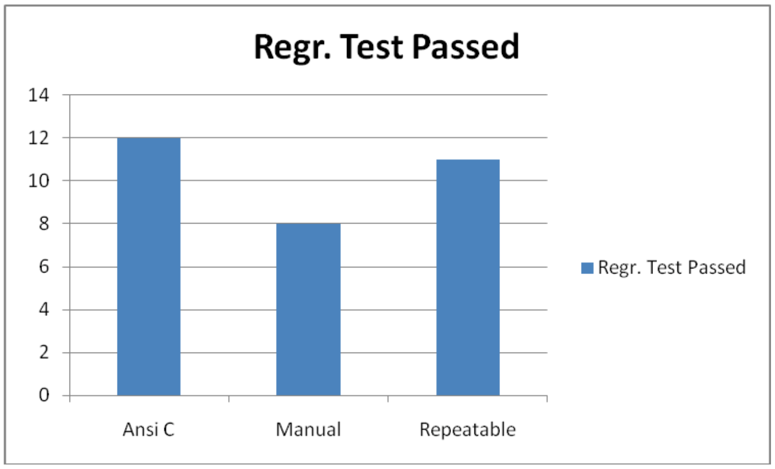

Figure 6. Number of regression tests passed by the original procedural code, the OO code developed using the ad hoc method, and the OO code developed using the repeatable method.

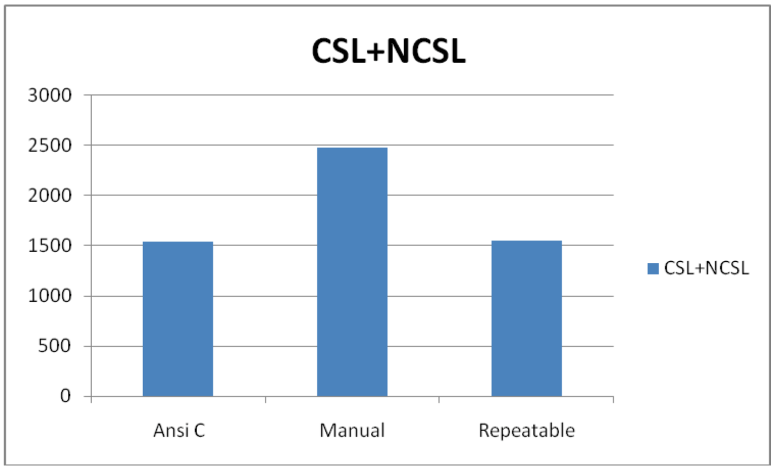

Figure 7. Difference in code sizes of the original procedural code, the OO code developed using the ad hoc method, and the OO code developed using the repeatable method.

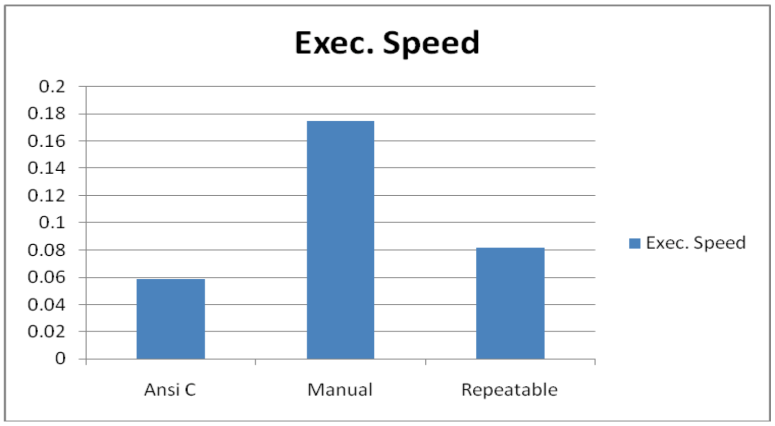

Figure 8. Difference in execution speed of the original procedural code, the OO code developed using the ad hoc method, and the OO code developed using the repeatable method.

The increase in code size was also reflected in numbers of methods. The ad hoc method produced 59 methods vs. 20 for the repeatable method. The ad hoc method produced four custom classes and reused the List class from the C++ standard template library. The repeatable method produced five custom classes. The average number of methods per class was therefore 14.75 for the ad hoc method and four for the repeatable method.

Of the 59 methods in the ad hoc method, 49 of them were public and 10 were private. The programmer using the ad hoc method included a large number of accessor methods in his code (27 in total), so this increase may in part be due to his design decision to make heavy use of accessor methods. Of the 20 methods produced in the repeatable method, 12 were public and 8 were private. This is much closer to the 16 functions in the procedural code. The repeatable code had 5 accessor methods, while the procedural code had 2. The ad hoc method also had an equal or greater number of I/O and read/write methods than the repeatable method. The ad hoc method produced 2 I/O methods and 23 read/write methods, while the repeatable method also produced 2 I/O methods, but produced only

17 read/write methods. The original code contained 3 I/O methods and 12 read/write methods. The number of I/O and read/write methods is suggested as a predictor of good reuse components in (Selby, 1989). The spider graph in Figure 9 gives an overview of these method numbers.

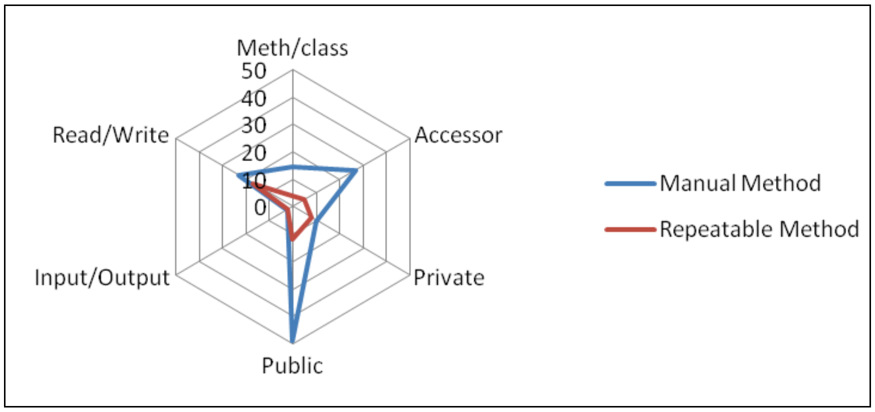

Figure 9. Spider graph comparison of metrics related to numbers of methods in code generated using the ad hoc and repeatable method.

5.2.1 Evaluation using object-oriented metrics

We evaluated the ad hoc and repeatable versions of the code using object-oriented metrics found in (Chidamber and Kemerer 1994). The metrics were weighted methods per class (WMC), coupling between object classes (CBO), response for a class (RFC), and lack of cohesion in methods (LCOM). The other metrics associated with this set, depth of inheritance tree (DIT) and number of children (NOC), were not used since neither method produced code with inheritance. The object-oriented metrics are designed to work on a per class basis. We report them here for each class and also give the average over all classes in each system. Interpreting these metrics is not always straightforward, though extreme numbers indicate a possible need to redesign the class (Chidamber and Kemerer 1994). There is also some indication that higher numbers can lead to more problems (Chidamber, Darcy et al. 1998).

The weighted-methods-per-class metric gives the number of methods in each class. Weighting certain methods higher than others can change this number. For example, one might decide to give a lower weight to accessor methods or private methods. We have given equal weight to all methods since we have already reported numbers of accessor and private methods. Table 9 gives the weighted methods per class, which in this case is the same as the number of methods per class. The methods per class in the code produced with the ad hoc method are almost always higher than those in the code produced with the repeatable method.

Ad hoc method		Repeatable method	
Class	**WMC**	**Class**	**WMC**
Cfile	19	CCounter	6
Cfunc	15	CError	1
Cline	21	CList	6
Util	5	CParams	6
		CReport	1
Mean	15	Mean	4
Median	17	Median	6
Range	16	Range	5

Table 9. Weighted methods per class (WMC) for the ad hoc and repeatable methods.

For the coupling-between-object-classes metric, a class is considered coupled to another class if it uses attributes or methods from the other class, or vice versa. Therefore, if class A is coupled to class B, then class B must be coupled to class A. Note that the coupling metrics used in the repeatable method were based on many different forms of coupling. In this study both the ad hoc method and repeatable method produced code in which all the attributes were private, so coupling occurs if one class uses the methods of another. Table 10 gives the CBO metrics for each class produced under the different methods. The numbers for the repeatable method are slightly higher.

Ad hoc method		Repeatable method	
Class	**CBO**	**Class**	**CBO**
Cfile	3	CCounter	2
Cfunc	1	CError	4
Cline	2	CList	3
Util	1	CParams	1
		CReport	2
Mean	1.75	Mean	2.4
Median	1.5	Median	2
Range	2	Range	3

Table 10. Coupling between object classes (CBO) for the ad hoc and repeatable methods.

The response for a class is the number of methods in a class plus the number of methods it calls from other classes. For example, if class A has one method that invokes two other methods, both from different classes, then the response for class A is three. All methods are counted only once. Table 11 gives the response for each class produced by the ad hoc and repeatable methods. The RFC values for the ad hoc method are decidedly higher than those for the repeatable method. The ad hoc mean is more than triple that of the repeatable method, and the range is more than double.

Ad hoc method		Repeatable method	
Class	**RFC**	**Class**	**RFC**
Cfile	28	CCounter	9
Cfunc	17	CError	1
Cline	23	CList	7
Util	6	CParams	7
		CReport	3
Mean	18.5	Mean	5.4
Median	20	Median	7
Range	22	Range	8

Table 11. Response for a class (RFC) for the ad hoc and repeatable methods.

The lack-of-cohesion-in-methods metric tries to measure the cohesiveness of a class. The higher this number, the less cohesive a class is. The most cohesive classes have an LCOM value of zero. The LCOM value is based on the notion that in a cohesive class, most methods will use most of the attributes of the class. If most methods do not use many attributes, the lack of cohesion is higher. The exact formula is LCOM = max(n □ m – 2 (A1 + A2 + … + Am)) where n is the number of attributes in the class, m is the number of methods in the class, and Ai is the number of attributes used by method i. Table 12 give the LCOM values for each class produced by each of the methods. As shown in the table, most of the LCOM values for the classes in the ad hoc method are significantly higher than the values for the classes in the repeatable method. The value for the Cline class is particularly extreme.

Ad hoc method		Repeatable method	
Class	**LOCM**	**Class**	**LOCM**
Cfile	40	CCounter	8
Cfunc	17	CError	0
Cline	138	CList	0
Util	4	CParams	14
		CReport	0
Mean	49.75	Mean	4.4
Median	28.5	Median	0
Range	134	Range	14

Table 12. Lack of cohesion in methods (LCOM) for the ad hoc and repeatable methods.

Figure 10 summarizes the object-oriented metrics using a spider graph. The average metric values for the ad hoc method are higher than those for the repeatable method for all metrics except the CBO metric.

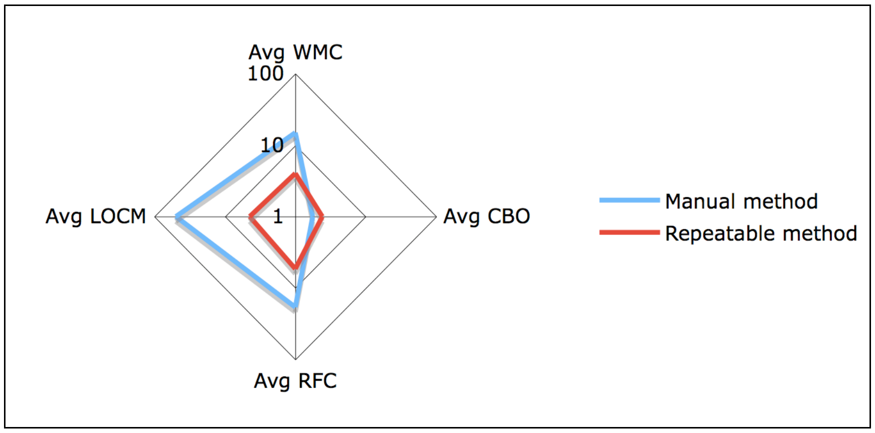

Figure 10. Spider graph summarizing the object-oriented metrics on a logarithmic scale

Metric	ANSI C	Ad hoc	Repeatable	See
Regression tests passed	12/17	8/17	11/17	Figure 5
CSL + NCSL	1542	2481	1547	Figure 6
Execution speed	0.0582 sec	0.175 sec	0.0816 sec	Figure 7
Public methods	N/A	49	12	Figure 8
Private methods	N/A	10	8	Figure 8
Accessor methods	2	27	5	Figure 8
Read/Write methods	12	23	17	Figure 8
Input/Output methods	3	2	2	Figure 8
Avg. WMC	N/A	15	4	Table 3
Avg. CBO	N/A	1.75	2.4	Table 4
Avg. RFC	N/A	18.5	5.4	Table 5
Avg. LOCM	N/A	49.75	4.4	Table 6

Table 13. Summary of evaluation metrics used for comparison.

6 Conclusion

6.1 Revised method

In Section 2 we gave an outline of the steps involved in the Pole method (Pole 1991), which was the method that ours was originally based on. During the course of this study, we discovered that these steps tended to emphasize some tasks that we thought less important, and deemphasize some tasks that we thought more important. We feel that the following list gives a more accurate description of the process.

1. The domain expert creates a function stop list. A stop list contains functions identified by the domain expert as utility functions that do not perform tasks specific to the domain.

2. Summary data is collected. The summary data contains information for each function that is not in the stop list, such as the types and names of parameters, variables, and functions used in the given function. The summary data includes information that would be found in a call graph and in dependency and context lists.
3. Metrics are calculated. Different coupling metrics describe different relationships between functions, such as how many times one function invokes another or how many parameters are shared by the functions. In our study we used eight different coupling metrics and evaluated each one individually for its effectiveness in identifying objects.
4. Identify candidate objects. The software engineer determines a threshold for each metric. If the metric for two functions is above the threshold, those functions are candidates to appear as methods in the same class.
5. Domain expert chooses objects. The domain expert examines candidate objects and determines whether they are reasonable. Variables common to two or more functions are examined for their appropriateness as object attributes. Leftover functions including the functions in the stop list can be converted into individual objects or packaged as utility objects.

This first step is unchanged from the list in Section 2. The second step in this list is a combination of steps 2, 3, and the first part of step 4 from the original list. The third and forth steps here are included in step 4 of the original list. Step 5 is unchanged.

6.2 Summary of results

This study examined two methods for reengineering procedural software systems to object-oriented systems. Our hypothesis was that it is possible to support this process with a repeatable method. We empirically evaluated our method to determine its utility, and found that the repeatable method produced more compact and efficient code, and passed more regression tests than did the ad hoc method. Analysis of object-oriented metrics indicated both simpler code and less variability among classes. Particularly striking was the order of magnitude difference between the average cohesion metric (LCOM) for the ad hoc and repeatable methods. Table 7 summarizes our findings.

Our analysis raises an interesting issue regarding the use of the repeatable methods. In general, we expect more variability in the ad hoc method, and we observed this. Programmers using the ad hoc method are redesigning the code from scratch, so their different design philosophies will be more apparent than they would be if using the repeatable method. Since the goal of the repeatable method is to provide the programmer with a suggested set of methods for each class, different programmers are more likely to produce similar code.

References

Abd-El-Hafiz, S. K. (2000). "Identifying Objects in Procedural Programs Using Clustering Neural Networks." Automated Software Engineering **7**(3): 239-261.

Achee, B. and D. Carver (1997). "Creating object-oriented designs from legacy Fortran code." Journal of Systems and Software **39**: 170-194.

Canfora, G., A. Cimitile, et al. (2001). "Decomposing legacy systems into objects: an eclectic approach." Information and Software Technology **43**(6): 401-412.

Chidamber, S. R., D. P. Darcy, et al. (1998). "Managerial use of metrics for object-oriented software: An exploratory analysis." IEEE Transactions of Software Engineering **24**(8).

Chidamber, S. R. and C. F. Kemerer (1994). "A metrics suite for object-oriented design." IEEE Transactions of Software Engineering **20**: 476-493.

Cimitile, G., A. D. Lucia, et al. (1999). "Identifying objects in legacy systems using design metrics." The Journal of Systems and Software **44**: 199-211.

Dunn, M. and J. Knight (1991). Software reuse in an industrial setting: A case study. 13th International Conference on Software Engineering**:** 329-338.

Fanta, R. and V. Rajlich (1998). Reengineering object-oriented code. International Conference on Software Maintenance.

Frakes, W. B., C. J. Fox, et al. (1991). Software engineering in the UNIX/C environment. Englewood Cliffs, Prentice Hall.

Frakes, W. B. and K. Kang (2005). "Software reuse research: Status and future." IEEE Transactions of Software Engineering **31**(7): 529-536.

Frakes, W. B., G. Kulczycki, et al. (2008). An empirical comparison of methods for reengineering procedural software systems to object-oriented systems. 10th International Conference on Software Reuse. H. Mei. Beijing, China, Springer**:** 376-389.

Frakes, W. B., G. Kulczycki, et al. (2006). Case study of a method for reengineering procedural systems into OO systems. 9th International Conference on Software Reuse. M. Morisio. Turin, Italy, Springer**:** 184-202.

Gamma, E., R. Helm, et al. (1994). Design patterns: Elements of reusable object-oriented software, Addison-Wesley.

Gui, J. (2003). "Software reuse through reengineering of legacy systems." Information and Software Technology.

Lanza, M. (2003). Object-oriented reverse engineering, University of Bern. **PhD**.

Liu, S. and N. Wilde (1990). Identifying objects in a conventional procedural language: An example of data design recovery. Conference of Software Maintenance. San Diego, CA, IEEE CS Press**:** 266-271.

Livadas, P. E. and T. Johnson (1994). "A new approach to finding objects in programs." Journal of Software Maintenance: Research and Practice **6**: 249-260.

McFall, D. and G. Sleith (1993). Reverse engineering structured code to an object oriented representation. 5th International Conference on Software Engineering and Knowledge Engineering. San Francisco, CA**:** 86-93.

Newcomb, P. and G. Kotik (1995). Reengineering procedural into OO systems. 1995 Working Conference on Reverse Engineering.

Pole, T. P. (1991). Pole method for C to C++ reengineering. W. B. Frakes.

Pressman, R. S. (2005). Software Engineering: A Practitioner's Approach, McGraw-Hill.

Siff, M. and T. Reps (1999). "Identifying modules via concept analysis." IEEE Transactions on Software Engineering **25**(6): 749-768.

Suryanarayanan, L. and W. B. Frakes (2003). Reengineering with reuse: A case study, Virginia Tech.

Valasareddi, R. and D. A. Carver (1998). Graph-based object identification process for procedural programs. 1998 Working Conference on Reverse Engineering.