

Restructuring legacy C code into C++¹

Richard Fanta, Václav Rajlich
Department of Computer Science
Wayne State University
Detroit, MI 48202, USA
rajlich@cs.wayne.edu

Abstract

In this paper, we describe restructuring of C code into new C++ classes. Such restructuring is done to facilitate both software reuse and software evolution. The restructuring is accomplished by restructuring tools and scenarios. We discuss the transformation tool-set and the design of the individual tools. The approach is demonstrated on a case study. We also discuss how this tool-set could be enhanced in the future.

1. Introduction

Object-oriented programs are composed of classes that group together data and operations belonging to a specific domain concept. Such programs facilitate both software reuse and software evolution. If the domain concepts are implemented as classes, they can be reused in future projects. Studies [12] have shown that when programmers reuse already implemented classes, object-oriented programming becomes significantly more effective than the traditional imperative programming. However, large legacy systems are mostly implemented in procedural languages and the domain concepts are de-localized throughout the code. Thus, even though there is valuable experience recorded in the legacy code, it must be first extracted in order to make it reusable. Research [4,9] suggests that classes can be identified in legacy procedural code and a case study [4] showed high level of reuse after reusable components were encapsulated as C++ classes.

A properly designed object-oriented architecture also facilitates software evolution. A class implementing a domain concept can easily replace another class with the same functionality but different implementation [5,16]. For example, when a program is ported to a different platform and the GUI interface must change, it is easier to

replace it if it consists of well-defined classes, rather than if it is de-localized throughout the code.

In order to introduce these advantages into a legacy code, it has to be restructured first. Procedural code is transformed into object-oriented code in three phases:

- search for classes
- restructuring into classes (encapsulation)
- improving the classes for better style or structure, including inheritance

Several research groups [5,17,7,13,18] have investigated methods for discovering classes in programs written in procedural languages, like C, Cobol, etc. This paper builds on this effort and presents the next step, i.e. restructuring into classes. The third issue was investigated in [14,15] and is dealt with briefly in the conclusions section of this paper.

Class encapsulation requires substantial effort because variables and code fragments belonging to a domain concept are often scattered throughout the code. As they are brought together into one class, ripple effects impact other parts of the code. The ripple effects may be hard to trace because of complex dependencies among software components. For this reason, encapsulation with only text editors is error prone and laborious. Compared to that, automated restructuring tools efficiently analyze the code and automatically deal with the ripple effects. They also ensure that all preconditions of the transformation hold.

In this paper we present a tool-set that we used to encapsulate new classes in procedural C code. We implemented a small and efficient tool-set that performs automatic transformations. For more complicated restructuring tasks, we used scenarios that combine the tools with human intervention.

In order to determine which tools need to be implemented, we adopted the following approach:

- search for potential C++ classes in a medium size C program (NCSA Mosaic)

¹ This work was partially supported by NSF grant #CCR-9803876

- specify class encapsulation scenarios
- implement transformation tool prototypes
- perform a case study

The paper is structured into 7 sections. The next section presents an overview of the NCSA Mosaic that we used in our case study. This section also contains a list of implemented tools. Section 3 describes transformation tools in detail. Then we present scenarios for new class encapsulation (section 4) and results of the case study (section 5). Section 6 reviews related work. Conclusions are summarized in section 7.

2. Overview of Mosaic code

To develop restructuring scenarios and to test the developed tools, we used publicly available source code of Mosaic browser, version 2.5 [19]. We selected Mosaic because it is a well-known application and it deals with several different application domains:

- GUI
- HTML processing and display
- network connections and protocols

In addition to these domains, Mosaic code also includes several standard data structures (e.g. lists, hash tables, caches).

The source code consists of 95,000 lines of C code. The major part of the code (about 70,000 lines) is in support libraries that implement networking protocols and html parsing and display. The rest of the source code (about 25,000 lines) is the browser implementation. The code is structured into approximately 200 header and source files. During the code comprehension we concentrated on two types of potential classes [9]:

- Classes that have their data already encapsulated as a C structure.
- Classes that have their data implemented as group of individual variables.

Code comprehension was done manually without automatic support. We discovered about 80 classes and identified their code. For the actual encapsulation, we selected a subsystem of about 3,000 lines. Results of the case study are presented in section 5.

As part of this work, we established requirements for the tool-set that would be used in encapsulation. The core of the tool-set is function encapsulation and function insertion. The function encapsulation tool encapsulates a sequence of consecutive statements or an expression into a new function and replaces the original code fragment with an appropriate function call. The function insertion tool inserts a standalone function into a target class. The original function must have the target class as one of its parameters. These tools were already used in our previous projects and covered in our previous paper [5] where

more details on these two transformations can be found. New transformations added to the tool-set are:

- variable insertion
- make access to non-local variable explicit /implicit
- add a formal parameter to a function
- change access specifier of a class member

All tools on this list support transformations that cause ripple effects in other parts of the code. For example, if a variable is inserted into a class, all references to that variable must be changed.

On the other hand, some transformations do not cause any ripple effects and in that case we choose not to support them with automatic tools. Examples are "creation of new class", "creation of new class instance", and "change structure into class". A standard editor can easily handle these transformations. The transformations are described in detail in the next section.

3. Transformation tool-set

This section describes the tools we selected for implementation. For every tool we give both the functionality and limitations.

Variable insertion

This transformation inserts a single variable into a class as either a public regular or a public static member. The transformation takes two inputs: the variable that the programmer intends to insert and the class or class instance where the variable will be inserted. If a class is specified as the target, the variable will always become a public static member. If a class instance is specified as the target, the variable will be inserted as either a static or a regular member.

To make that decision, we must consider the expected lifetime of the variable and of the target class instance. A variable can be inserted into a class instance as a regular member only if the target class instance is not destroyed during existence of the selected variable. For example, global variables and local variables can be inserted into global class instances as regular members. If this rule does not hold, the variables can be inserted only as static members.

In order to simplify the tool, we decided to restrict its functionality in a way that does not affect common use of the tool. The following restrictions are checked before the transformation is performed:

- A variable can be inserted into array of classes only as a static member
- A variable cannot be inserted if the class contains a member with the same name.
- A global variable can be inserted into a target class only if its initialization (assignment or constructor invocation) depends only on constants.

```

class A{
public:
char* str;
A(char* ns){str=ns;}
};

A a("initial string");

void func(){
int len=0;

len=strlen(a.str);
if(len>MAX)...
...
}

```

Figure 1.

```

class A{
public:
char* str;
A(char* ns){str=ns;}
int len;
};

A a("initial string");

void func(){
a.len=0;

a.len=strlen(a.str);
if(a.len>MAX)...
...
}

```

Figure 2.

```

class A{
public:
char* str;
A(char* ns){str=ns;}
};

int len;

void func(){
len=0;

A a("initial string");
len=strlen(a.str);
if(len>MAX)...
...
}

```

Figure 3.

```

class A{
public:
char* str;
A(char* ns){str=ns;}
static int len;
};

int A::len;

void func(){
A::len=0;

A a("initial string");
A::len=strlen(a.str);
if(A::len>MAX)...
...
}

```

Figure 4.

- A variable cannot be inserted if a pointer to the variable is involved in any pointer arithmetic other than simple assignment.
- If a variable is inserted into class instance, the tool converts it into a regular public member only if it can safely determine that the instance is not destroyed in the scope of the selected variable. In all other cases variables can be inserted only as static members.
- If the previous condition holds, the variable is inserted as a regular public member by default. However the user can override this default and insert the variable as static.

The tool makes the following changes in the code:

- For regular members, the declaration of the selected variable is moved into the class specification (see figures 1,2).
- For static members, the declaration is copied into the class specification and specified as `static`. At the same time, the original declaration is moved to the global scope and the variable name is prefixed with the class name followed by the `::` operator (see figures 3 and 4).
- If the original variable was declared and initialized in a local scope, the initialization is converted to either an assignment statement or an explicit constructor invocation in its original place.
- If the original variable was declared and initialized in global scope (only constant initializations are considered), the initialization is moved into the member initialization list of the target class's constructor.
- All references of the original variable are replaced by the appropriate expression with `."` or `"->"` or `"::"` operators (see figures 2, 4).
- If the variable is of a user-defined type, the definition of the type is moved to a separate file and this file is included in both the original file and the file containing the target class definition.

Make access to non-local variable explicit/implicit

The transformation "make access explicit" converts implicit access to a non-local variable into explicit access through a new parameter. The non-local variable can be either a global variable or a member variable of the same class as the target function. In figure 5, global variable `a` is accessed implicitly by function `func()`. The result of the transformation is shown in figure 6. The purpose of this transformation is to prepare the code for the function insertion tool, which can insert a function only into a class that is passed as one of the function parameters [5]. The transformation performs the following changes:

1. Add a new parameter to the function. If the original variable is involved in an "address of" (`&`) operator, the type of the new parameter will be "pointer to the variable type". In all other cases the parameter type will be "reference to the variable type".
2. To every function call, add the "implicit" non-local variable as actual parameter.
3. Every occurrence of the non-local variable in the function body is replaced with the new parameter.

The inverse tool "make access implicit" converts a function parameter into implicit access. The example in figures 8, 9, and 10 shows how the tool supports function insertion. In figure 8, `insta`, an instance of class `A`, is

```

class A{
public:
int ai
...};

A a;

void func(){
...
a.ai=4; //access a
...
}

```

Figure 5.

```

class A{
public:
int ai
...};

A a;

void func(A& pa){
...
pa.ai=4; //access a
...
}

```

Figure 6.

passed together with its member `insta.ai` to every call of the function `func()`. After the function `func()` is inserted into the class `A` using the function insertion tool (figure 9), the parameter `i` is made implicit (figure 10).

This transformation performs the following changes:

1. Replaces all occurrences of the parameter with the implicit variable in the function body
2. Removes the parameter from the formal parameter list
3. Removes the parameter from the actual parameter list in every function call.

To guarantee that the behavior of the program will be preserved, the transformation can be performed only if the following conditions hold:

- This transformation can be done if and only if the very same variable is passed to the function as an actual parameter in every function call.
- Access to global variables or members can be made implicit only if they are passed by reference or pointer or if they do not change their value in the function body.

Add parameter

This transformation adds a new parameter to a function. It is designed to support function insertion. In example in figures 7 and 8, a new parameter of type `A&` is added to the function `func()`. In figures 9 and 10 the function `func()` is inserted into the class `A` using function insertion tool and make implicit tool. To add a new parameter the following changes are performed:

1. Add the new parameter to the function's formal parameter list
2. For every function call specify the instance which will be passed in the place of the new parameter
3. For every function invocation pass the instance in place of the new parameter

When new classes are created using the scenarios and transformation tools, they do not comply with accepted views on data and code encapsulation. The following

```
class A{
public:
    int ai;
...};

void func(int& i){
...
i=4;
...}

//code fragment calls func()
...
A insta;
func(insta.ai);
....
//code fragment ends
```

Figure 7.

```
class A{
public:
    int ai;
...};

void func(A& a,int& i){
...
i=4;
...}

//code fragment calls func()
...
A insta;
func(insta, insta.ai);
....
//code fragment ends
```

Figure 8.

```
class A{
public:
    int ai;
    void func(int&);
...};

void A::func(int& i){
...
i=4;
...}

//code fragment calls func()
...
A insta;
insta.func(insta.ai);
....
//code fragment ends
```

Figure 9.

```
class A{
public:
    int ai;
    void func();
...};

void A::func(){
...
ai=4;
...}

//code fragment calls func()
...
A insta;
insta.func();
....
//code fragment ends
```

Figure 10.

transformation is designed to improve data encapsulation in the classes

Change access specifier of a class member

Using this tool, the programmer can change the access specifier of a class member. However, this transformation can be applied only if it does not make the code inconsistent. For example, `public` member cannot be changed to `protected` member if it is accessed in code outside of its class. The tool analyzes the code outside the class to determine whether the change can be made.

Implementation issues

For the transformation implementation we used GEN++ [3] to analyze the source code and C++ to perform actual changes to the code. GEN++ is a programmable code analyzer based on AT&T's Cfront. It parses the code and stores the result in an abstract semantic graph. Programmer can write queries analyzing the graph in LISP-like scripting language. Every transformation is composed from three parts:

- restriction analyzer (implemented in GEN++)
- transformation analyzer (implemented in GEN++)
- code transformer (implemented in C++)

The restriction analyzer scans the code to make sure all required preconditions for a transformation hold. The transformation analyzer locates all places of code that must be changed. It compiles an output file that specifies all places that must be changed and what change is required. The code transformer uses this file to perform the transformation and also applies all necessary changes to compensate for ripple effects of a transformation. There exist several different versions of C, like K&R, ANSI, etc. Current implementation of the prototype tools requires that GEN++ parser can successfully parse the input C or C++ source code. Additional implementation issues were discussed in [5].

4. Restructuring scenarios

Restructuring scenarios of this section involve both the tools and actions of the human user. The tools are described in the previous section and in [5]. The process of creating new classes in C code is divided into three phases:

- create data-only classes
- remove class clones
- add member functions

However, all three phases are not used in every case. Some future classes may already be in the form of C structures. Because the structures are similar to classes in C++, only the last phase is employed.

To create data-only classes out of individual variables, we use the following scenario:

1. Create new empty class
2. Create new class instance
3. Insert variables into the new class

In the first step new empty class specification is created using the standard editor. The class specification contains only empty constructor (without parameters) where initializations of inserted variables will be transferred in step 3. In the next step, a new instance of the class is created by declaration. These two steps are performed by the standard editor since they do not affect the rest of the code. In the third step, variables are

```
static mo_window *winlist = NULL;
static int wincount = 0;

mo_status mo_add_window_to_list (mo_window *win)
{
    wincount++;

    if (winlist == NULL)
    {
        win->next = NULL;
        winlist = win;
    }

    else
    {
        win->next = winlist;
        winlist = win;
    }

    return mo_succeed;
}

mo_window *mo_next_window (mo_window *win)
mo_window *mo_fetch_window_by_id (int id)
mo_status mo_remove_window_from_list (mo_window *win)
```

Figure 11.

inserted into the new class using the variable insertion tool.

Some domain concepts in the original code may be independently implemented in several locations. After encapsulation, they may produce class clones, i.e. identical or near identical classes. Therefore it is necessary to perform clone removal in this phase, before proceeding to the third phase. The clone removal is performed using the same tools and scenarios summarized in [6]. During the clone removal, a single class replaces all clone classes.

In the next and last phase, programmer adds relevant code to data-only classes. In our case study we encountered four different situations:

1. A function that contains the target class in its parameter list is inserted by function insertion tool
2. A function that accesses the target class through a global instance is dealt with in the following way: “make explicit” transformation to convert access to the global instance to parameter, as in figures 5, 6 and after that the function insertion tool inserts the function into the class
3. A function that accesses only individual members of the target class through parameters:
 - add new parameter of the target class type
 - insert the function into the target class
 - convert original parameters that take class members to implicit member access, as in figures 7, 8, 9, 10

```
class Wlist {
public:
    mo_window *winlist;
    int wincount;
    Wlist(mo_window* wl=NULL, int wc=0)
        :winlist(wl),wincount(wc){
};

Wlist wlist(NULL,0);

mo_status mo_add_window_to_list (mo_window *win){
    wlist.wincount++;
    if (wlist.winlist == NULL)
    {
        win->next = NULL;
        wlist.winlist = win;
    }
    else
    {
        win->next = wlist.winlist;
        wlist.winlist = win;
    }

    return mo_succeed;
}

mo_window *mo_next_window (mo_window *win)
mo_window *mo_fetch_window_by_id (int id)
mo_status mo_remove_window_from_list (mo_window *win)
```

Figure 12.

- Code fragments that perform an operation on a target class are first encapsulated into a new function and then inserted as in case 1 or 2.

5. Results of the case study

In our case study we selected a 3000 line subsystem of NCSA Mosaic browser. The following classes were encapsulated:

The class `color_rec` represents a color. Its data members describe the color as a triple of integers – one for each color component (red, green, and blue). The class `c_box_rec` is a linked list of color records. It contains a pointer to the first record, limits on red, green and blue values, and a counter that indicates number of records stored. The class `color_hash` implements a hash table of color records. The class `color_box_list` represents a list of `c_box_rec` classes. The list has a fixed length (256) and is implemented using an array. The class also contains several data members necessary for efficient function. Class `url` represents the URL address, a domain concept typical for www browsers. It has only one data member (a string representing the address) and several function members. Among the member functions there are `hash_url()` to compute the hash value for the

```
class Wlist {
public:
    mo_window *winlist;
    int wincount;
    Wlist(mo_window* wl=NULL, int wc=0)
        :winlist(wl),wincount(wc){}
};

Wlist wlist(NULL,0);

mo_status mo_add_window_to_list (mo_window *win, Wlist& wp){
    wp.wincount++;

    if (wp.winlist == NULL)
    {
        win->next = NULL;
        wp.winlist = win;
    }
    else
    {
        win->next = wp.winlist;
        wp.winlist = win;
    }

    return mo_succeed;
}

mo_window *mo_next_window (mo_window *win, Wlist& wp)
mo_window *mo_fetch_window_by_id (int id, Wlist& wp)
mo_status mo_remove_window_from_list (mo_window *win, Wlist& wp)
```

Figure 13.

image cache and `mo_url_canonicalize()` to convert the URL string into its canonized form.

Several classes that handle personal annotations were also encapsulated. The class `annotation_entry` represents personal annotations for a single URL. Its most important data members are the URL and an array of annotation indexes. The annotation entries are stored in the `annotation_table` class.

From the image cache subsystem we recovered 3 classes. The class `cached_data` stores the actual cached data together with other data members specifying the last access and local name. The cached data is stored in the class `cache_entry` that also stores the URL of the cached data. The entries are stored in the class `image_cache` that is implemented as a hash table. The class `Htext` represents the hypertext stream transferred from a remote www server. The class contains methods for parsing and displaying the hypertext stream.

Of the 3,000 lines of the subsystem, we encapsulated approximately 60% of the code into the 12 classes listed above. Clone removal was applied for the `url` class because during the first phase of class encapsulation we created 5 separate classes representing the URL concept.

As an example of application of the scenario we present encapsulation of two global variables together with relevant code in a new class `Wlist` that implements

```
class Wlist {
public:
    mo_window *winlist;
    int wincount;
    Wlist(mo_window* wl=NULL, int wc=0)
        :winlist(wl),wincount(wc){}
    mo_status mo_add_window_to_list (mo_window *win);
    mo_window *mo_next_window (mo_window *win);
    mo_window *mo_fetch_window_by_id (int id);
    mo_status mo_remove_window_from_list (mo_window *win);
};

Wlist wlist(NULL,0);

mo_status Wlist::mo_add_window_to_list (mo_window *win){
    wincount++;
    if (winlist == NULL)
    {
        win->next = NULL;
        winlist = win;
    }
    else
    {
        win->next = winlist;
        winlist = win;
    }
    return mo_succeed;
}

mo_window *Wlist::mo_next_window (mo_window *win)
mo_window *Wlist::mo_fetch_window_by_id (int id)
mo_status Wlist::mo_remove_window_from_list (mo_window *win)
```

Figure 14.

a linked list of mosaic windows. Consider the original code fragment in figure 11. The two global variables `wlist` and `wincount` represent the data part of the link list and the functions represent the functions that we decided (based on the code review) belong to the future class. For clarity the full code is given only for the `mo_add_window_to_list()` function, which adds a new window to the list. For the remaining functions we give only signatures.

First, the scenario to create a data-only class out of individual variables is used. The result of the scenario application is represented in figure 12; changes are marked in bold. Then, the restructuring proceeds with the application of the scenario that adds functions to the data-only class. The functions relevant to the `Wlist` class fall into the third category as described in the scenario, since they access the class through global instance (`wlist`). In the first step we convert the access of the list in the code from the global variable `wlist` into the parameter `wp` (see figure 13) and we insert the relevant functions into the class (see figure 14).

Finally, the data encapsulation can be improved by the Change access specifier tool, to make the data members protected, as they are used only by the member functions of the class `Wlist` (see figure 15).

6. Related work

There has been extensive previous work that concentrates on the discovery of classes in imperative code. In [18] Yeh proposes to identify classes by relating global variables to routines that reference them. Yeh also presents heuristics that relate functions to user-defined types. A function should be included in the class if it has access to the type's internal fields.

In [13] Liu groups user-defined types and operations based on the assumption that an operation belongs to a type if the type is included in the signature of the function. However, if the function signature contains two such user-defined types, where one is a container and the other is a component, the function is always coupled to

```
class Wlist {
protected: mo_window *wlist;
protected: int wincount;
public:
    Wlist(mo_window* wl=NULL, int wc=0)
        :wlist(wl),wincount(wc)
    {}
    mo_status mo_add_window_to_list (mo_window *win);
    mo_window *mo_next_window (mo_window *win);
    mo_window *mo_fetch_window_by_id (int id);
    mo_status mo_remove_window_from_list (mo_w indow *win);
};
```

Figure 15.

the container rather than to the component. This precondition ensures that functions that operate on elements and their containers (insert element into hash tables) will not be coupled with the element types. This method can not group standalone variables into classes - it works only on records (C structures).

The clustering approach [8] identifies groups of entities (functions, user-defined types and global variables) that should belong to the same class based on the level of similarity of the relationships they are involved in. The authors present an iterative algorithm that combines entities with similar relations into one group. The algorithm terminates when all groups of entities have their similarity measure below a preset threshold.

A study in [9] shows none of the approaches is truly satisfactory. In our case study we identified classes intuitively based on personal object-oriented expertise.

Other related work deals with restructuring. Johnson and Opdyke [14,15] study class restructuring of classes related by composition and inheritance. Their transformation set includes the creation of an abstract superclass, subclassing, and refactoring to capture aggregations and components. Among the transformations they propose in [14,15], there are transformations similar to our function encapsulation and add/remove parameter transformations. They also propose a number of complementary low level transformations.

Griswold [10] investigates meaning-preserving transformations in the block-structured language Scheme. He defined the meaning-preserving transformations on a program dependency graph that is extracted from the source code. Transformations are defined by graph transformation rules and manipulate statements within one block. His transformation rules include:

- Transitivity - used for adding new variable to store intermediate result,
- Control - used for moving an assignment statement into or out of conditional block,
- Substitution - used to rename a local variable.

Lakotia [11] presents a collection of evolutionary transformations defined on a procedural language without global variables. The language contains assignment statement, branch statement, and function call statement (function call cannot be used in expression). His Fold transformation is similar to our function encapsulation tool, as it creates a function from a set of statements. The transformation set contains further transformations that can bring together non-continuous regions of code. Lakotia also proposes a Group transformation that collects a set of variables to form a structure. This transformation can be used for tasks similar to our variable insertion.

NCSA Mosaic was also used in reengineering case studies [2].

7. Conclusions

Our goal is to make the restructuring process practical for industrial applications. This goal is reflected in the choice of a small but efficient tool-set and in the application of the tools in restructuring scenarios where the tools are complemented by frequent programmer interventions. These scenarios summarize activities that must be performed, whether they are handled by automated tools or by the human user. The tools automatically perform precondition checks, changes, and ripple effect compensations. The advantage gained from the use of the tools increases with the size and complexity of the code. While these tools and scenarios speed up the programmer's work and help avoid errors that are common during restructuring, they cannot replace a qualified programmer, who still has to make all the decisions.

The code produced by transformations, when compared to the code produced by an experienced object-oriented programmer, may have obvious deficiencies. For example, alias pointers to variables that were inserted into classes are pointing to members inside of a class instance, which is a poor practice. It is desirable to convert this type of pointer from the "pointer to member type" to the "pointer to class type" and then access the member through the `->` operator.

Another problem is that many class data members are public. One way to lessen this defect is to create access functions, i.e. to create functions `set()` and `get()` that set and return the value of the selected data members, respectively. The next step is to replace direct accesses to the data members by access function calls and change the specifiers of the data members towards a higher level of protection.

These and similar goals belong into the category of "improving the encapsulated classes"; sometimes, they depend on a particular programming style or individual preferences. The tools that would support these transformations are among the tasks for future work.

In our future work, we intend to improve the analysis parts of the tools and make them handle more complicated situations. We also want to add tools that improve the quality of the resulting classes, making them look more like classes produced by an experienced object-oriented programmer. For frequently repeated scenarios, we intend to aggregate the tools into more powerful and specialized assemblies that will provide more integrated support with less need for human intervention. We hope that these efforts will result in increased reusability and evolvability of legacy software.

References

- [1] Canfora G., Cimitile A., Munro M.: "Extracting abstract data types from C programs: A Case Study", in Proceedings of the International Conference on Software Maintenance, , IEEE Computer Society Press, Los Alamitos CA, 1993, pp.200-209.
- [2] Clayton L., Rugaber S., Taylor L., Wills L., "A Case Study of Domain-based Program Understanding", in Proceedings of the International Workshop on Program Comprehension, IEEE Computer Society Press, Los Alamitos CA, 1997, pp. 102-110.
- [3] Devanbu P., "GENOA A Customizable, Language- and Front-end Independent Code Analyzer", In Proceedings of ICSE '92, 1992, pp. 307-317.
- [4] Dunn M.F., Knight J.C., "Software Reuse in an Industrial Setting: A Case Study", in Proceedings of the International Conference on Software Engineering, 1991, pp.329-338.
- [5] Fanta R., Rajlich V., "Reengineering Object-Oriented Code", in Proceedings of the International Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos CA, 1998, pp. 238 - 246.
- [6] Fanta R., Rajlich V., "Removing Clones from the Code" submitted to Journal of Software Maintenance, 1998.
- [7] Girard J.F., Koshke R., "Finding components in a hierarchy of modules: step towards architectural understanding" in Proceedings of the International Conference on Software Maintenance, IEEE Computer Society Press, Los Alamitos CA, 1997, pp.66-75.
- [8] Girard J.F., Koshke R., "A metric-based approach to detect abstract data type and abstract state encapsulation", Automated Software Engineering Conference, Nevada, 1997, pp. 82-89.
- [9] Girard J.F., Koshke R., "A Comparison of Abstract Data Types and Abstract State Encapsulation Detection Techniques for Architectural Understanding", Working Conference on Reverse Engineering, Amsterdam, 1997, pp. 66-75.
- [10] Griswold W.G., "Automated Assistance for Program Restructuring", ACM Transactions on Software Engineering and Methodology, Vol. 2, No. 3, July 1993, pp. 228-269.
- [11] Lakotia A., Deprez J-C. "Restructuring programs by tucking statements into functions", Information and Software Technology, 1998, 677-689
- [12] Lewis J.A., Henry S.M., Kafura D., "An Experiment Study of the Object-Oriented Paradigm and Software Reuse", In proceedings of OOPSLA'91, 1991, pp.184-196.
- [13] Liu S., Wilde N., "Identifying objects in conventional procedural language: An Example of Data Design Recovery" in Proceedings of the International Conference on Software Maintenance, , IEEE Computer Society Press, Los Alamitos CA, 1990, pp. 266-271.
- [14] Opdyke W.F., Johnson R.E., "Refactoring and Aggregation", In Proceedings of ISOTAS'93: Object Technologies for Advanced Software, Lecture Notes in Computer Science, vol.742, Springer-Verlag, Berlin Germany, 1993, pp. 264-278.

- [15] Opdyke W.F., Johnson R.E., "Creating Abstract Superclasses by Refactoring", In Proceedings of Computer Science Conference, ACM Press, New York NY, 1993, pp. 66-73.
- [16] Parnas D.L., "On Criteria to be Used in Decomposing Systems into Modules", Communications of ACM 15(12), 1972, 1053-1058.
- [17] Wills L.M., "Flexible Control for Program Recognition", Working Conference on Reverse Engineering, Baltimore, 1993, pp. 134-143.
- [18] Yeh A.S., Harris D., Reubenstein H., "Recovering Abstract Data Types and Object Instances from a Conventional Procedural Language", Second Working Conference on Reverse Engineering, 1995, pp.227-236.
- [19] <ftp.ncsa.uiuc.edu/Mosaic/Unix/source>