

Removing Clones from the 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 discuss the elimination of function and class clones from industrial object-oriented code. Clone removal can decrease code size and facilitate maintenance. We eliminate clones by reengineering scenarios that are based on automated restructuring tools. The paper presents examples of clones, reengineering scenarios, and restructuring tools. The usefulness of the approach is demonstrated in a case study*

1. Introduction

Our research group has been studying the deterioration and restructuring of object-oriented industrial code [3]. During a review of industrial code we noticed that programmers often use identical or almost identical software components (e.g. classes, functions) in multiple places. These identical or almost identical components are called clones and previous research [7,8,9] suggests that they account for 5-15% of code in large software projects.

One reason for the existence of clones in the program is that maintenance programmers do not fully understand the program and therefore they re-implement some already existing functionality. Another reason is time pressure and the effort not to introduce any bugs into already working code, particularly in a situation when the code contains complicated or not fully understood dependencies. In that case, programmers frequently copy-and-paste the code and update only some copies. Such duplications increase code size and lead to bloated code. They also make maintenance and comprehension more difficult, since they de-localize concepts implemented in the code. Because as much as 80% of the total life cycle cost is spent on maintenance, clone elimination could translate into substantial savings.

In order to remove duplicated code, the code must be restructured. The restructuring is performed in two phases:

- Search for clones
- Replace clones by a single code entity

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

Other authors investigated the identification of clones in code, see for example [1,7,8]. In this paper, we present the next step, which removes the previously identified clones. The paper belongs to a broader field of software restructuring. It deals with two kinds of clones: function clones and class clones. It discusses restructuring tools and their broader context called reengineering scenarios. It does not discuss clone detection as it builds on the results of previous research [1,7,9].

Clone removal requires substantial effort, because identical or near identical clones are often scattered throughout the code. When clones are removed, ripple effects can affect related code. The ripple effects are often hard to detect, particularly in large-scale projects with complex dependencies. Therefore, manual removal of clones is difficult and error prone.

In this paper we present a tool-set that we used to remove clones in a medium-size C++ project. The tools analyze the code, make the desired changes, and compensate for ripple effects. The tools also ensure that all preconditions hold before any transformation is applied. The individual tools presented in this paper perform small and efficient transformations. For more complex tasks, we used scenarios that combine the tools with human intervention.

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

- Search for clones in medium size C++ program (PET)
- Specify clone removal scenarios
- Implement transformation tools
- Perform a case study

Section 2 of this paper describes restructuring tools that we use for clone removal. Sections 3 and 4 define the use of the tools in reengineering scenarios. Section 5 contains a case study of an industrial object-oriented code. Related work is reviewed in section 6 and section 7 contains conclusions and future work.

2. Tools for restructuring

To support the restructuring of object-oriented code, we designed and implemented several high-level restructuring tools. The tools are universal and can be used in other projects as well. They cover editing changes that potentially affect large portions of code, i.e. transformations that have to be done simultaneously and consistently in several places. Transformations that affect only a few spots and produce no ripple effects are not covered and are performed manually with standard editors. The initial set contains:

- *Function insertion*
- *Function expulsion*
- *Function encapsulation*
- *Renaming*
- *Argument reordering*

When using the transformation tools, the programmer specifies the input data and the tool performs the transformation on the entire code. After each transformation, the code can be successfully compiled and tested. Because of the complexity of C++, we decided to implement the transformations with certain limitations on their functionality. The limitations do not adversely affect common use of the transformations and they simplify

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

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

Figure 1.

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

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

Figure 2.

tool implementation. The transformations are summarized in the rest of this section.

Function insertion inserts a standalone function into a class as a new public member. Before the insertion, the target class must be one of the arguments of the function. An example of the starting situation is in figure 1, the result is in figure 2 and 3. The following code changes are performed:

- The transformation inserts the function header into the class specification.
- Members of the target class are accessed directly in the function body (see figure 2).
- The function header is qualified by the class identifier, and the parameter that is now replaced by membership is removed.
- All calls to the function are qualified with a class instance (see figure 3).
- All forward declarations of the function are removed.

In order to simplify the implementation we accepted the following limitations:

- The inserted function cannot be a member of any class.
- The inserted function cannot be called through a pointer, cannot be overloaded, cannot have a variable number of parameters, and cannot be a template function.
- The function can be inserted only 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 within the body of the function.

The function insertion tool takes the following data as input:

- Name of the target class
- Position of parameter that is to be replaced by membership
- Name of the selected function
- List of files that are affected by the transformation

After receiving its input data the tool analyzes the source code and makes sure that all of the limitations hold. Then the tool performs the transformation. The tool accepts three different class parameter types to be converted to membership: reference, value and pointer. The tool also supports the insertion of recursive functions.

Function expulsion is complementary to the function insertion. It removes a member

```
main() {
    A la;
    B lb;
    ...
    foo(lb,la);
    ...
}

main() {
    A la;
    B lb;
    ...
    la.foo(lb);
    ...
} 3
```

Figure 2

function from a class and makes it a standalone function. The class is now passed to the function as a parameter. The following code changes are performed in the code:

- The transformation removes the function header from the class specification.
- Access to private and protected members is performed through public access functions
- The class qualifier is removed from the function header and an additional parameter is added.
- Members of the class accessed directly are accessed through the additional parameter.
- In all function calls, the qualifying instance is changed into a parameter.
- A forward declaration of the expelled function is added to the file that holds the source class specification.

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

- A pointer to the old function cannot be assigned to any pointer variable throughout the code.
- The old function cannot have a variable number of parameters.
- The old function cannot be a member of a class embedded into an inheritance hierarchy.
- The old function cannot be a template or overloaded function.

Even though this transformation is complementary to function insertion, its implementation is significantly more complex, because the function after expulsion cannot access private and protected members of the class. We deal with this problem by replacing direct access to such members with access through public access functions.

Function encapsulation encapsulates a sequence of statements into a new function. The user selects a block of code for encapsulation and the tool decides 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. In figure 4, the framed text area represents code selected for encapsulation. The result of the encapsulation is shown in figure 5.

In the selected block of code, all variables are classified into one of the following categories:

- Local variable
- Global variable
- Parameter passed by value
- Parameter passed by reference

The tool generates a *local variable* if the original variable does not carry any information into or out of the selected block. Such a 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 local variable must be written before being read inside the selected code, and if it is read in the outside code, it must be written in the outside code before it is read there.

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 selected code but any modifications to this variable in this block are not used outside the block. It also must be declared local or passed by value in the original code. The rest of the variables that do not qualify as local variables, global variables or value parameters 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. Also template functions are not supported. The transformation performed by this tool is similar to function encapsulation of [4,5,6].

```
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 4.

```
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 5.

Renaming is used to avoid naming conflicts if a function or variable is moved between different name scopes. The tool is composed of two components: the code analyzer and the text replacement tool. The code analyzer analyzes all of the identifiers in the given scope and makes sure that the new name does not conflict with any other identifier name (local, class member, global). In the second step, all old name occurrences are replaced by the new name.

Argument reordering reorders function arguments while preserving code execution if the order of the arguments is significant. The tool makes the following changes:

- Changes the order of the arguments in a function specification and in all forward declarations
- For function calls where the order of its arguments is significant, new auxiliary local variables are introduced. They are initialized in the original order and passed to the function in the new order.
- If a new auxiliary variable replaced an old variable's lvalue in a reference parameter, the old variable is updated after the function call.

This transformation cannot be used on overloaded functions. In some cases parameters are reordered using auxiliary variables even though their order is not significant. This inaccuracy, however, does not affect code execution.

The implementation of the transformations uses GEN++ [10] for analysis of the source code and C++ for the actual changes to the code. GEN++ pre-compiles the code and then parses the result. The parsed code is stored in an abstract semantic graph and queried by a LISP-like scripting language that GEN++ provides. Several problems were encountered during the implementation of the transformation tools.

Pre-compiler directives, particularly conditionally compiled blocks, often occur in C++ code. Figure 7 shows the resulting code configuration when the variable TEST is defined in the example in figure 6. Since the GEN++ parser pre-compiles before building the abstract semantic tree, 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 find all pre-compiler variables. 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 be changed 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 limitations for every transformation. The limitations concern unusual situations and did not occur in our case study.

The next two sections describe the use of the tools in restructuring scenarios. A restructuring scenario is a sequence of steps that are applied to a particular problem, where some steps are performed by the tools, while other steps are done by the

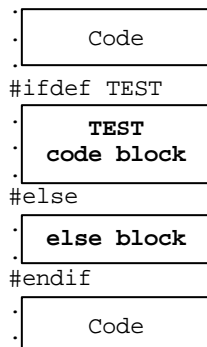


Figure 6.

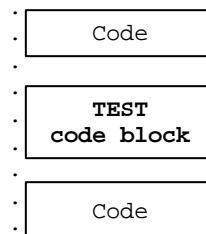


Figure 7.

programmer, who makes all key decisions.

3. Function Clones

Semantically equivalent function clones are functions that provide identical functionality. They can have different names, different order and names of arguments, and different names of local variables. Programmers who do not fully understand the system introduce these clones by the re-implementation of an already existing functionality. These clones are different from *function clones sharing common code*, which are created when programmers copy-and-paste an already implemented function and modify one of the copies. Because of the different nature of the two kinds of the clones, different scenarios are used for clone removal.

Semantically equivalent function clones are replaced by a single function in the following scenario:

1. Identify group of semantically equivalent function clones
2. Select one function from the group
3. Reorder the argument lists of the clones to match the selected function's argument list.
4. Replace all calls to the original clones with calls to the selected function
5. Delete the clones

In this scenario, step 1 is performed manually or with help of available clone detection technology [7,8,9]. Step 2 is performed manually. Step 3 is performed using the argument reordering tool. Step 4 is performed using automatic text replacement available on standard editors. In step 5, no longer used clones are deleted by standard editor.

When dealing with clone functions sharing common code, we use the following scenario:

1. Encapsulate common code into new functions. These functions are semantically equivalent clones
2. Replace semantically equivalent function clones by a single function as in the

```
void delete(Key key){
ListElement* eptr;
bool found;

eptr=head;
found=false;
while(eptr && !found)
{
if(eptr->data->key==key)
found=true;
else eptr=eptr->next;
}
if(eptr){
//code to delete
//item from the list
}
}
```

Figure 8.

```
void update(Key key) {
ListElement* eptr;
bool found;

eptr=head;
found=false;
while(eptr && !found)
{
if(eptr->data->key==key)
found=true;
else eptr=eptr->next;
}
if(eptr) eptr->data->trigger();
}
```

Figure 9.

```
Item* get_item(Key key){
ListElement* eptr;
bool found;

eptr=head;
found=false;
while(eptr && !found)
{
if(eptr->data->key==key)
found=true;
else eptr=eptr->next;
}
if(eptr) return eptr->data;
else return NULL;
}
```

Figure 10.

```

void delete(Key key) {
ListElement* eptr;

search(key,eptr);
if(eptr){
    // code to remove
    // the proper item
    // from the linked list
}
}

void search(Key key,ListElement* &eptr){
bool found;

eptr=head;
found=false;
while(eptr && !found){
    if(eptr->data->key==key)
        found=true;
    else eptr=eptr->next;
}
}

```

Figure 11.

```

void update(Key key){
ListElement* eptr;

search(key,eptr);
if(eptr) eptr->data->trigger();
}

Item* get_item(Key key){
ListElement* eptr;

search(key,eptr);
if(eptr) return eptr->data;
else return NULL;
}

```

Figure 12.

previous scenario.

This scenario does not change the clients of the clone functions and therefore no change to the client code is necessary.

An example of a scenario with three function clones is shown in figures 8 through 12. Clones in figures 8, 9, 10 update items stored in a linked list and share the code that searches the list. The shared code is encapsulated in the function `void search(Key,ListElement*)`. The resulting code after the scenario is shown in figures 11 and 12.

3. Class Clones

Class clones are classes with identical or near identical code. Similarly to function clones, we classify them as class clones representing almost identical concepts, or class clones representing different concepts but sharing a code.

Class clones representing almost identical concepts share implementation of function and data members. They are replaced by a single class, which contains union of all class members. The following scenario describes how a single class replaces a group of class clones:

1. Identify group of class clones.
2. Select one class as the target class.
3. Rename members of the other clones so that they have identifiers identical to the matching members of the target class.
4. Copy data members from the other clones into the target class so that the data set is the superset of all of the clones.
5. Expulse unique functions from the other clones.
6. Replace all instances of the clones by the target class instances.

```

class List1 {
public:
    void    Insert(Item*);
    void    StoreAtEnd(Item*);
    void    StoreAtBeginning(Item*);
    bool    Delete(key);
    Item*   Find(key);
    bool    Empty();
protected:
    eptr*   head;
    eptr*   tail;
};

```

Figure 13.

```

class List2 {
public:
    void
    InsertAtCurrent(Item*);
    Item* Current();
    bool    Remove();
    void    First();
    void    Advance();
    bool    Empty();
protected:
    eptr*   head;
    eptr*   tail;
    eptr*   current;
};

```

Figure 14.

7. Insert all the functions expulsed in step 5 into the target class.
8. Delete no longer used clones

The rename tool supports step 3. The insertion and expulsion tools support steps 5 and 7. Step 6 is achieved by simple text replacement available in standard text editors. Unused clones are removed in step 8 by standard editor.

We demonstrate the scenario on the example in figures 13 through 17. The group of clones is represented by two classes `List1` and `List2` that both provide variations of list functionality, see figures 13 and 14. Both `List1` and `List2` share the implementation of function `Empty()` and the data members `head` and `tail`. Class `List2` provides functions for iterative access to its items while `List1` provides functions to support the ordered list. Both classes share implementation algorithms.

In the step 2 of the scenario, `List2` is selected to replace `List1`, because its data members are a superset of the data members of `List1`. This simplifies the scenario because we can now avoid step 4. In step 5 all of the unique functions of class `List1` are expulsed using the expulsion tool, and the result is shown in figure 15. In step 6 all of the occurrences of class `List1` are replaced by `List2` (see figure 16). In step 7, all of the former functions of class `List1` are inserted into class `List2` using the function insertion tool. This completes the scenario and the final specification of class `List2` is

```

class List1 {
public:
    bool    Empty();
protected:
    eptr*   head;
    eptr*   tail;
};

```

```

void    Insert(Item*,List1);
void    StoreAtEnd(Item*,List1);
void    StoreAtBeginning(Item*,List1);
bool    Delete(key,List1);
Item*   Find(key,List1);

```

Figure 15.

```

void    Insert(Item*,List2);
void    StoreAtEnd(Item*,List2);
void    StoreAtBeginning(Item*,List2);
bool    Delete(key,List2);
Item*   Find(key,List2);

```

Figure 16.

shown in figure 17. As the class `List1` is no longer used in the code it can be safely

```

class List2 {
public:
    void    InsertAtCurrent(Item*);
    Item*   Current();
    bool    Remove();
    void    First();
    void    Advance();
    bool    Empty();
    void    Insert(Item*);
    void    StoreAtEnd(Item*);
    void    StoreAtBeginning(Item*);
    bool    Delete(key);
    Item*   Find(key);
protected:
    eptr*   head;
    eptr*   tail;
    eptr*   current;
};

```

Figure 17.

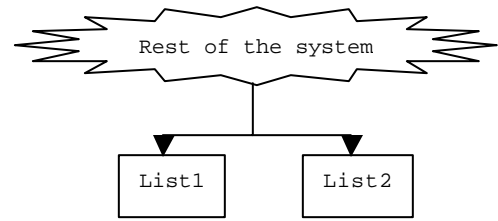


Figure 18.

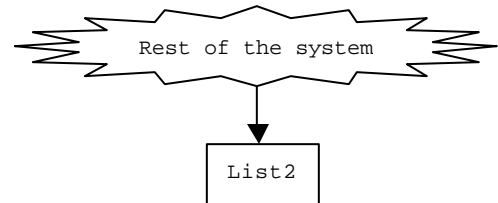


Figure 19.

deleted. The original architecture is in figure 18 and the resulting architecture is in figure 19.

An advantage of this scenario is that it replaces a group of clones with a single class and thus simplifies the class dependencies. A disadvantage is the fact that the scenario can create a class that does not represent a concept but is merely a coincidental union of the functionality of the original clones.

Class clones representing separate concepts are classes that share some code, but it is not appropriate to replace them by a single class because such class would not represent any concept. In this case the shared code is factored out into a new class. Each clone is then replaced by the composition of two classes, one of them being the new class containing all of the common code. The following scenario is used:

1. Identify the common code for a group of clones
2. Rename the members so that matching members have identical identifiers
3. For each clone, create a new class as a component of the original class
4. Move shared data members to the new component class
5. Move shared functions to the new component class
6. Eliminate identical clones, as in the previous scenario

Steps 4 and 5 are identical with scenarios for restructuring misplaced code, which we described in detail in [3]. Both scenarios used in steps 4 and 5 not only move data and functions into a component class but also apply compensating changes to all clients that were using moved members.

We demonstrate this scenario on the same example as the previous one. The classes `List1` and `List2` were selected in step 1. In step 2 two new classes were created: `BaseList1` and `BaseList2`. Figures 19 and 21 show the classes `List1` and `List2` and their new components after the application of steps 3 and 4. The final architecture after the application of step 5 is shown in figure 22.

```

class List1{
public:
    void        Insert(Item*);
    void        StoreAtEnd(Item*);
    void        StoreAtBeginning(Item*);
    bool        Delete(key);
    Item*       Find(key);
    BaseList*   list;
};

class BaseList1{
public:
    bool        Empty();
    eptr*       head;
    eptr*       tail;
};

```

Figure 20.

```

class List2{
public:
    void        Store(Item*)
    Item*       Current();
    bool        Remove();
    void        First();
    void        Advance();
    BaseList2*  list;
protected:
    eptr*       current;
};

class BaseList2{
public:
    bool        Empty();
    eptr*       head;
    eptr*       tail;
};

```

Figure 21.

This scenario is more likely to create cohesive classes implementing specific concepts. A disadvantage is that the new classes it creates complicate the dependency graph. It also makes all of the moved data and functions public and therefore it violates the encapsulation rules. The proper encapsulation can be restored by additional transformations, which are not currently supported by our tool-set.

The scenarios presented in this section were applied to clones on the project PET described in the next section.

5. Case study: Project PET

PET is a CAD tool developed at Ford Motor Company [2] to support the design of the mechanical components (transmission, engine etc.) of a car. 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 composed of an engine block, pistons, shafts etc. Each component is characterized by a set of parameters

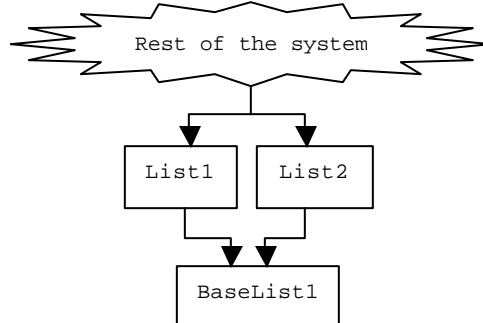


Figure 22.

```

//PET code . . .
#ifdef DEBUG
cout<<"too many lines"<<endl;
#endif
int indx;
for(indx=0;indx<20;indx++){
    delete []return_str[indx];
    return_str[indx]=0;
}
for(indx=20;indx<MAX_LINES;indx++){
    return_str[indx-20]=return_str[indx];
}
for(indx=MAX_LINES-20;indx<MAX_LINES;indx++){
    return_str[indx]=0;
}
numlines=MAX_LINES-20;
//PET code . . .

```

Figure 23.

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, divided into 200 files and structured into approximately 80 classes and 50 global functions. It is interfaced with other CAD software, including optimization software and 3-D modeling software.

Ford engineers use PET to support transmission design. Therefore all changes to PET were performed as quickly as possible in order to make the new functionality available. This situation prevented conceptual changes to the architecture, and the architecture progressively deteriorated. The introduction of the clones was one of the symptoms of this deterioration. During the code review we identified 10% of the PET code as clones (both class and function clones). The longest clone found had 66 lines and the smallest clone found had 6 lines of code.

```

key = (int)pkey;
#ifdef DEBUG
cout <<"set_toggle_button"<<endl;
#endif
if(key == 0){
    XtVaSetValues(toggle_button,XmNset,
                  True,NULL);
}
else{
    if(apDtoggglew_list != NULL) {
        tb=(Widget)apDtoggglew_list->get_first();
        count=1;
        while(tb!=NULL && count!=key){
            tb = (Widget)apDtoggglew_list->get_next();
            count++;
        }
        if(tb!=NULL){
            XtVaSetValues(tb, XmNset, True, NULL);
        }
    }
}
}

```

Figure 24.

In the first phase of the case study, the code comprehension step was performed, because it is essential for clone detection. The code fragment shown in figure 23 deletes the first twenty elements from the `return_str` array, where each array element represents a text line. This fragment repeats fourteen times in PET. Another example of function clones in PET is the function `intersect` that returns the intersection of two linked lists. This function is defined in two clones in PET:

- `Linked_List* intersect(Linked_List*, Linked_List*)`
- `Linked_List* LinkedList::intersect(Linked_list*)`

A simple linear search algorithm is implemented in the fragment shown in figure 24. The code fragment contains a linear search algorithm that finds an item placed in position `key` in the link list `apDtogglew_list`. There are eight function clones in PET, all implementing this algorithm. The algorithm requires the items to be stored in a certain order in the list. If during the maintenance the code is changed so that the order is no longer guaranteed, the search algorithm must be also changed so that the item can be located on different basis, for example by its name. In that case, all eight clones must be visited and correctly updated.

In the second phase of the case study, the actual clone removal was performed in the inference engine subsystem of 10,000 lines. The clones were classified into different groups and the appropriate scenario was applied to each group. In the case of function clones we successfully removed exact duplicates and clones that perform identical operations on different data. However we also located clones that contain common code but differ from each other by an embedded code sequence, expression, or different data type. Such clones cannot be removed using our current tools, because the common code in these clones cannot be easily replaced by a single C++ function. A new and complicated set of tools would be required for the removal of these clones. We found that these clones constitute less than 2% of the inspected code, and therefore we decided to leave the elimination of these clones to the programmer. Please note that in a related

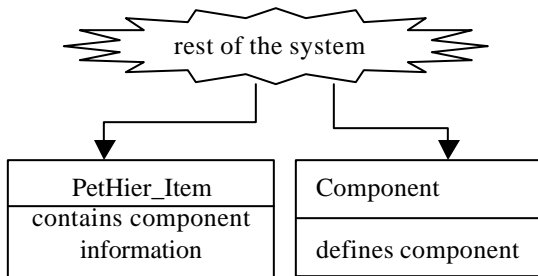


Figure 25.

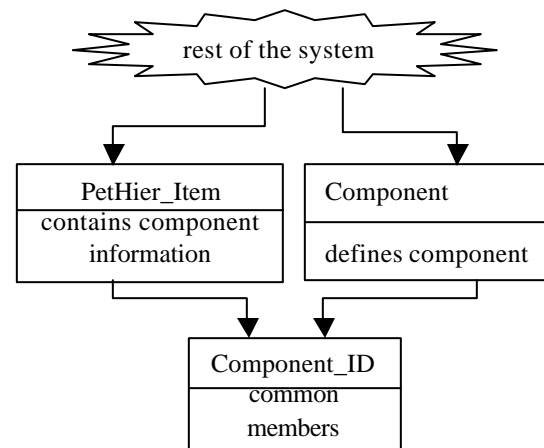


Figure 26.

research, Baxter [8] replaces such clones by C++ macros and passes the different code parts as text replacement arguments. This approach works well, but we believe that the

overuse of macros may produce hard-to-read and hard-to-analyze code that will create difficulties in future comprehension and maintenance.

In the case of class clones, we were able to merge all of the clones with identical functionality and extract the common parts of classes that share common code. The class clones of PET are not exact duplicates and contain extra data or function members. An example is in Figure 25 where there are two classes describing a mechanical component: `PetHier_Item` and `Component`. Data contained in `PetHier_Item` is also present in the class `Component`. The same information is kept in two places and must be consistently updated and maintained. In order to remove the clone, we factored out the common data and members into a separate `Component_ID` class (see figure 26).

6. Related Work

Most of the related research is focused on clone detection and the assessment of the benefits of clone removal rather than on clone removal itself. The methods for clone detection are based on two principles: string-based [9] and Abstract Syntax Tree (AST) based [1,7,8]. Because of the differences in the definition of clones, they report slightly different percentage of clones in code.

The string based method [9] detects exact and near-exact clones. This method uses a lexical analyzer to detect clones that differ only by blanks and comments. The authors reported that 13-20% of the code in large-scale applications consists of clones.

Research in [1,7] parses code into AST that is later converted into an intermediate representation language, and 21 different metrics are computed. The metrics characterize the functions in the following categories:

- Name
- Layout
- Expressions
- Control flow

Based on the metrics two kinds of clones are identified: `ExactCopy` and `DistinctName` clones. Using this method [7], the authors found that clones represented 5-10% of code in a large software project. However, only exact duplicate functions and distinct name duplicate functions were considered.

The implications of duplicated code for software maintenance and reuse were investigated in [15]. The study was conducted on legacy COBOL code. The results reported suggest that code duplication significantly increases code size, and it also increases de-localization of the same functionality throughout the code. In addition, authors attributed numerous errors in the code to the cut-and-paste techniques that were responsible for the duplication of the code. The authors also investigated the relation of duplicated code and reuse. They argue that code replication is not reuse. Code replication (unlike reuse) increases cost and decreases the quality of the software. On the other hand, replicated code suggests a potential for reuse.

In parallel with our work, another interesting work [8] on clones was published. In [8], a parser processes the source code and produces an AST. In order to compare different sub-trees of the parsed code, a hash function is computed and only sub-trees hashed into the same bucket are compared. This method is able to identify a broad group of near-miss clones. Using the described clone detector, the authors reported a 13% clone content in a large software project. The authors also provide a simple clone removal tool that replaces clones with pre-processor macros.

The high level editing tools for C++ code restructuring presented in this paper, are related to previous research [4,16,17,6,13].

Johnson and Opdyke [16,17] study the 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 there are transformations similar to our function encapsulation transformation. They also propose a number of complementary low level transformations. Refactorings proposed in their research were embedded into the Smalltalk Refactory Browser [18]. The Refactory is an advanced browser that in addition to standard functions performs some of the refactorings. Among the features implemented in the Refactory is a drag-and-drop function that can be used to drop methods on classes or protocols.

Griswold investigated meaning-preserving transformations in the block structured language Scheme [4]. He defined meaning-preserving transformations on a program dependency graph that was extracted from the source code. Transformations were defined by graph transformation rules and they manipulated statements within one block. The transformations included:

- adding a new variable to store an intermediate result,
- moving an assignment statement into or out of a conditional block,
- replacing an expression with another equivalent expression,
- renaming a local variable.

Lakhotia [6,13] presents a collection of evolutionary transformations. The transformations are defined on a procedural language without global variables. The language contains an assignment statement, a branch statement, and a function call statement where function call cannot be used in expression. The "Fold" transformation is similar to our function encapsulation tool, as it creates a function from a set of statements. The transformation set also contains transformations that can bring together non-continuous regions of code.

Blaha and Premerlany [11] give a catalog of transformations that can be used for object-oriented database restructuring. They use OMT notation [12] for transformation specifications, which makes the transformations independent of a specific programming language or database.

7. Conclusion

From our experience with PET reengineering and from the findings of other research groups [7,8,15] we infer that clones in code are a serious problem, and that they are not considered as a type of reuse [15]. Our experience shows that the negative factors of duplicated code include the following:

- Clones add unnecessary code to the system, which must be maintained and understood.
- If a clone is changed it is likely that identical changes must be performed in the other clones as well.
- Code polluted with clones has a complicated and confusing architecture.
- Cut-and-paste techniques are error prone.

Data reported by [7] suggest that programmers in industry are aware of the problem and that they perform clone removal occasionally. However we feel that specialized support for these activities is important. This support must include both clone detection and clone removal technology. The experience gained in the case study suggests that the advantage offered by the support increases with code size, because the length of the code that has to be scanned for potential ripple effects also increases.

However, the tools must be used by an experienced programmer who makes important decisions, for example which functions are clones. The code after clone removal can still contain residual imperfections. In our future work we want to focus on the improvement of the analyzers used for the restructuring, so that the tools could handle more complicated situations and produce better code. For frequently repeated actions we want to implement additional tools to reduce human intervention in the restructuring process.

In our approach, the scenarios and tools for clone removal are very similar to the scenarios and tools for moving misplaced data and code [3]. When a new need for restructuring arises, the tools can be reconfigured into new scenarios. This gives us hope that a practical set of transformation tools for code restructuring can be gradually developed. This set would be suitable for all restructuring tasks in an object-oriented code.

Acknowledgements

We want to acknowledge the support provided by Ford personnel, particularly Garry Vrsek, Tony Mikulec, Libor Soucek and Xianren Li.

References:

- [1] J. Mayrand, C. Leblanc, E. M. Merlo: Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics, IEEE, 1996

- [2] T. Mikulec, X. Li, G. Vrsek: Powertrain Engineering Tool an its application. XXVI Congress FISITA, Prague 1996.
- [3] V. Rajlich, R. Fanta: Reengineering Deteriorated Object-Oriented Code, Proc. IEEE Int. Conf. On Software Maintenance, 1998.
- [4] William G. Griswold. Program Restructuring as an Aid in Software Maintanance. PhD thesis, University of Washington, 1991.
- [5] Opdyke, W.F., Refactoring Object-Oriented Frameworks, PhD thesis, University of Illinois at Urbana-Champaign, 1992.
- [6] Arun Lakhotia: DIME: A direct manipulation environment for evolutionary development of software, In proceedings of IEEE IWPC'98, 1998
- [7] B. Laguë, D Proulx, J. Mayrand, E. M. Merlo, J. Hudepohl: "Assesing the Benefits of Incorporating Function Clone Detection in a Development Process", IEEE International Conference on Software Maintenance, 1997.
- [8] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, L. Bier: "Clone Detection Using Abstract Trees", IEEE International Conference on Software Maintenance, 1998.
- [9] B. Baker: "On Finding Duplication and Near-Duplication in Large Software Systems", IEEE Working Conference on Reverse Engineering, 1995.
- [10] P. Devanbu, "GENOA A Customizable, Language- and Front-end Independent Code Analyzer", In Proceedings of ICSE '92, 1992, pp. 307-317.
- [11] M. Blaha, W. Premerlany, "A Catalog of Object Model Transformations", Proceedings of WCRE '96, 1996, pp 87-96.
- [12] J. Rumbaugh, M. Blaha, W. Premerlani, W. Lorensen, "Object-Oriented Modeling and Design", Prentice-Hall, 1991.
- [13] A. Lakotia, J-C Deprez: Restructuring programs by tucking statements into functions", Information and Software Technology, 1998.
- [15] E. Burd, M. Munro: "Investigating the Maintenance Implications of the Replication of Code", IEEE International Conference on Software Maintenance, 1997.
- [16] R. E Johnson, W. F. Opdyke, "Refactoring and Aggregation", In Proceedings of ISOTAS'93: Object Technologies for Advanced Software, Lecture Notes in Computer Science, vol.742, Springer-Verlag, November 1993, pp. 264-278.
- [17] R. E Johnson, W. F. Opdyke. "Creating Abstract Superclasses by Refactoring", In Proceedings of CSC'93, The ACM Computer Science Conference, February 1993, pp. 66-73.