# Re-engineering with reuse: a case study

Lakshmanan Suryanarayanan
lsuryana@vt.edu

William B. Frakes
frakes@cs.vt.edu

Computer Science Department
Virginia Tech

**Abstract**

This paper describes a case study in reuse and reengineering. A C based metrics system was re-engineered to C++ using standard reusable components and a design pattern.

# 1 Introduction

This paper describes a case study in reuse and reengineering. The system used in the study is ccount, a C based metrics tool. This paper is structured as follows.

We first discuss the various options available when converting an existing application from C to C++. Then we examine the design decisions and the rationale behind them and discuss more on the selected option. Next, we discuss the mapping of the various files and functions in C to classes, objects, and methods. This mapping table provides an easy way to understand how the C++ code was organized based on what was known.

Then we describe the use of the Standard Template Library, some of the powerful tools that it provides, and a specific design pattern, Singleton, in our re-engineering effort. We then show the UML Class diagrams and objects relationships and the interaction diagrams that graphically represent the physical and logical relationships between the various components of the software. The section concludes by showing the output produced from the C++ version along with the output from the ANSI-C version and discusses the two minor areas that are missing in the C++ version.

# 2 Related work

The history of programming and software engineering demonstrates the continual evolution towards larger grained programming constructs, and more human focused languages [Frakes, 2000]. 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++, for example, as opportunities for better reuse.

[Dunn and Knight , 91] have reported on such a project for the Sperry Marine corporation. They used four existing C based systems in the domain of shipboard navigation as the basis for a re-engineering effort that yielded reusable objects in four subdomains.

[Pole, 92] has reported similar work reported for the smart card domain. In this case from the C code are created a stoplist of functions, a dependency list (what an item uses or depends on) and a context list (what uses or depends on an item), Then, by analysis of these lists, a reasonable set of objects and their functions are determined. These objects and their attributes and functions

determine the classes to be designed and coded. The leftover functions that do not belong to any object can then be packaged as a utility class or utility object.

# 3 ccount

Ccount is a metrics tool for the C language that calculates the number of commentary and non-commentary sources lines in a C program, and their ratio. It was developed as a simple programming quality tool, and was used as an example program in [Frakes, Fox, and Nejmeh, 91]. The original version of ccount was written in K&R C on AT&T Unix. An ANSI-C version of ccount has recently been created and this version is the one used for re-engineering to C++.

# 4 C to C++ conversion

There is more than one way to convert a software product from one language to another. A very simple approach would be to take the modules or functions in the existing language and wrap them around in modules or functions in the other. This ensures that the resulting product is in the target language, while not changing the functionality and the results by much. This is not elegant 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 backward compatible to C and supports everything that is C, a very simple conversion would have been to change the extensions on the files to .cpp and change printf's to cout's and be done. But, the resulting product is still C in C++ clothing. The second option would be to use a C to C++ re-engineering effort such as that described by Pole.

The third option is to start from first principles and look at the problem statement, identify the objects that stand out in the problem and design and develop the product from the ground up by defining attributes and functions for the various objects and creating classes for these objects. This option produces the most elegant code with most utilization of the features of the destination language. 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. The added expense that comes with it is that of fresh complete software development life cycle and this means the need for a much longer time to complete the conversion.

## 4.1 Design Decisions

The option we chose is a combination of the second and third and the mix can be fairly categorized as an even distribution on choices. We chose to start from first principles in identifying objects, but once the objects are identified the existing functions were re-mapped into methods that were appropriate for these objects. By doing so, we eliminated some functions, added some new ones and replaced existing ones with those from the standard C++ libraries or with simpler one's that took advantage of the progress made in software platforms and portability of code that comes with using ANSI standards.

Since, we decided to use the existing functionality and not re-write from the ground up, we were left with some functions that did not belong to any of the objects we identified and some of these needed to be global since they maintained state information within the function between calls. At this point, we took the approach in the second option above of packaging these functions into a

utility class. Moreover, since this utility class had global functions that maintained state, we needed to ensure that there was only one instance of such a class.

This could have been achieved by using a utility class consisting of all static member data and member functions, thereby not requiring the creation of the object and being able to use class methods to get the functionality of a single object. But that is not a good design, since it leaves all data accessible to the external methods and thereby voiding the encapsulation principle of C++.

Hence, the solution to this was to use the Singleton design pattern to achieve a single instance of the object. More about the Singleton design pattern is below. In addition, due to time constraints, we left the ad-hoc parsing algorithm used for the classification of a line using token parsing the same as in the C version. This is an area of improvement and can be taken up as a future work.

So, from the statement of the problem and first principles, we can identify 3 distinct objects,

- File --- an object that needs to be analyzed and CSL, NCSL and ratio of CSL/NCSL determined. There are 1 – N files that need to be analyzed at any invocation.
- Func --- an object that is the lowest granularity that needs to be analyzed and the same above metrics reported. Every function belongs to 1 File and a file can contain 1 – N functions. (Note that code external to a C function, is treated as belonging to the function "external").
- Line --- An object that needs to be classified as extern or belonging to a function and as a comment, non-comment, neither or both. Every line belongs to only one function and a function has 1 – N lines.

All of the above objects have certain attributes and we find a very good match of C functions into 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, which are either external to these objects or are not confined to one object. In addition, the sturdier, more generic and more optimal <list> container from the Standard Template Library for C++ can safely replace the code for linked-list generation, maintenance and deletion coded in C.

## 4.2 Mapping between C and C++

The section presents a pictorial mapping showing how the functions and modules in C mapped into objects and methods in C++. The following shows the mapping between C Files and Functions to C++ Classes and Methods. As highlighted in the picture, the boxes in grey were eliminated, the boxes in blue were optimized, the boxes in green were added and the rest either changed very little or did not change.

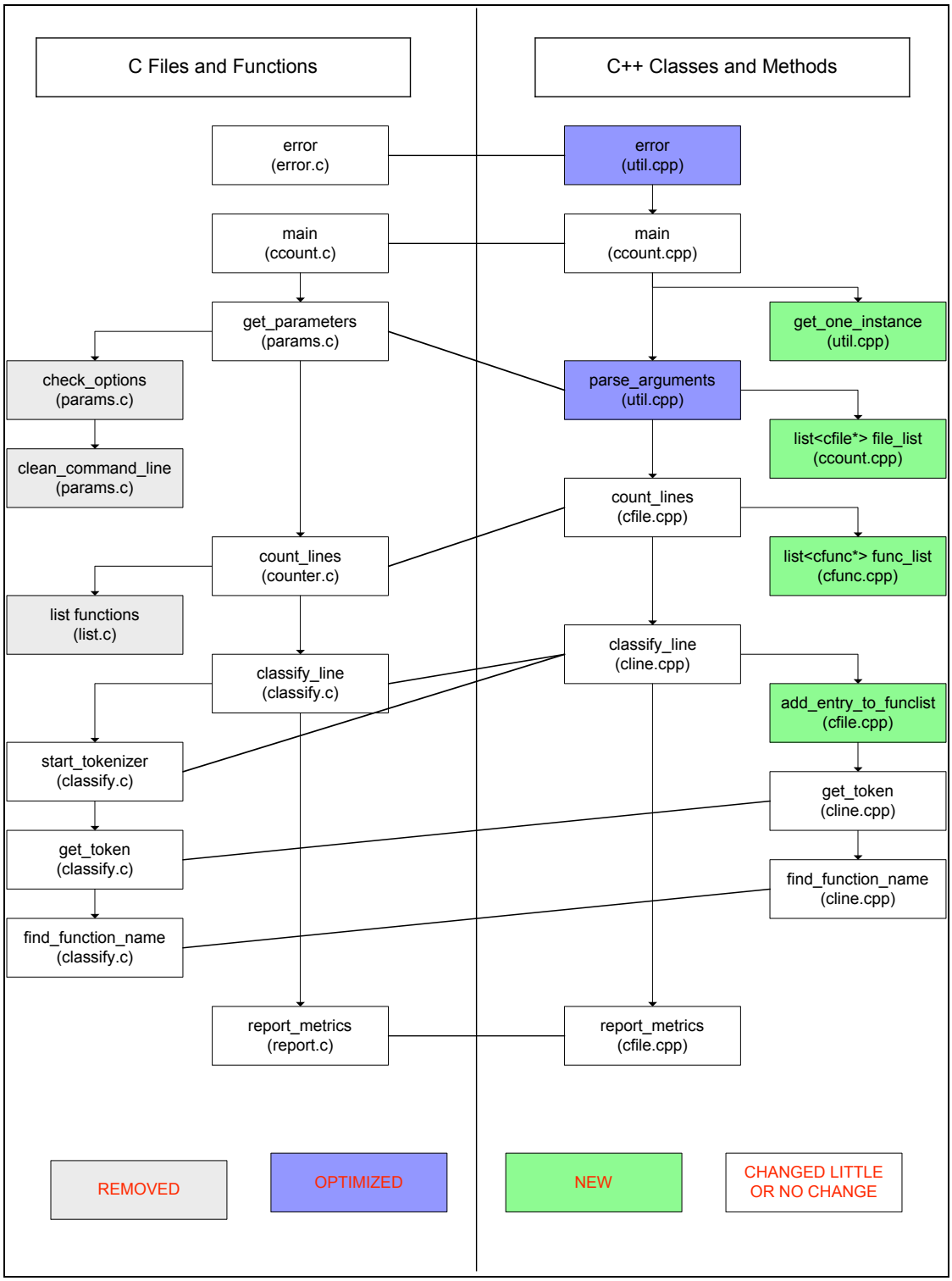


**Figure 1: Mapping between C and C++**

## 4.3 Using the Standard Template Library and Singleton Design Pattern

This section discusses two important concepts of C++ and object oriented programming that we used in the C++ implementation and provides examples from our implementation to illustrate their use.

The Standard Template Library (STL) is a general-purpose C++ library of algorithms and data structures. It is based on the concept of generic programming, and is now part of the standard ANSI C++ library. C++ templates form the basis of the implementation of the STL. The STL has good generality, a key feature of promoting and facilitating reuse of data structures and algorithms.

Some of the commonly used data structures and algorithms are

- The string class – A flexible block of memory that provides the functionality of a char * string without needing to allocate or deallocate memory. The string automatically grows and reduces as needed.
- The vector container template – A vector container is similar to an Array in that it can store 1 or more objects of the same type and they can be accessed and operated on individually. The key difference is in the fact that the elements in the vector can be of different data types.
- The list container template – This implements a classic linked list kind of structure and it is doubly linked. The elements cannot be accessed directly but need to walked starting at the head or the tail of the list.
- The iterator classes – Iterators are objects that represent positions of elements within a container, just like index into C++ arrays. They enable you to access and iterate through a list or vector or any such container and support operations such as increment and decrement.
- The find algorithm – Finds an element in a container list or vector given the value to be searched.
- The sort algorithm – sorts the elements in a container based on the order specified.

In our implementation we make use of string class, the list container and iterators for lists. Examples of each are shown in Figure 12 below.

```
  private:
      string filename;              /* name of the file */
      int csl;                      /* count of commentary source lines */
      int ncsl;                     /* count of non-commentary source lines */
      list<cfunc*> function_list;   /* list of functions */
…
…
     list<cfunc*>::iterator iter;
     for (iter = this->function_list.begin();
          iter != this->function_list.end();
          iter++) {
         delete (*iter);
     }
```

**Figure 2: Illustration of string, list container and iterator from STL**

As mentioned above, it is generally a bad practice to use global variables and global methods and in the case of C++, global objects that can be used anywhere in the program. From the above discussion, we need the ability to create a single object of the "util" class for the lifetime of the execution of the program. This is best achieved by implementing the "util" class as a Singleton design pattern. The Singleton Pattern is a kind of Creational Pattern, which deals with the best ways to create objects. The Singleton Design pattern is used, where only one instance of an object is needed throughout the lifetime of an application. The Singleton class is instantiated at the time of first access and same instance is used thereafter till the application quits. The UML representation of a Singleton Pattern is shown below. This is the pattern that is used in the creation of an object of class "util".

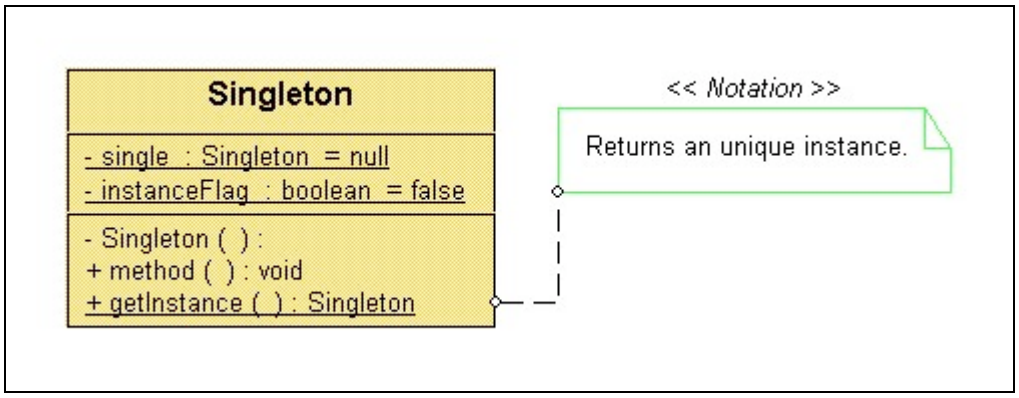


**Figure 3: UML representation of Singleton Pattern**

## 4.4 UML Diagrams (Class and Sequence)

This section pictorially represents the call sequence diagram that shows how classes are instantiated and used during the process of analysis the code for the various files.

Figure 14 on the following page shows the <uses> relationship between the various classes as described below. Figure 15 shows the sequence diagram depicting the order of activation and execution of the object's code.

- A single object of type Class ccount creates a list of Class cfile objects and executes the count_lines method followed by report_metrics method for each cfile object. In addition, the ccount object also uses a single instance of the util Class and calls the parse_arguments and error methods.
- A cfile object consists of a list of function objects of class cfunc and its own private variables csl and ncsl to keep track of total counts for the file. A cfile object's count_lines method creates an instance of a cline object, classifies the line, and then uses classification information to increment appropriate counts in the appropriate cfunc object.
- Once the classification of all lines in a file is complete and count_lines returns, the ccount object then calls the report_metrics method of the corresponding cfile object, which in turn calls the get_csl and get_ncsl methods for each of its cfunc, function objects. This information is then printed to standard output.

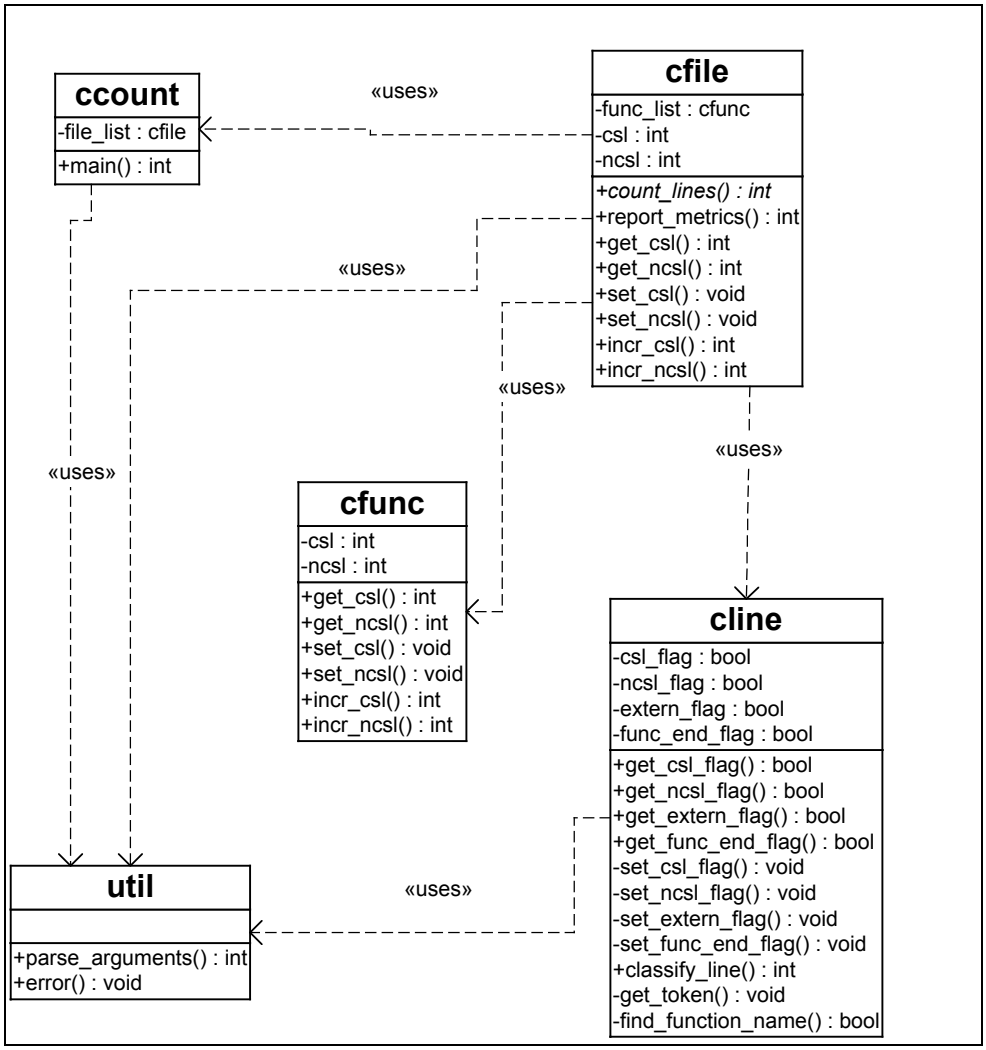


**Figure 4: Uses relationship between the various C++ classes**

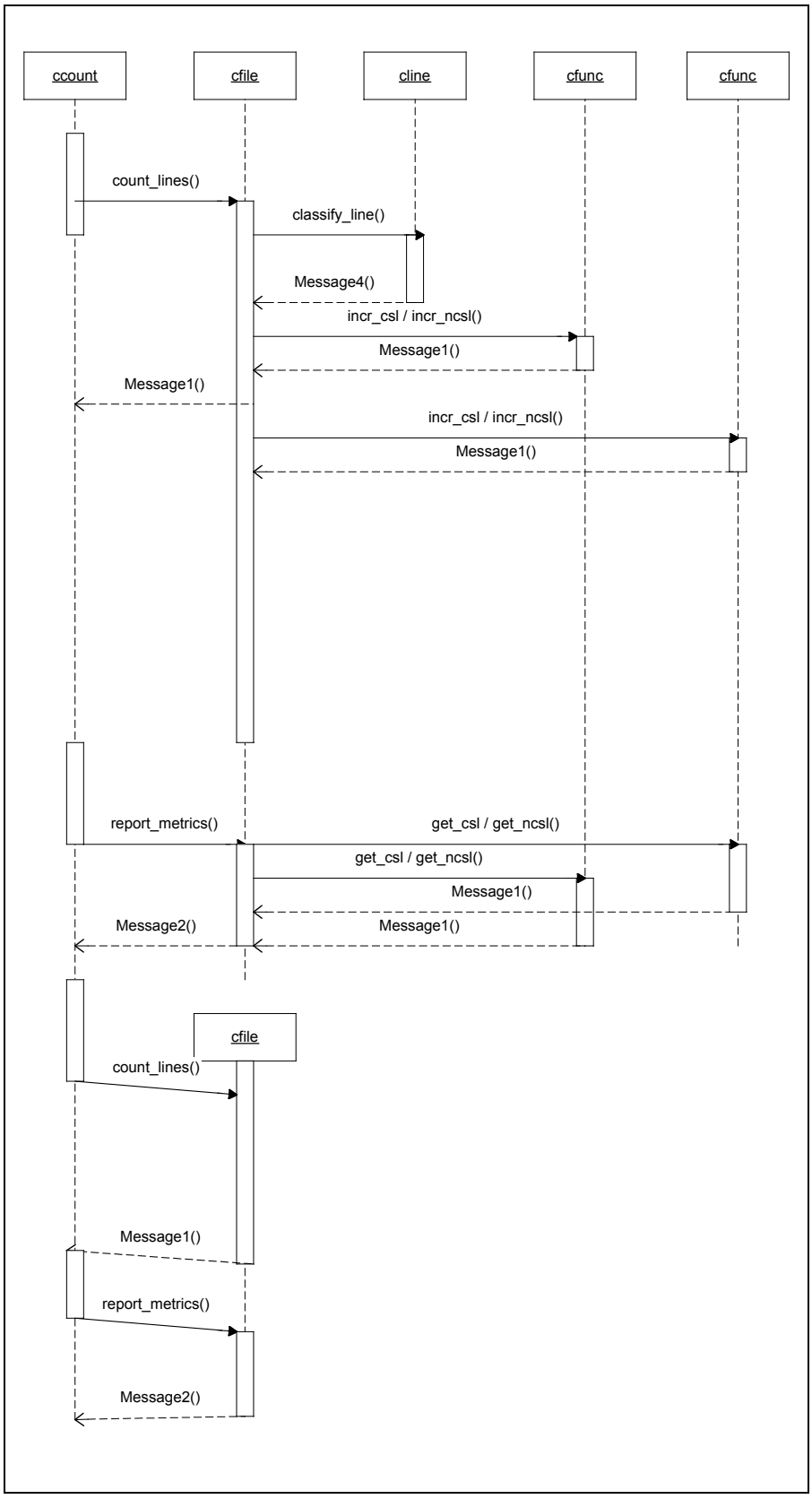


**Figure 5: Sequence diagram for ccount in C++**

# 5 Summary

This paper describes a case study in reuse and reengineering. A C based metrics system was re-engineered to C++ using standard reusable components and a design pattern. This demonstrates the use and benefits of existing reusable code, such as STL. Based on earlier work on C to C++ reengineering, we argued for using the first principle approach to make the most of of new programming language features. This approach supports the creation of reusable classes that can be easily extended to include other programming languages and other metrics.

# 6 REFERENCES

1. Frakes, W. B., Fox, C. J., & Nejmeh, B. A. (1991). *Software Engineering in the UNIX/C Environment*. Englewood Cliffs, NJ: Prentice-Hall.
2. M. Dunn and J. Knight, "Software Reuse in an Industrial Setting: A Case Study," Proceedings 13th International Conference on Software Engineering, 1991, pp. 329-338.
3. Frakes, W. B. (2000). Practical Software Reuse. In 3rd IEEE Symposium on Application-Specific Systems and Software Engineering Technology (ASSET'00), (pp. 115-116). Richardson Texas: IEEE Computer Society.
4. Crash Course on STL, http://www.geocities.com/ResearchTriangle/Node/2005/stl.htm
5. Using STL Containers, Bill Whitney, http://www.bridgespublishing.com/articles/issues/9910/Using_STL_containers.htm
6. Pole, T. (1992). Recovering the Implicit Reusable Objects from a Non-Object Oriented Implementation. In 3rd Reverse Engineering Forum, . Northeastern University Burlington, Massachusetts, USA:
7. Unix Power Tools, Third Edition, October 2002, Shelley Powers, Jerry Peek, Tim O'Reilly, Mike Loukides, O'Reilly and Associates
8. Design Patterns, Elements of Reusable Software; Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides; Addison-Wesley; 1995