Case Study of a Method for Reengineering Procedural Systems into OO Systems

William B. Frakes, Gregory Kulczycki, Charu Saxena

Computer Science Department, Virginia Tech, Falls Church, VA USA
frakes@cs.vt.edu, gregorywk@vt.edu

Abstract. This study evaluates a method for reengineering a procedural system to an object-oriented system. Relationships between functions in the procedural system were identified using various coupling metrics. While the coupling metrics used for analysis were helpful in identifying candidate objects, domain expert analysis of the candidate objects was required. The time taken at each step in the process was captured to help determine the effectiveness of the method. Overall the process was found to be effective for identifying objects.

Introduction

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 if it can be avoided. 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.

This case study extends the Pole method described in [1] with new metrics and uses it to identify potential reusable objects in the *ccount* metrics tool [3], which is written in C.

The steps in the method are described briefly along with the required metrics. The process and time taken for each step was captured and reported, and the data collected was used to determine the overall effectiveness of the method. The goal of this process is to identify reusable objects in the application domain. Once the code has been reengineered using these objects, traditional refactoring methods can be applied to further refine these objects and strengthen the design of the object-oriented code.

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 steps to be taken in the reengineering process 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. For clarity, we break its description into three sub-steps.
   (a) **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.

(b) **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 this study we used eight different coupling metrics and evaluated each one individually for its effectiveness in identifying objects.

(c) **Candidate objects are identified.** 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.

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.

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 [3], 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 [6, p. 306]. 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 [3], 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 [3]. 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 6 of this paper.

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 1. Functions used in the definitions of the eight coupling metrics.

The following subsections present eight different metrics—three invocation metrics, one shared parameter metric, and four shared variable metrics. Table 1 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.

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 [6]. 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

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

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

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

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 [3]. 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).

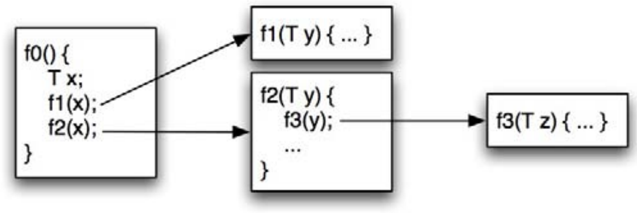

Figure 1. The functions all use a variable that can be traced to the same source.

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

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

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 2 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 $\mid N(f_1, f_{mid}) \mid > 0$ and $\mid N(f_2, f_{mid}) \mid > 0$
Indirect Invocation Metric	$N_{ind}(f_1, f_2)$	$N(f_1, f_{mid}) + N(f_{mid}, f_2)$ where $\mid N(f_1, f_{mid}) \mid > 0$ and $\mid N(f_2, f_{mid}) \mid > 0$
Shared Parameter Metric	$P(f_1, f_2)$	$\mid params(f_1) \cap params(f_2) \mid$
Shared Variable Metric	$V(f_1, f_2)$	$\mid \{ v \mid history(v, f_1) \cap history(v, f_2) \mid \neq \varnothing \} \mid$

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

Example: 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 [3]. 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. We captured the process and time taken at each step. The authors acted as the domain experts.

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.

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.

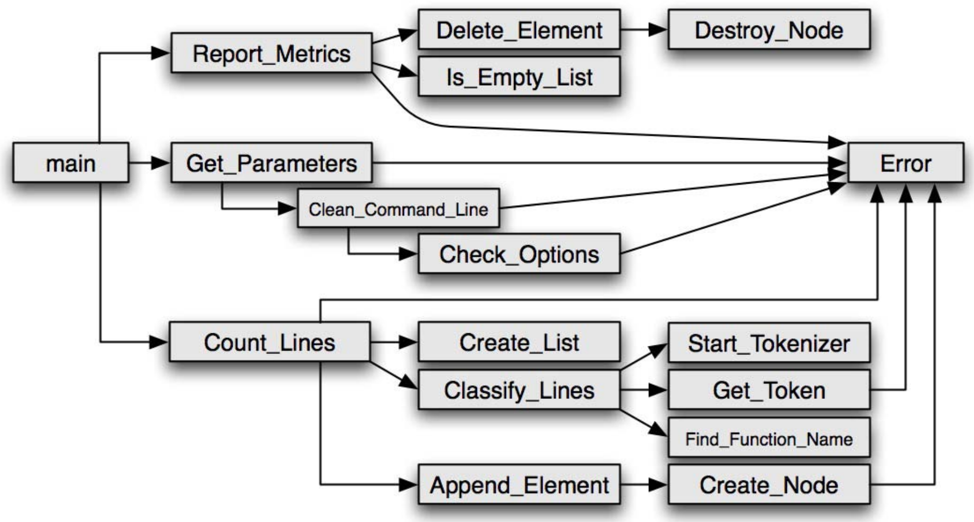

Figure 2. Call graph for ccount tool.

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 function 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.

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 time taken for this step was 48 hours.

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 functions 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 3. Summary data for functions Check_Options and Clean_Command_Line.

An example of the information collected in this step is given in Table 3. 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.

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 4 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 4. 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 5 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 5. 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.

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.

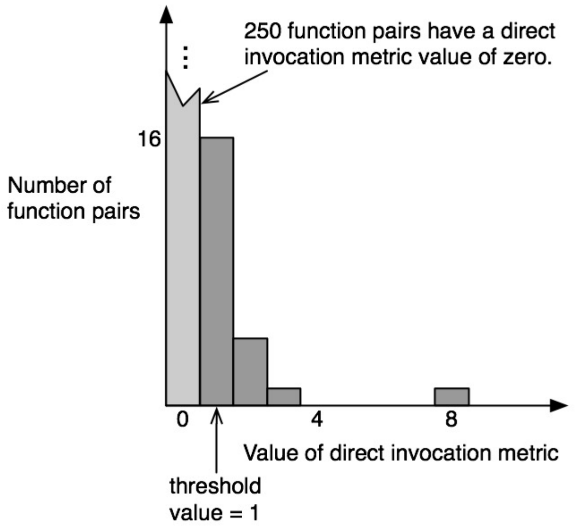

Figure 3. Distribution of values for the direct invocation metric.

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 6. Candidate objects for each of the coupling metrics.

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 given below. 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.

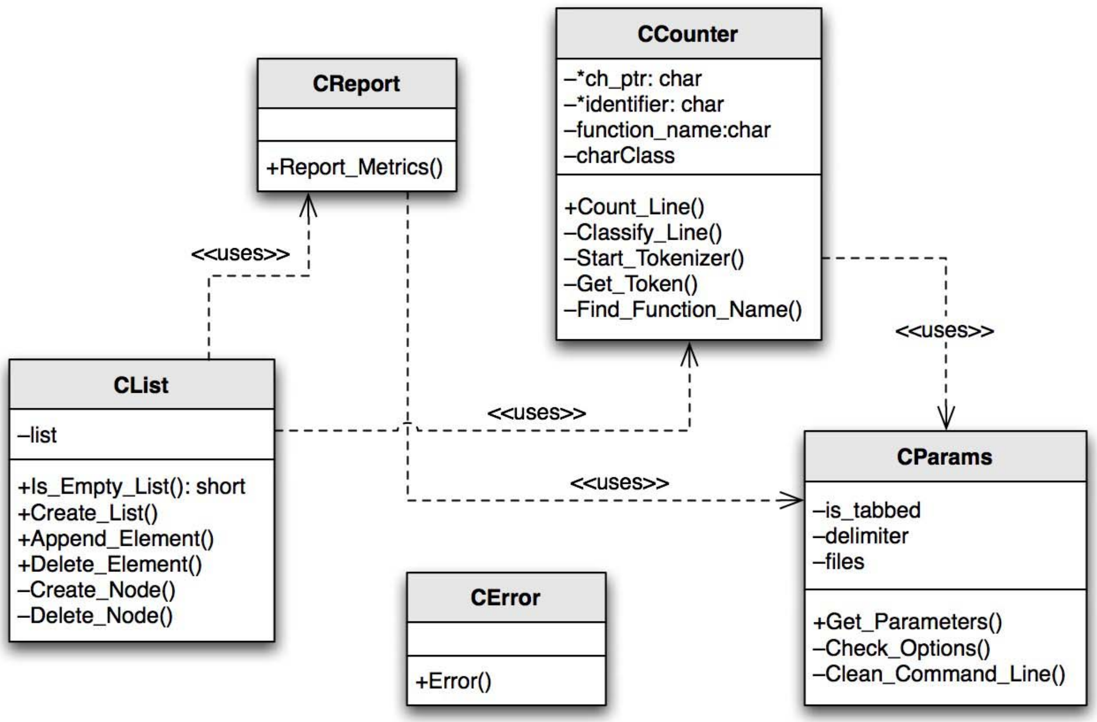

Figure 4. Class diagram for object-oriented ccount application.

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()
```

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 [3] 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 and it passed all the test cases.

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

Process variables captured

The times taken for each step are shown in Table 7. 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 7. 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

Conclusion and future work

Coupling metrics provide a good starting point for identifying objects, but the metrics we used in this study had limitations. For example, they were not able to completely identify the list data type in ccount. Hence domain expert analysis is an important step in the process—it is necessary for finalizing the optimal objects from the candidate objects identified from the coupling metrics.

The largest amount of time spent in the process was in determining the coupling metrics. The direct invocation metric and shared parameters metric were found to provide reasonable objects very close to the objects finalized by the domain expert. The time taken to determine these two metrics was considerably less than the time it took to determine the shared variable metrics, since they do not require the collection of the detailed summary data shown in Table 3. Therefore, the process could be accelerated if only the direct invocation and shared parameter metrics were taken into consideration. Using a tool like the CIA (C Information Abstraction System) [4] would also help in speeding up the process.

Some things to consider for future case studies would be using the direct invocation and shared parameter metrics in conjunction to arrive at candidate objects. When more than one metric is used, one could either sum the metrics or assign a weight to each metric, indicating that one form of coupling is considered more relevant [5]. For example, we might calculate the combined direct invocation and shared parameters metric as 2 * direct invocation metric + 3 * shared parameters metric. In this case the higher weight attached to the shared parameters metric indicates a data definition coupling is more relevant than routine call coupling.

This case study presents a good first step in determining how to reengineer a legacy procedural system into an object-oriented system. The methodology we examined was found to be helpful in identifying objects. It can also serve as a framework that is usable with coupling metrics other than those presented here.

References

[1] Thomas P. Pole, *Pole Method for C to C++ Reengineering*. Personal Communication.

[2] William B. Frakes and Thomas P. Pole, "An Empirical Study of Representation Methods for Reusable Software Components," *IEEE Transactions on Software Engineering*. Vol. 20, No. 8, August 1990, pages 617–630.

[3] William B. Frakes, Christopher J. Fox, and Brian A. Nejmeh, *Software Engineering in the UNIX/C Environment*. Prentice Hall, Englewood Cliffs, 1991.

[4] Yin-Farn Chen, Michael Y. Nishimoto, and C. V. Ramamoorthy, “The C Information Abstraction System,” *IEEE Transactions on Software Engineering*. Vol. 16, No. 3, March 1990, pages 325–334.

[5] Michael Whitney, Kostos Kontogiannis, J. Howard Johnson, Morris Bernstein, Brian Corrie, Ettore Merlo, James McDaniel, Renato De Mori, Hausi Muller, John Mylopoulos, Martin Stanley, Scott Tilley, and Kenny Wong, “Using Integrated Toolset for Program Understanding,” in *Proceedings of the CAS Conference (CASCON 95)*. pages 262–274.

[6] Roger S. Pressman, *Software Engineering: A Practitioner’s Approach*. Boston: McGraw-Hill, 2005.