# Refactoring Model of Legacy Software in Smart Grid based on Cloned Codes Detection

**Fanqi Meng[1], Zhaoyang Qu[2] and Xiaoli Guo[3]**

**[1] School of Information Engineering, Northeast Dianli University,Jilin, Jilin 132012, China**

**[2] School of Information Engineering, Northeast Dianli University, Jilin, Jilin 132012, China**

**[3] School of Information Engineering, Northeast Dianli University, Jilin, Jilin 132012, China**

**Abstract**

The construction of smart grid relies on the development of many new software systems, whereas it would be very expensive and time-consuming if these new software systems are completely developed anew. Since the existence of many legacy software systems in the former power grid, the problem may be solved well supposing that those legacy software systems are reused reasonably and efficiently in the construction of smart grid. In view of this situation, a refactoring model of legacy software is proposed. The model is based on reverse engineering and its kernel is cloned codes detection and components extraction. Firstly, the cloned codes in the scanned source code of the legacy software will be detected by means of CCFinder. Secondly, the abstract syntax trees of the functions which include the cloned codes will be created. Thirdly, the degree of variation between the functions which include the cloned codes belonging to the same clone set will be calculated according to their abstract syntax trees, and then some functions whose similarities of abstract syntax trees are in the allowed range will be combined. Finally, the combined functions and other frequently invoked functions will be encapsulated in a new class (or a DLL file), and all of these classes (or DLL files) will be reused as components in the development of new software systems of the smart grid.

***Keywords:*** *Smart Grid, Legacy System, Code Clone, Refactoring*

## 1. Introduction

Although the term “smart grid” has been used since at least 2005 [1], it still hasn’t a uniform definition in the entire world. However, all of the countries consider smart grid as the inevitable trend of the development of electrical grid, so that smart grid has another name called “electrical grid 2.0”. A smart grid is an advanced electrical grid that can gather and process the information by using computers and other technology. The gathered and processed information has widely resources, ranging from the behaviors of suppliers and consumers to the status of devices running in smart grid. The processing of the collection and computation of the information should be in an automated fashion, in order to enhance the efficiency, reliability, economics, and sustainability of the supply of electricity [2]. The above background and the features (especially for efficiency and reliability) of smart grid decide that its construction relies on the development of many new software systems (such as monitoring software, controlling software, marketing software, etc.) to gather and process the information. However, it must be very expensive and time-consuming if these new software systems are totally developed anew. The cost and deadline of the task must be considered under the circumstances. Thus, an efficiency software development mode for smart grid is imperative.

Legacy software usually is a large-scale and complex software system which has run for a long time (more than 20 years) [3]. Since the development language of legacy software mostly is the third or early programming language (such as ASM, COBOL or Turbo C etc.), and the development framework of legacy software has been outdated, the legacy software is hardly to be maintained and evolved. Even if it is no longer used, legacy software may continue to impact the organization due to its historical role. Most functions in legacy software are stability and credibility in processing the existing business, thus the method that reuses these functions in developing new software systems which can handle both existing business and emerging business has been adopted by many programmers. The smart grid commonly has many legacy software systems which had been used by former electrical grid. Using them efficiently in the construction of a smart grid is potentially the solution to the above problem (expensive and time-consuming). So the study of the method to efficiently use legacy software is significant to the construction of smart grid.

Refactoring is a programming technique for optimizing the structure or pattern of an existing body of code by altering its internal nonfunctional attributes without changing its external behavior[4][5][6][7]. By applying a series of "refactorings", the software can obtain some advantages,

including improved code readability and reduced complexity to improve the maintainability of the source code, as well as a more expressive internal architecture or object model to improve extensibility. The code refactoring of legacy software is one of basic methods to achieve efficient reuse. This paper presents a study of how to reuse the legacy software by means of refactoring. The remainder of the paper is organized as follows. Section 2 gives an overview of the related work in the area. The refactoring model of legacy software is proposed in Section 3. And its kernel processes, cloned codes detection and components extraction are presented in Section 4. Section 5 analyzes the results obtained in a number of experiments and Section 6 outlines the conclusions and future work.

## 2. Related work

Software reuse is still a popular research issue in the field of software engineering. The modernization of legacy software is an important research direction of software reuse. The evolution of software systems can be divided into three types: maintenance, modernization and replacement. Maintenance can only meet the small changes of the requirements by correcting and enhancing the functions of the software system. Replacement has high risk and will cost long time to develop new system. Replacement will not happen unless the system can't be maintained or modernized. So the modernization of legacy software is now regarded as the most feasible method in software reuse. There already exist several modernization methods to deal with legacy systems, for reusing them in the development of new systems. These methods mainly fall into three categories: Redevelopment, Wrapping and Migration.

### 2.1 Redevelopment

Redevelopment is a high-risk and low-reuse method, almost abandons whole codes of the legacy systems. Since the methods of this kind realize the functions of the legacy systems in the new system via programming anew, they are usually used to eliminate the structural flaws of legacy systems. CORUM (Common Object-based Re-engineering Unified Model), CORUMⅡ, MARMI-RE and OSET are all the methods belonging to redevelopment[8][9][10].

### 2.2 Migration

Wrapping methods can be classified into three types: UI-based wrapping (UI, user interface), data-based wrapping and function-based wrapping, according to the wrapped contents. UI-based wrapping reuses the UIs of the legacy system in the new system by interface mapping. Data-based wrapping includes the means, such as DB (data base) gate, XML and data copy etc. Data-based wrapping inherits the data structure of the legacy system, so the data of the legacy system can be used in the new system. Function-based wrapping uses component wrapping, object wrapping and gate wrapping etc. to realize the reusing of the service logic. These wrapping methods can reuse legacy systems for a short time, but they will increase difficulties in the maintenance and management of the new system.

### 2.3 Migration

The migration of legacy systems usually divides into two types: component-based migration and system-based migration. Component-based migration classifies the legacy system into independent components, and then migrates the components singly. System-based migration integrates the whole legacy system and its data into the new system. Representative methods and models of migration are Chicken Little, Butterfly, SGF and AGRIP etc. Migration merely suits for small-scale legacy systems, since it is more possible for losing information if the scale of the legacy system is larger.

## 3. Refactoring model

The refactoring of legacy software is a process to reengineering the old software system by component technology. This process can be roughly divided into two steps: The first step is reverse engineering. Reverse engineering is the process of analyzing a subject system to create representations of the system at a higher level of abstraction [11]. It can also be seen as going backwards through the development cycle. Reverse engineering often involves taking computer program apart and analyzing its workings in detail to be used in maintenance, or to try to make a new program that does the same thing without using or simply duplicating (without understanding) the original. The second step is forward engineering. Forward engineering has the process similar to conventional development of software. It follows the flow: requirements analysis, outline design, detailed design, testing and modification. Figure 1 shows the refactoring model that is built to reengineer the legacy software in smart grid based on component extraction, update and reuse.

In this model, firstly, Requirement Change leads to the Architecture Readjustment of legacy software system; Then, Architecture Readjustment needs Component Update to provide new components; Finally, Component Update helps Software Refactoring coming true. On the stage of requirement analysis, according to the change of requirement, Requirements Analysis Engineers increase

new requirements or delete useless requirements based on the result of Requirement Analysis that comes from the reverse engineering of legacy software.

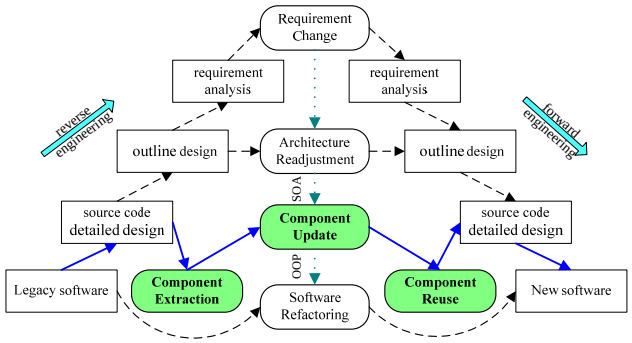


Fig. 1 The refactoring model of legacy software in Smart Grid

On the stage of outline design, engineers readjust the architecture of legacy software to Service Oriented Architecture (SOA). On the stage of detailed design, programmers update component, including abandoning useless components and regaining new components, in order to make the components compatible with the demand of Architecture Readjustment and Object-Oriented Programming (OOP). Usually, three ways can be used to gain the needed components. The first one is to purchase from others; the second one is to renew development by yourself; the last one is to extract components from the source code of legacy software. The model we proposed adopts the third method to get components, so the Component Extraction in the model is both the beginning of refactoring process and the kernel method of the refactoring model.

## 4. Components extraction

The refactoring of legacy software is also known as Software Systems Modernization. Software Systems Modernization using SOAs and Web Services represents a valuable option for extending the lifetime of mission-critical legacy systems [12]. Components play an important role in SOA. Software engineers regard components as part of the starting platform for service-orientation. Actually, the refactoring model uses component-based software engineering (CBSE) as the forward engineering method. Figure 2 shows the refactoring process from component perspective.

### 4.1 Cloned codes detection

Copying code fragments and then reuse by passing with or without minor modifications or adaptations are common activities in software development. This type of reuse approach of existing code is called code cloning and the passed code fragment (with or without modifications) is called a clone of the original. For instance, Baker has found that on large systems between 13% - 20% of source code can be cloned code. For an object-oriented COBOL system, the rate of duplicated code is found even much higher, about 50% [13].

Although code clones may adversely affect the software systems' quality, especially their maintainability and comprehensibility, the cloned code in legacy software are potentially most valuable code to be refactored into new components. One piece of code is cloned more times, and then it has more reuse value. So we should detect cloned code before refactoring. In addition, cloned code detection will compress the length of source code in legacy software, and reduce the work load in component extraction.

We have used CCFinder to detect the cloned code in legacy software. CCFinder is a token-based cloned code detection tool [14]. The work principle of CCFinder is followed: First, each line of source code is divided into tokens by a lexer and the tokens of all source code are then concatenated into a single token sequence. The token sequence is then transformed. After that, each identifier related to types, variables, and constants is replaced with a special token. A suffix-tree based sub-string matching algorithm is then used to find the similar sub-sequences on the transformed token sequence where the similar sub-sequence pairs are returned as clone pairs/clone classes. Once the clone pair/clone class information is obtained with respect to the token-sequence(s), a mapping is required for obtaining the clone pair/clone class information with respect to the original source code. Figure 3 shows the detection interface of CCFinder 10.2.5.0 (download from http://www.ccfinder.net/)

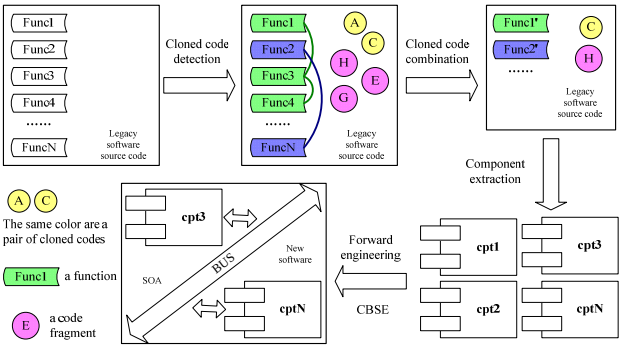


Fig. 2 The process of component extraction

### 4.2 Abstract syntax trees creation

Abstract syntax tree is a production generated after the lexical analysis and parsing of source code. Abstract

syntax tree fully reflects the grammatical structure of the source code, and its leaves represent identifier or constant etc. Figure 4 shows an abstract syntax tree of a code segment.

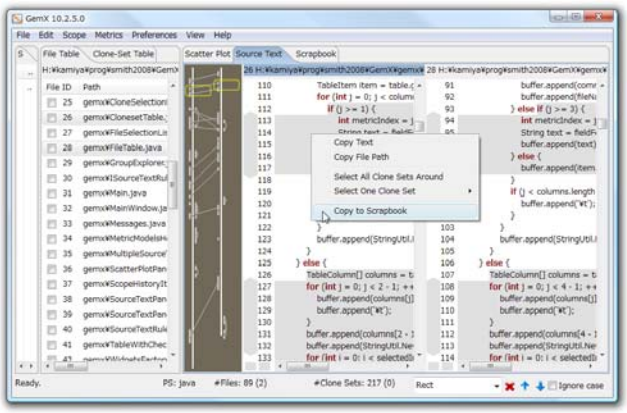

Fig. 3 The detection interface of CCFinder

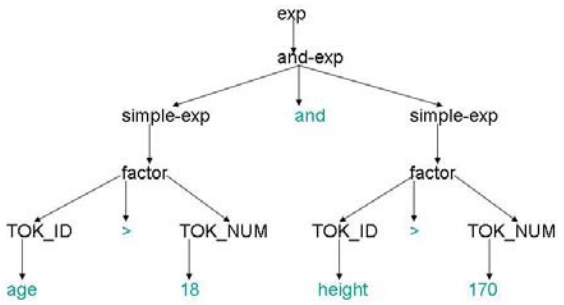


Fig. 4 An example of abstract syntax tree

Function is the basic unit in the third generation programming languages which are the main tools used in software development 20 years ago. Therefore the emphasis of reusing legacy software is functional refactoring for reuse. According to the refactoring model, the functions include cloned code which will be compared for calculating the similarity. Generally, the two functions with similar syntax structure are probably the same. So the abstract syntax tree of the functions includes cloned code which should be built after code clone detection. Various tools for building abstract syntax tree can be downloaded easily from Internet, for example The GNU Compiler Collection and JavaCC.

## 4.3 Differences degree calculation

The functions may be very different even though they have cloned code belonging to the same clone set. The relationship between the function and the cloned code probably has two cases. In **the first case**, the function is totally cloned (mainly exists between different legacy software). In **the second case**, the function includes the cloned code. Even though some large cloned code may also include functions, but the larger cloned code can finally be divided into the functions totally cloned and the small cloned code pieces included by functions. In addition, the relationship between cloned codes which are detected by CCFinder also has three cases. In **case one**, the two cloned codes are the same. In **case two**, they are merely different in some identifiers' name. In **case three**, they may be minor different in variable types or syntax. All above cases happened because the detective method of CCFinder is token-based.

### 4.3.1 Totally cloned

If the function is totally cloned according to the detection result presented by CCFinder (it still has three cases discussed above), then we should calculate its difference degree with other related cloned functions by traversing abstract syntax tree twice. We directly compare the value of the node of the abstract syntax trees with the functions in the first traversing. If the result shows that the abstract syntax trees are the same, it means that the two functions are the same (note it as **Type A**). If not, we change all customer identifiers into $ when traversing the abstract syntax trees. If the result shows the same, the two functions are merely different in some identifiers' names (note it as **Type B**); else they are different in variable types or others (note it as **Type C**).

### 4.3.2 Partly cloned

If the function is partly cloned, it means that the function includes cloned code in its body. We traverse the abstract syntax tree of the function with changing customer identifiers into $. Those functions which include the cloned code belonging to the same clone set will be compared with traversing results. We adopt Levenshtein Distance (or Edit Distance) for the compare method. The algorithm of the calculation of Levenshtein Distance between the two string fp1 and fp2 is shown as follows [15]. The different degree between two functions can be gotten via calculating the expression: ***DD= (matrix (len1, len2)/max (len1, len2))*100%***. The bigger the value of DD is, the more different the two functions are. We can use a threshold value to decide whether the functions are similar. If the value of DD is below the threshold value, note the two functions as **Type D**, else noted them as **Type E**.

## 4.4 Cloned functions combination

A cloned function is a function with cloned code. The combination of cloned functions can be divided into five cases according to the types of cloned function.

```
1: len1 ← strlen(fp1)
2: len2 ← strlen(fp2)
3: initialize_two_dimensional_matrix(matrix, len1, len2)
4: for i = 0 → len1 do
5:    for j = 0 → len2 do
6:       if fp1[i] = fp[j] then
7:          cost = 0
8:       else
9:          cost = 1
10:      end if
11:      matrix[i, j] = min(matrix[i−1, j]+1, matrix[i, j−
         1] + 1, matrix[i − 1, j − 1] + cost)
12:   end for
13: end for
14: return matrix(len1, len2)
```

Fig. 5 The algorithm of Levenshtein Distance calculation [15]

Case 1, the functions which will be combined are Type A. In this case, all of the functions are the same, so we select one of them and add it into function base.

Case 2, the functions which will be combined are Type B. In this case, all of the functions are very similar except individual identifier's name, so we select the shortest one for saving space and add it into function base.

Case 3, the functions which will be combined are Type C. In this case, the differences between the functions are in types of variable or in other aspects, so we select the longest one for retaining enough information and add it into function base.

Case 4, the functions which will be combined are Type D. In this case, even though the difference degree is lower than a preset threshold value, the functions are more different with each other than Type A, Type B and Type C. so we should flexibly adopt various existent refactoring method to combine the functions. The combined function will be added into function base.

Case 5, the functions are Type E. Since the similarity is too low, the functions are not recommended to combine. All of the functions are respectively added into function base.

All of the functions in the function base have a value which denotes invoked times in legacy software. The value is an important reference for component extraction. In Case 1 to Case 4, the invoked time of a combined function is the sum of invoked times of all related cloned functions.

### 4.5 Component extraction

A component may be a software package, a Web service, or a module that encapsulates a set of related functions (or data). Programmers can use these functions which have been stored in function base as various forms. For example, some functions can be encapsulated into a new class as function members by tiny modification, or assembling some functions to generate a DLL files. In addition, several tools have been used in extracting components, such as CodeMiner, CARE (Computer-Aided Reuse Engineering) and PATRicia (Program Analysis Tool for Reuse) [16].

## 5. Analysis and result

We did some experiments about cloned code detection which is the basic work in the model. We selected the former versions of Cook, Snns, Weltab and Postgresql as our experimental subjects. The four applications were both written in C or C++. We used CCFinder to detect cloned codes hidden in the four applications. Before the detection, the value of Minimum Clone Length was set at 120 and the value of Minimum TKS was set at 30. These values were more suitable for two reasons: firstly, the codes will be no much reuse value if its length is shorter than 40 characters and 10 TKS; secondly, the cloned codes usually do not have a length longer than 200 characters and 50 TKS. The metric results are shown as Table 1, Table 2 and Table 3.

Table 1: File metrics

| ***Name*** | ***Min.*** | ***Max.*** | ***Average*** |
|---|---|---|---|
| LEN | 1 | 33586 | 1000.28 |
| CLN | 0 | 11 | 0.128866 |
| NBR | 0 | 11 | 0.181701 |
| RSA | 0 | 1 | 0.056779 |
| RSI | 0 | 0.51 | 0.008532 |
| CVR | 0 | 1 | 0.064539 |
| RNR | 0.024 | 1 | 0.900082 |

The meaning of the Names in the tables can be found at http://www.ccfinder.net/doc/10.2/en /tutorial-gemx.html

Table 2: Clone set metrics

| ***Name*** | ***Min.*** | ***Max.*** | ***Average*** |
|---|---|---|---|
| LEN | 123 | 3220 | 732.071 |
| POP | 2 | 12 | 3.38571 |
| NIF | 1 | 12 | 2.85714 |
| RAD | 0 | 6 | 1.2 |
| RNR | 0.52 | 0.996 | 0.789677 |
| TKS | 30 | 53 | 34.5143 |

| | | | |
|---|---|---|---|
| LOOP | 0 | 17 | 3.35714 |
| COND | 2 | 48 | 14.4714 |
| McCabe | 3 | 58 | 17.8286 |

Table 3: Line-based metrics

| *Name* | *Total* | *Min.* | *Max.* | *Average* |
|---|---|---|---|---|
| LOC | 432005 | 2 | 11618 | 278.354 |
| SLOC | 217849 | 0 | 5562 | 140.367 |
| CLOC | 12750 | 0 | 603 | 8.21521 |
| CVRL | - | 0 | 1 | 0.058527 |

From Table 3, we know that the four applications have 432,005 lines in their source files, and 217,849 lines including at least one token. In other words, 217,849 lines are executable codes. 12,750 lines are cloned codes. The ratio of the lines including cloned codes is 0.058527. The ratio is lower than common case because of the bigger preset value for Minimum Clone Length and Minimum TKS. According to these results, we found some cloned functions which can be treated as reusable component candidates. Figure 6 shows a pair of Type A cloned functions, named *next_token*. They are found in different files of the same application (hba.c and miscinit.c in postgresql). Figure 7 shows a pair of Type B cloned functions, named *TEST_JE_Backprop* and *TEST_JE_BackpropMomentum*. They are detected in the same file of same application (learn_f.c in snns). Besides the different function name, the two functions have another difference in line 5698 and 5743 (if condition, <3, <5). Figure 8 shows a pair of Type B cloned functions found in different files of different applications (Lex.yyz.c in snns and bootscanner.c in postgresql). More cloned functions are Type D or Type E, but we didn't further study these partly cloned functions in our experiments.

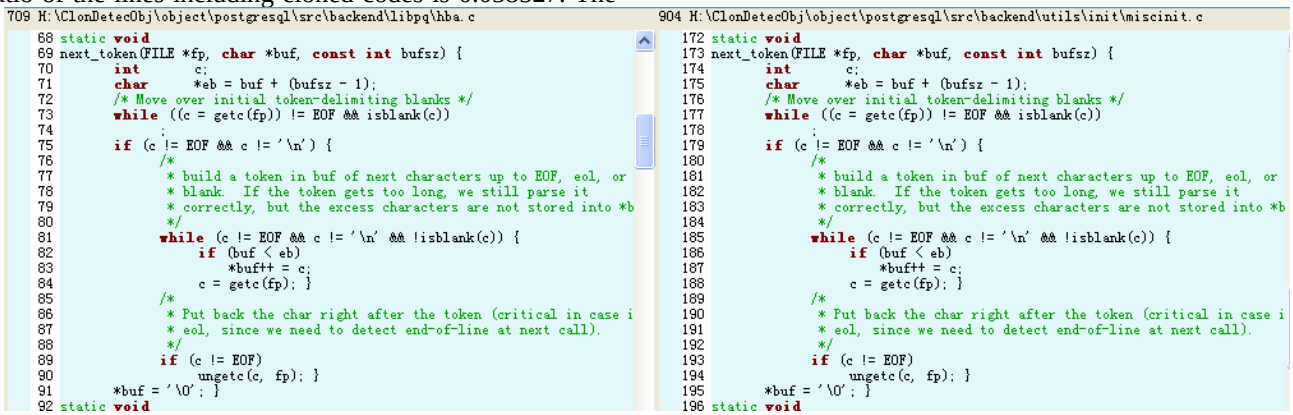


Fig. 6 An example of Type A cloned functions from different files of the same applications

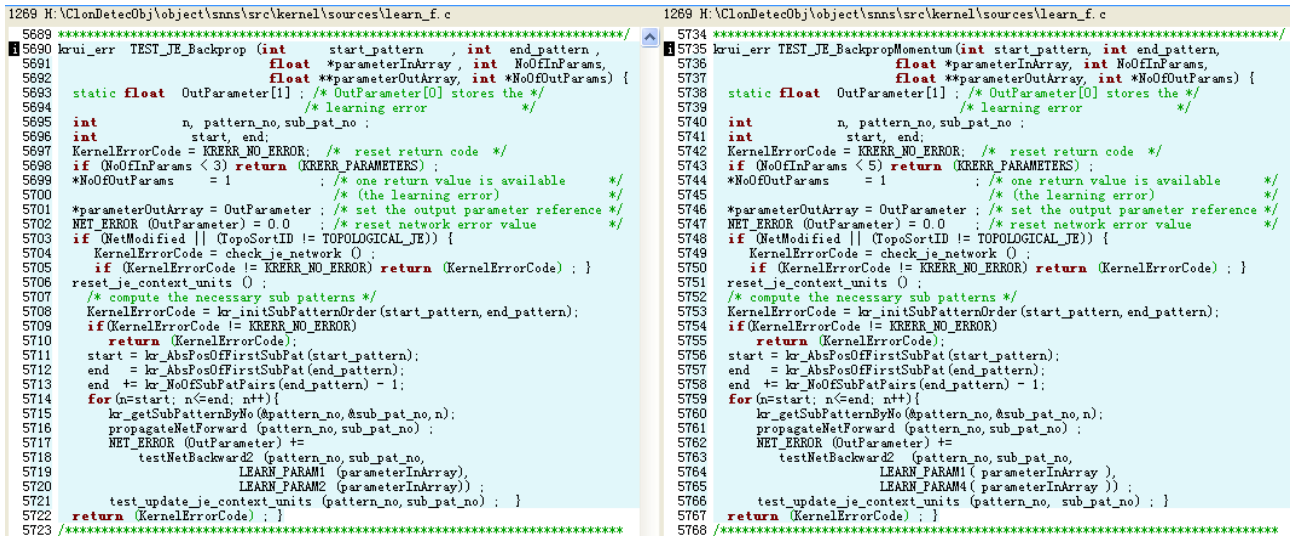


Fig. 7 An example of Type B cloned functions from a same file of a same application

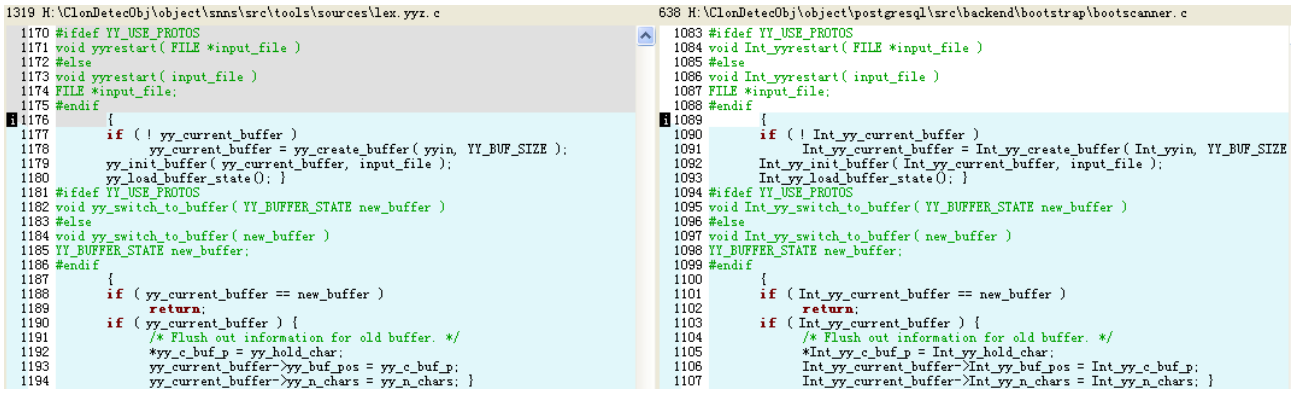


Fig. 8 An example of Type B cloned functions from different files of different applications

## 6. Conclusions

The legacy software that was used by former electrical grid may help the construction of smart grid in efficiencies and costs, but it depends on whether the legacy software can be reused. In order to reuse the legacy software efficiently, we proposed a refactoring model based on cloned code detection. By detecting cloned code, we can firstly reduce the candidates for component extraction, thereby lower the complexity; secondly, the remained cloned functions after function combination are more valuable for components generation, thus enhance the reliability of the refactoring. However, the result shows that the valuable cloned functions are not too much in legacy software, so the method of this model should be used as a subsidiary method in refactoring large-scale legacy software.

### Acknowledgments

This work was funded in part by Natural Science Foundation of China (Grant Number 51077010) and Natural Science Foundation of JiLin province of China (Grant Number 20101517).

## References

[1] S. Massoud Amin, Bruce F. Wollenberg, “Toward a smart grid”, IEEE P&E Magazine, vol. 3, no. 5, 2005, pp. 34 – 41.

[2] Amrita Dey, Nabendu Chaki, Sugata Sanyal, "Modeling Smart Grid using Generalized Stochastic Petri Net", JCIT, AICIT, vol. 6, no. 11, 2011, pp. 104 – 114.

[3] Wen-Shin Hsu, Jiann-I Pan, Hua Hu, "Exercise Prescription Monitoring System: Using Sensor Network and Service-Oriented Architecture", IJMIA, AICIT, vol. 2, no. 2, 2012, pp. 44 – 55.

[4] Nien-Lin Hsueh, Peng-Hua Chu, "A Pattern-based Refactoring Approach for Multi-core System Design", IJACT, AICIT, vol. 3, no. 9, 2011, pp. 196 -209.

[5] Ibrahim, Safwat M.”Identification of Nominated Classes for Software Refactoring Using Object-Oriented Cohesion Metrics”, International Journal of Computer Science Issues, v 9, n 2 2-2, p 68-76, 2012

[6] Ananda Rao, A. “Identifying clusters of concepts in a low cohesive class for extract class refactoring using metrics supplemented agglomerative clustering technique”, International Journal of Computer Science Issues, v 8, n 5 5-2, p 185-194, 2011.

[7] Arora, Madhulika. “Refactoring, way for software maintenance”, International Journal of Computer Science Issues, v 8, n 2, p 565-570, 2011

[8] Woods Steven, O'Brien Liam, Lin Tao, Gallagher Keith, “An architecture for interoperable program understanding tools”, 6th International Workshop on Program Comprehension, 1998, pp. 54 – 63.

[9] Rick Kazman, Steven G. Woods, S. Jeromy Carrière, “Requirements for Integrating Software Architecture and Reengineering Models: CORUM II”, WCRE '98 Proceedings of the Working Conference on Reverse Engineering, 1998, pp. 154 – 163.

[10]Eun Sook Cho, Jung Eun Cha and Young Jong Yang, “MARMI-RE: A Method and Tools for Legacy System Modernization”, Lecture Notes in Computer Science, Vol. 3647, 2006, pp. 42-57.

[11]Chikofsky, E. J.; Cross, J. H, "Reverse engineering and design recovery: A taxonomy", IEEE Software, vol. 7, no. 1, 1990, pp. 13–17.

[12]Gerardo Canfora, Anna Rita Fasolino. “A wrapping approach for migrating legacy systeminteractive functionalities to Service Oriented Architectures”, The Journal of Systems and Software,vol. 81, 2008, pp. 463 – 480.

[13]Dongxiang Cai, Miryung Kim, “An Empirical Study of Long-Lived Code Clones”, International Conference on Fundamental Approaches to Software Engineering, 2011, pp. 432–446.

[14]Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue, “CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code”, Transactions on Software Engineering, Vol. 28, no.7, 2002,pp. 654- 670.

[15]Wu Zhou, Yajin Zhou, Xuxian Jiang, "Detecting Repackaged Smartphone Applications inThird-Party Android Marketplaces", In Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy,2012,pp.120-130.

[16]MF Dunn, JC Knight, "Automating the detection of reusable parts in existing software", In Proceedings of the 15th international conference, 1993, pp.10-19.

**Fanqi Meng** is a member of ACM, and received his M.S. degree in computer science and technology from Northeast Dianlil University, Jilin, China, in 2010. He is now a lecturer in the School of Information Engineering, Northeast Dianlil University. His main research interest is cloned code detection and refactoring.

**Zhangyang Qu** received his Ph.D. degree in power system automation from North China Electric Power University, Bejing, China, in 2010. He is now a professor and tutor of postgraduates in the School of Information Engineering, Northeast Dianlil University. His main research interest is power system information processing.

**Xiaoli Guo** received her M.S. degree in computer science and technology from Changchun University of Science and Technology, Jilin, China, in 2006. She is now a professor and tutor of postgraduates in the School of Information Engineering, Northeast Dianlil University. Her main research interest is computer education.