

Reengineering Object-Oriented Code¹

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 the reengineering of a deteriorated object-oriented industrial program written in C++. The main problem of the program was misplaced code, most often functions that were members of the wrong class. In order to deal with this problem, we designed and implemented several restructuring tools and used them in specific reengineering scenarios. We also discuss how this set of tools could be enhanced in the future, and the importance of restructuring for object-oriented software maintenance.

1. Introduction

Architectural properties of object-oriented software are influenced by a number of factors. One of the most important is software specification. Complete and accurate specification ensures that all user requirements will be taken into account when a software system is developed. Unfortunately, experience indicates that a complete specification cannot be obtained in advance, and that the specification will change during the development as more experience is gained. Cusumano and Selby in their case study [1] found that at most 70% of specification was accurately predicted and the rest changed during the development. This indicates that the software architecture, which is based on the original requirements, is likely to already experience significant changes during development. Since changes usually result in a deterioration of the architecture, there is a great likelihood that the architecture will already deteriorate in the early parts of the software lifecycle.

Deterioration further continues through mistakes and omissions, which may include an incorrect class encapsulation, or a missed encapsulation of important domain concepts. Finally, deterioration during maintenance is a

well-documented fact [18]. The combination of all these factors means that the deterioration of an object-oriented architecture is very likely, perhaps inevitable. When this deterioration reaches a certain point, it makes the comprehension and further evolution of the program difficult or impossible. In that situation, reengineering is unavoidable.

In this paper, we describe our approach to the reengineering of an object-oriented program developed at Ford Motor Company. In particular, we describe the design and implementation of three automatic transformation tools: a function expulsion tool, a function insertion tool and an encapsulation tool. In addition, we specify several reengineering scenarios that employ these tools.

Related research [6,9,10,12] deals with code restructuring through behavior-preserving code transformations. Our function encapsulation tool performs operations similar to the transformations proposed by Opdyke [6] and Griswold [10]. Opdyke [6] focused on restructuring classes related by composition and inheritance. His transformations include the creation of abstract superclasses, subclasses, aggregations, and components. Griswold in [10] specified a set of transformations for block structured languages.

The paper consists of five sections. Section 2 provides a short description of the PET project and the structural problems of its code. In section 3 we discuss the implementation of the restructuring tools. Scenarios of work with the tools are described in section 4. The conclusions are in section 5.

2. Project PET

PET is a CAD tool developed at Ford Motor Company [2] to support the design of the mechanical components of a car, like transmission, engine, etc. It is implemented in C++ and every mechanical component is modeled as a C++ class. Components are hierarchically composed into more complex components. For example, an engine is

¹This work was partially supported by a grant from Ford Motor Co. and NSF grant #CCR-9803876

composed of an engine block, pistons, shafts, etc. Each

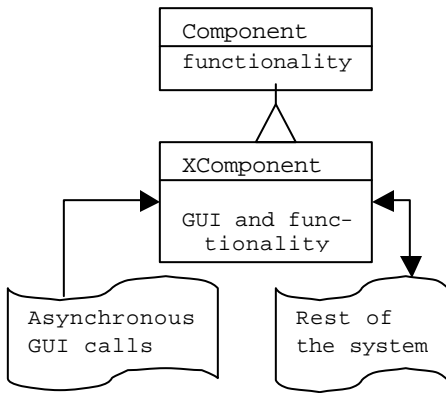


Figure 1.

component is characterized by a set of parameters dependent on the parameters of neighboring components. The component dependency is described by a set of equations. Relationships among components and their parameters constitute a complex dependency network. Whenever a parameter value is changed, an inference algorithm traverses the entire network and recalculates the values of all dependent parameters. The value of each calculated parameter is checked for consistency against pre-set constraints.

Pet consists of 120 000 lines of C++ code, structured into approximately 80 classes and 50 global functions. The code is divided into 200 files. It is interfaced with other CAD software, including optimization software and 3-D modeling software.

The object-oriented architecture of PET was influenced by large changes in the specification. PET was developed as a novel tool, and user requirements substantially changed during initial use. We estimate that the initial specification covered no more than 30% of the current functionality. The design process did not fully anticipate these changes. In particular, several important concepts were not encapsulated in separate classes. The design also suffered from the fact that the early version of the C++ compiler did not support templates. Because of the heavy use of the program, all changes to PET were performed as quickly as possible in order to make the new functionality immediately available. This situation prevented any conceptual changes to the architecture, and the architecture progressively deteriorated. New programmers had difficulty understanding this rapidly expanding system, and new parts of the program were often structurally

```

class A {
public:
    int i;
protected:
    char c;
};

int foo(A& a){
    a.i=4;
}
  
```

Figure 2.

incoherent with the original architecture. With the growing size of the source code and a deteriorated architecture, further evolution became difficult. We identified misplaced data and misplaced code as the major structural problem of the PET architecture.

Code and data are misplaced if they are members of a different class than the class they naturally belong to. Misplaced code and data result in a program that is difficult to read, test and change. Examples in PET are list operations for a search in the list and the intersection of two lists, neither of which is a member of the class `List`. Both are members of other classes and have the class `List` as arguments. In this case, every class that uses `List` has to provide its own implementation of search and intersection.

Another example is PET's user interface implemented in Motif. Many PET classes mix the user interface and application functionality. For example, there are two classes that represent mechanical components in PET: `Component` and `XComponent`, see figure 1. The class `Component` defines the basic data structures of the mechanical components. This class is the base class of the `XComponent` class, which contains additional data and function members. The members of `XComponent` mix both the user interface code and the application functionality code. Moreover the class `XComponent` is accessed by Motif callback functions whose calls are generated by asynchronous user actions.

3. Tools for reengineering

To support PET reengineering, we implemented three high level editing tools for C++ code restructuring. The tools are general and can be used in other projects as well. The initial set contains:

- *Function insertion* - moves a function into a class.
- *Function encapsulation* - encapsulates a consecutive fragment of code into a new function.
- *Function expulsion* - moves a function out of a class.

These transformations were the ones most extensively required during the restructuring. They cover only part of the restructuring task, and the rest is being performed manually using standard editors, see the next section. We decided to implement these transformations with certain limitations on their functionality. These limitations made the implementation easier, and they do not adversely affect common use of the transformations. They are sum-

```

class A {
public:
    int foo();
    int i;
protected:
    char c;
};

int A::foo(){
    this->i=4;
}
  
```

Figure 3.

```

main() {
A la;
...
foo(la);
...
}

main() {
A la;
...
la.foo();
...
}

```

Figure 4.

marized below.

Function insertion inserts a standalone function into a target class and makes the function a public member of that class. Before the insertion, the target class must be one of the arguments of the function. An example of the starting situation is in figure 2, the result is in figure 3. The following code changes are performed:

- The function header is inserted into the class specification.
- Function body update: Members of the target class must be accessed directly in the function body, see figure 3.
- Function header update: The function header must be qualified by the class identifier. The parameter that is now replaced by class membership must be removed.
- Function call updates: All calls to the function must be qualified with a class instance, see figure 4.
- All forward declarations of the function have to be removed, because they are replaced by the new function declaration in the target class specification.

The pseudo code of the function insertion is in the Appendix. In order to simplify the implementation, we accepted several limitations:

- The inserted function cannot be a member of any class.
- The inserted function cannot be called through a pointer.
- The inserted function cannot be overloaded, cannot have a variable number of parameters, and cannot be a template function. We restricted the function name overloading in order to prevent name conflicts when a function is moved into the namespace of the target class.

```

void foo(char c) {
int i,count,len;
char str[MAX];

cin>>str;
len=strlen(str);

count=0;
for(i=0;i<=len;i++)
    if(str[i]==c) {
        count++;
        str[i]='\n';
    }

cout<<str<<" "<<count<<"\n";
}

```

Figure 5.

- The function can only be inserted into a class of one of its parameters. If the parameter is a pointer to a class then this pointer cannot be interpreted as an array, or involved in any pointer arithmetic expression within the body of the function.

Function encapsulation encapsulates a sequence of statements into a new function. The user selects a block of code for encapsulation and the tool will decide whether the block is syntactically complete and can be changed into a function body. If yes, a new function is created. The selected block is then encapsulated as a function body and replaced by a function call. If the encompassing function is a member of a class, the new function also becomes a member of the same class. For example in figure 5, the framed text area represents code selected for encapsulation. The result of the encapsulation is shown in figure 6.

This transformation analyzes the selected block of code, and all encountered variables are classified into one of the following categories:

1. Local variable
2. Global variable
3. Parameter passed by value
4. Parameter passed by reference

The tool generates a *local variable* in the new function if the original variable does not carry any information into or out of the selected block. Such situation can occur even when the original variable is declared outside the selected code or when it is passed as a value parameter to the original function. The candidate for local variable must be written before being read inside of the selected code and at the same time it must be written before it is read or not accessed at all in the rest of the code of the original function.

If a variable is *global* in the original code, it remains global in the encapsulated function.

A variable will be passed as a *value parameter* to the new function if its value is used within the code selected

```

void foo(char c) {
int i,count,len;
char str[MAX];

cin>>str;
len=strlen(str);
newfun(count,len,str,c);
cout<<str<<" "<<count<<"\n";
}

newfun(int& count,int len,
char* str,char c){
int i;

count=0;
for(i=0;i<=len;i++)
    if(str[i]==c) {
        count++;
        str[i]='\n';
    }
}

```

Figure 6.

for encapsulation but any modifications to this variable in this block are not used elsewhere. For a variable to fall into this category, it must be declared local or passed by value in the original code and all accesses must be read-only within the selected code. If the value of the variable in question is changed within the specified block or if it is passed by reference to the encompassing function, then it must be overwritten in the rest of the code before any read access is performed.

The rest of the variables that do not qualify as local variables, global variables or value parameters may carry information or side effects from the new function. Such variables are passed as *reference parameters*.

In order to simplify the implementation we allowed some input only parameters to be classified as reference parameters. This, however, does not affect the code execution. As a further simplification, template functions are not supported.

Function expulsion removes a member function from a class and makes it a standalone function. The class is now passed to the function as a parameter. This transformation is complementary to the function insertion. The following code changes must be performed in the code:

- The function header is removed from class specification.
- Access to private and protected members must be performed through newly generated public access functions.
- Function header update: The class qualifier must be removed from the function header and the additional parameter must be added.
- Function code update: Directly accessed members of the class must now be accessed through the additional parameter.
- Function call updates: The qualifying instance must be changed to a parameter in all function calls.
- A forward declaration of the expelled function must be included in the file that holds the source class specification.

In order to simplify implementation, we accepted the following limitations on function expulsion:

- A pointer to the old function cannot be assigned to any pointer variable throughout the code.
- Functions with a variable number of parameters cannot be expelled.
- Function members of classes that are part of inheritance hierarchies cannot be expelled.
- Template and overloaded functions cannot be expelled. Preventing overloaded functions from being expelled ensures that no naming conflicts are introduced into the program.

In the implementation of the transformations, we used GEN++ [3] to analyze the source code and C++ to per-

form actual changes to the code. GEN++ first runs the source code through the C++ pre-compiler and then parses the resulting code. The parsed code is stored in an abstract semantic graph. GEN++ also provides a LISP-like scripting language that can be used for writing queries investigating the graph. Several difficulties were encountered during the implementation of the transformation tools.

Pre-compiler directives, particularly conditionally compiled blocks, often occur in PET and in C++ code in general. Figure 8 shows the resulting code configuration when the variable TEST is defined in the example in figure 7. Since the GEN++ parser runs code first through the C++ pre-compiler, code blocks that are excluded in the pre-compiler phase will not be parsed and thus they will not be transformed. As a solution to this problem, we examine the code first in order to discover all pre-compiler variables used for conditional compilation. Based on this pre-analysis, we run the GEN++ analyzer several times to cover all possible software configurations, and the results of this multiple analysis are combined in a single output file.

The dependencies between C++ files may change by the transformations. As an example, consider the expulsion tool. Whenever a function is expelled, its prototype is removed from the class definition. The C++ files that include the definition of that class lose the function prototype definition. This situation could cause compile time errors. As a solution, the expulsion tool adds an additional forward declaration of the function into the file that contains the class specification. In this way, the tools ensure that the code will compile successfully.

Complexities of C++ that make the implementation of the tools difficult include automatic constructor and destructor invocations, automatic generation of temporary variables by the compiler, and pointer arithmetic. We deal with these problems by setting certain limitations for every transformation. These restrictions concern unusual situations and are not likely to occur in practice.

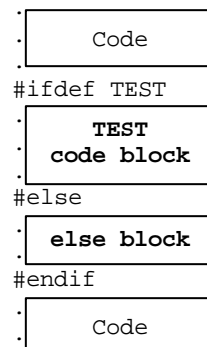


Figure 7.

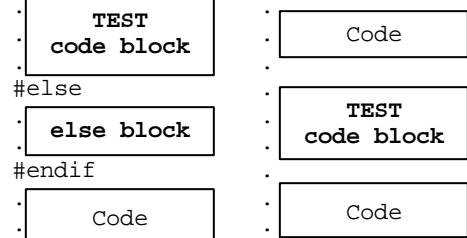


Figure 8.

```

class A {
public:
    B* b;
    int a_data;
    void a_fun();
    void foo();};

void A::a_fun() {
//call foo
this->foo();
}

void A::foo(){
//access A
this->...
//access B
b->...
}

class B {
public:
    A* a;
    int b_data;};

```

Figure 9.

```

class A {
public:
    B* b;
    int a_data;
    void a_fun();
    void foo(B*);};

void A::a_fun() {
//call foo
this->foo(b);
}

void A::foo(B* lb){
//access A
this->...
//access B
lb->...
}

class B {
public:
    A* a;
    int b_data;};

```

Figure 10.

```

class A {
public:
    B* b;
    int a_data;
    void a_fun();};

void A::a_fun(){
//call foo
b->foo(this);
}

void B::foo(A* d){
//access A
d->...
//access B
this->...
}

class B {
public:
    A* a;
    int b_data;
    void foo(A*);};

```

Figure 11.

```

class A {
public:
    B* b;
    int a_data;
    void a_fun();};

void A::a_fun(){
// call foo
b->foo();
}

void B::foo(){
//access A
a->...
//access B
this->...
}

class B {
public:
    A* a;
    int b_data;
    void foo();};

```

Figure 12.

In section 5 we also discuss complexities inherent to the object-oriented paradigm, independent of any particular programming language.

4. Reengineering scenarios

The application of our high level editing tools during restructuring is described in this section. The tools are used in scenarios that involve the actions of both the tools and their human user. Specifically, we present reengineering scenarios for moving misplaced data and functions between classes that are related to each other by either composition or use. As indicated earlier, this problem was identified as the most serious problem in the structure of PET.

Composition is a relation between composite class and several other classes called components. A composite class has several member variables of the component type. Sometimes the component has an additional reference to the composite class, and in that case it can access members of the composite.

Use relationship means that a class is using another class as a formal argument in one of its functions. The tools, described in section 3, are better suited for supporting scenarios for moving functions between classes related by the use relation. However, they can be also used in scenarios for moving functions between classes related by composition. In section 5 we propose a richer transformation set to remove this asymmetry.

When a function is misplaced to a using class then it has the used class as one of the formal parameters. The

following scenario moves the misplaced function into the used class:

1. Expulse the function from the using class
2. Insert the function into the used class

These steps are accomplished by the expulsion and insertion tools, described in section 3.

Functions in PET are often misplaced to the composite instead of the component. A simplified example of such a situation is in figure 9, where function `foo()` is misplaced to the composite `class A` and will be moved to the component `class B`, see figures 10 through 12. All changes performed during this scenario are printed in bold. The following scenario does the restructuring:

1. Add a new parameter of the composite type (`B* lb` in figure 10) to the misplaced function.
2. In the misplaced function, access all component members through the new parameter.
3. Expulse the misplaced function from the composite.
4. Insert the misplaced function into the component. See figure 11.
5. If the component has a member referencing the composite (like `A* a` in `class B`), use the reference to access the composite's members, see figure 12. If such member is not available, omit steps 5 and 6.
6. Remove the composite from the parameter list. See figure 12.

Steps 1 and 6 are performed manually, steps 2 and 5 are performed using automatic text replacement and steps 3 and 4 are performed by the expulsion and insertion tools. This scenario can be performed only when the multiplicity of the association between the component and the composite is 1:1.

```

class A {
public:
    void a_fun();
protected:
    B* b;
    int a_data;
};

void A::a_fun(){
//access a_data
...this->a_data;
}

class B{
public:
    void b_fun();
protected:
    int b_data;
}

```

Figure 13.

```

class A {
public:
    void a_fun();
protected:
    B* b;
};

void A::a_fun(){
//access a_data
...b->a_data;
}

class B{
public:
    void b_fun();
protected:
    int b_data;
}

```

Figure 14.

For the complementary operation (moving a function from a component to the composite), an identical scenario can be used. It is, of course, possible that the misplaced function is used in one of the component's other member functions and that the component does not have access to the composite class through a member reference. In such cases, the member reference has to be added or all affected member functions must receive an additional formal parameter.

Data misplaced from components to composites are moved to their proper place by the following scenario, see an example in figures 13 and 14.

1. Move the selected data into the component. See figure 14.
2. In the composite code, access the moved data through the component reference. See figure 14.
3. In the component code, the original data were accessed through a member reference to the composite class or through a parameter passed to a component member function. In both cases, the access to the moved data must be changed, and the data will be accessed directly.

This scenario can be safely applied only when the cardinality of the association between a composite and a component is 1:1 and when the data are not accessed in different components. Possible naming conflicts or violation of access specifiers must be resolved before scenario application. All of the steps of this scenario are manual.

For the complementary operation (moving data from a

```

class XComponent {
public:
    Widget top_level();
    void change_color(int);
    int update_component();
    ...
protected:
    Widget dialog;
    Widget text_field;
    double data;
    ...
};

```

Figure 16.

component to the composite), an identical scenario can be used. If the component does not have access to the composite through its data member, all member functions of the component using the moved data must be provided

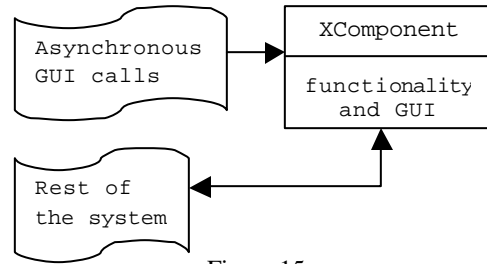


Figure 15.

with access to the composite class through an additional parameter before the scenario is applied.

In PET, one of the main tasks was to separate the user interface code from the application code. For that, we used a general reengineering scenario which separates two different concepts mixed in a single class. In our particular situation, the scenario separates Motif code from application code. The complete scenario is summarized in the following way:

1. Create a new user interface class as a component of the original class.
2. Move the user interface data into the user interface class.
3. Encapsulate all user interface code as new functions.
4. Encapsulate the remaining functional code as new functions.
5. Move all user interface functions into the user interface component.

Step 2 is the previous scenario (moving data between composites and components). Steps 3, 4 and 5 are supported by a repeated application of the insertion, encapsulation and expulsion tools. Step 4 is necessary because sometimes we need to encapsulate functional code that is nested within a user interface function newly encapsulated in step 3.

In order to illustrate this scenario, we restructure the code presented in figure 1. Before the scenario is applied, we move all data and all functions from class Component into class XComponent and delete class Com-

```

class XComponent {
public:
    Widget top_level();
    void change_color(int);
    int update_component();
    ...
protected:
    CompWin* w;
    Widget dialog;
    Widget text_field;
    double data;
    ...};

class CompWin {
public:
    XComponent* x;};

```

Figure 17.

```

class XComponent {
public:
    Widget top_level();
    void change_color(int);
    int update_component();
    ...
protected:
    CompWin* w;
    double data;
    ...};

class CompWin {
public:
    XComponent* x;
    Widget dialog;
    Widget text_field;};

```

Figure 18.

ponent. This is easily done because Component does not have any clients. Its sole derived class - XComponent - does not override any functionality and does not contain any virtual functions. Figure 15 gives the modified architecture diagram. Figure 16 contains a fragment of the class definition before the scenario is applied.

In the first step, a new empty class CompWin is created as a component of the class XComponent (figure 17). This includes the creation of an instance of CompWin during XComponent's initialization. In the second step, user interface data are moved into the new class (figure 18). In the third step, new pure user interface functions are encapsulated. This step is performed using the encapsulation tool. An Example of such a function is SaveCB() in figure 19. The function is called by a Motif callback function whenever an appropriate asynchronous event is generated by the user. In step 4, new functions that contain only functional code are encapsulated. Then, the user interface code and the functional code are separated into distinct functions. Finally, the user interface functions are moved from class XComponent to the

```

class XComponent {
public:
    int update_component();
    ...
protected:
    CompWin* w;
    double data;
    ...
};

class CompWin {
public:
    XComponent* x;
    Widget dialog;
    Widget text_field;
    Widget top_level();
    void change_color(int);
    void SaveCB();};

```

Figure 20.

```

class XComponent {
public:
    Widget top_level();
    void change_color(int);
    int update_component();
    void SaveCB(CompWin*);
    ...
protected:
    CompWin* w;
    double data;
    ...};

class CompWin {
public:
    XComponent* x;
    Widget dialog;
    Widget text_field;};

```

Figure 19.

class CompWin. The final code is shown in figure 20, and the resulting architecture is in figure 21.

5. Conclusions

Our initial experience showed that the original set of three transformations can be used efficiently but also that additional restructuring tools are needed. Here, we list the tools that we intend to implement in order to fully support all of the scenarios described in Section 4.

- Transfer of data between composites and components.
- Transfer of functions between composites and components.
- Function encapsulation based on data flow analysis. This tool could facilitate code reordering and enhance the functionality of the encapsulation transformation.
- Change of an access specifier of a member, from private to public, etc.
- Generator of access functions for data members.
- Tool renaming a data member, variable, or a function argument.

The first two tools are similar to the expulsion and insertion tools already in existence. The last three tools facilitate smaller changes to the code, but they are frequently required to support larger transformations and scenarios. Especially important is the generator of access functions. This generator is already a part of the function

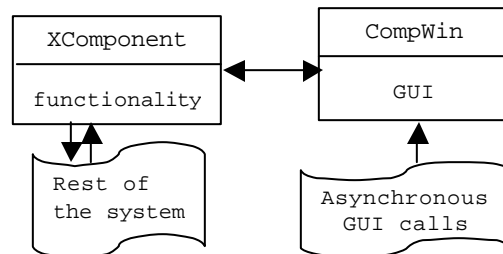


Figure 21.

expulsion tool, but it is often required in other contexts and should be implemented as a separate tool.

The transformations listed here do not dictate any specific architecture. They are a basic set that fixes the most glaring deficiencies. Our experience showed that even small regroupings of data and code has a positive impact on the code architecture, substantially improving its comprehensibility and maintainability.

Object-oriented technology has become one of the most popular software technologies. However, experience has shown that object-oriented architectures are bound to deteriorate and that transformations that rectify the architecture are a necessary ingredient of the technology. However, these transformations are inherently complex and surprisingly hard to implement. Some of the difficulties are due to the complexities of the specific languages involved, but others are caused by the very nature of object-oriented principles. For example, moving functions out of classes requires the automatic generation of access functions for the private data, which in full generality is a complicated transformation. Another example is a function that is a member of a base class. It can be inherited and used in derived classes, making its move to component classes or using classes difficult or impossible. Dynamic binding creates another set of difficulties. Hence, not only the accidental properties of C++ but also the principles of object-oriented programming make the transformations complex.

Our approach has been to limit the scope of the transformations to the most likely cases and avoid theoretically interesting but difficult and impractical situations. Further research toward a practical set of transformations and scenarios is needed to support efficient object-oriented code maintenance.

Acknowledgements

We want to acknowledge the contribution made by other people on the project, in particular Kunrong Chen, Steve Moenssen, Yihui Sun and Cathy Yang. We also want to acknowledge the support provided by Ford personnel, particularly Garry Vrsek, Tony Mikulec, Libor Soucek and Xianren Li.

6. References

- [1] M. A. Cusumano, R. W. Selby, "How Microsoft Builds Software", Communications of ACM, Vol. 40, June 1997, pp.53-61
- [2] T. Mikulec, X. Li, G. Vrsek, "Powertrain Engineering Tool an its application", XXVI Congress FISITA, Prague 1996.
- [3] P. Devanbu, L.E. Eaves, "How to write GEN++ Specification" AT&T Draft, 1994.
- [5] J. Mayrand, C. Leblanc, E. M. Merlo, "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics", IEEE, 1996
- [6] W.F. Opdyke, "Refactoring Object-Oriented Frameworks", PhD thesis, University of Illinois at Urbana-Champaign, 1992.
- [7] R. E Johnson, W. F. Opdyke, "Refactoring and Aggregation", In Proceedings of ISOTAS'93, November 1993.
- [8] R. E Johnson, W. F. Opdyke. "Creating Abstract Superclasses by Refactoring", In Proceedings of CSC'93, The ACM Computer Science Conference, February 1993.
- [9] M. Blaha, W. Premerlany, "A Catalog of Object Model Transformations", Proceedings of WCRE '96, 1996
- [10] W.G. Griswold, "Automated Assistance for Program Restructuring", ACM Transactions on Software Engineering and Methodology, Vol. 2, No. 3, July 1993, pp. 228-269.
- [11] W.G. Griswold, "Program Restructuring as an Aid in Software Maintenance", PhD thesis, University of Washington, 1991.
- [12] E Casais, "The Automatic Reorganization of Object Oriented Hierarchies, case study", Centre Universitaire d'Informatique, University of Geneva, 1991
- [13] E. Casais, "An incremental Class Reorganization Approach", In Proceedings of 6th ECOOP Conference, Springer Verlag, Utrecht 1992, pp. 114-132
- [14] V. Rajlich, S. Adnapaly, "VIFOR 2: A Tool for Browsing and Documentation", Proc. 1996 IEEE International Conference on Software Maintenance, 1996, pp. 296-299.
- [15] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns", Addison-Wesley, 1995.
- [16] V. Rajlich, "Decomposition/Generalization Methodology for Object-Oriented programming". J. Systems Software 24, 1994, pp. 181-186,.
- [17] J. Rumbaugh, M. Blaha,, W. Premerlani,, W. Lorensen,, "Object-Oriented Modeling and Design", Prentice-Hall, 1991.
- [18] L. A. Belady, M. M. Lehman, "Programming System Dynamics or the Metadynamics of Systems in Maintenance and Growth", IBM Research Report RC 3546, 1971.

Appendix

In this Appendix, a pseudo-C description of the function insertion tool is given. The tool uses the following input data:

1. List of C++ source files
2. Name of a function to be inserted
3. Name of a class the function is being inserted into
4. Position of the parameter to which the function is being inserted
5. Flavor of the parameter - the parameter type can be class, reference to a class, or pointer to a class.

In the following description, we first describe three auxiliary functions (Pre-compiler directive analyzer, Restriction analyzer, Change analyzer) and then the Insertion tool, which calls them.

Pre-compiler directive analyzer

```
scan a C++ file line by line {
```

```

    if(new conditionally compiled block found) {
        construct new command line to compile this
        block;
        store command line into a list;
    }
}
scan the list of command lines and delete all
duplicates;
save the list into a file.

```

Restriction analyzer (implemented in GEN++)

```

read the input data {
    read function name into fname;
    read class identifier into whichclass;
    read parameter position into pposition;
    read parameter flavor into pflavor;
}
start from the root of the g-tree;
go down through all nodes {
    if(current node is function identifier)
        if(function name==fname){
            find parent for the node;
            if(a parent is a pointer expression)
                print error message;
        };
}
start from the root of the global declarations;
go down through all nodes {
    if(definition node of fname function is reached)
    {
        if (template function) print error message;
        if (overloaded function) print error message;
        if (variable number of parameters) print er-
        ror message;
    }
    find type of the argument on the pposition;
    switch(type) {
        class type: OK;
        class pointer: OK;
        class reference: OK;
        default: print error message;
    }
}
go to the node representing the body of the
fname function;
if(flavor of the argument on the pposition is
    pointer to a class){
    if(the argument is array) print error message;
    if(the argument is used in pointer arithmetic)
        print error-message;
}
}

```

Change analyzer (implemented in GEN++)

```

read the input data {
    read function name into fname;
    read class identifier into whichclass;
    read parameter position into pposition;
    read parameter flavor into pflavor;
}
go to the root of all global symbols;
go through all global declaration nodes;
{
    if(a class definition is reached &&
        class name==whichclass)
        print "line# file whichclass";
    if(a function declaration is reached &&
        function name==fname)
        print "line# file fname pposition";
}

```

```

if(a function definition is reached &&
    function name==fname) {
    print "function name line# file name po-
    sition";
    find argument name on position pposition
    in argument list;
    go to the function body {
        examine all expressions in the body;
        if(argument name is found)
            print "line# file type of change";
    }
}
}
go to the root of the tree;
go through all nodes;
if(node is a function call)
    if(fname function call)
        print "line# file fname pposition pflavor";
}

```

Insertion tool

```

read list of C++ files;
for(each file in the list)
    call Pre-compiler directive analyzer;
for(each file in the list)
    for(each command line generated by
        Pre-compiler directive analyzer){
        call Restriction analyzer;
        merge all output files;
    }
if(no error messages were printed) {
    for(each C++ file in the list){
        for(each command line generated by
            Pre-compiler directive analyzer){
            call Change analyzer;
            merge all output files;
        }
        for(every line in the output file){
            find the line # in the C++ file;
            make the change;
        }
    }
}

```