Quality of the Source Code for Design and Architecture Recovery Techniques: Utilities are the Problem

Heidar Pirzadeh, Luay Alawneh, Abdelwahab Hamou-Lhadj
Department of Electrical and Computer Engineering
Concordia University
1455 de Maisonneuve West
Montréal, Québec, Canada
{s_pirzad, l_alawne, abdelw}@ece.concordia.ca

Abstract

Software maintenance is perhaps one of the most difficult activities in software engineering, especially for systems that have undergone several years of ad hoc maintenance. The problem is that, for such systems, the gap between the system implementation and its design models tend to be considerably large. Reverse engineering techniques, particularly the ones that focus on design and architecture recovery, aim to reduce this gap by recovering high-level design views from the source code. The course code becomes then the data on which these techniques operate. In this paper, we argue that the quality of a design and architecture recovery approach depends significantly on the ability to detect and eliminate the unwanted noise in the source code. We characterize this noise as being the system utility components that tend to encumber the system structure and hinder the ability to effectively recover adequate design views of the system. We support our argument by presenting various design and architecture recovery studies that have been shown to be successful because of their ability to filter out utility components. We also present existing automatic utility detection techniques along with the challenges that remain unaddressed.

Keywords: Design and architecture recovery, quality of the source code in software maintenance, utility components

1. Introduction

Software maintenance and evolution is an essential part of the software life cycle. In an ideal situation, one relies on system documentation to make any change to the system that preserves the system's reliability and other quality attributes. However, it has been shown in practice that documentation associated with many existing systems is often incomplete, inconsistent, or even inexistent [1], which makes software maintenance a tedious and human-intensive task. This is further complicated by the fact that key developers, knowledgeable of the system's design, commonly move to new projects or companies, taking with them valuable technical and domain knowledge about the system [2].

These factors contribute to making these software systems difficult to comprehend and to evolve, forcing software engineers to spend a considerable amount of time understanding the way they are implemented prior to undertaking a maintenance task. Design and architecture recovery techniques have been introduced to alleviate the impact of this problem. The objective is to recover high-level design views of the system such as its architecture or any other high-level design models from low-level system artifacts such as the source code. Software engineers can use these models to gain an overall understanding of the system that would help them accomplish effectively the maintenance task assigned to them [2, 3, 4].

Although existing design and architecture recovery techniques (e.g., [2, 3, 4, 5, 6]) vary significantly in their designs, the applied process, and evaluation strategies, they all rely on the source code as the sole reliable source of information about the system under study that remains valid in the absence of adequate documentation. As such, the effectiveness of a design and architecture recovery technique depends greatly on the quality of the source code on which the technique is applied - The more dependencies among the system components[1]; the harder it is to recover adequate design views.

However, not all component dependencies have the same level of importance. This applies particularly to utility components which tend to be called by many other components of the system, and as such they encumber the structure of the system without adding much value to its understandability. Therefore, an effective design and architecture technique must consider utility detection and removal techniques as a core mechanism in their overall design.

[1] We use the term component to refer to any system module such as methods, classes, packages, or any other module.

Our definition of a utility component is similar to the one presented in [7], where the authors conducted a brainstorming session where they asked several software engineers from industry to discuss the concept of utilities, particularly what they consider as a utility component. Their main finding is that utilities are the components of a system that are called by many other components. They are used to help implement the core functionality of the system, while they are at a lower level of abstraction than that the design elements they help to implement.

In this paper, we show how utility components are the unwanted noise that needs to be removed for a design and architecture recovery process to be effective. We support our argument by showing examples of design and recovery techniques that have been successful because of the fact that utilities have been discarded from the recovery process (Section 2). We also discuss existing automatic utility detection techniques along with the issues that need to be addressed, and the need to have more advanced techniques (Section 3). We conclude the paper in Section 4.

2. Utilities: Unwanted Noise

In this section, we demonstrate the fact that utilities are noise in the data that need to be filtered out for a design recovery approach to be effective, We achieve this by presenting various design and architecture recovery techniques that have been shown to be successful because they involve steps in which utility components are removed and excluded from the recovery process.

In [8], Mancoridis et al. proposed a tool, called Bunch, to group a system's modules (e.g., files, classes, routines, or any other component of the system) into clusters. Their approach is based on cluster analysis techniques [16]. The clustering process was performed by partitioning the component dependency graph into disjoint clusters using a clustering algorithm. The authors used high cohesion (dependency between the modules of the same partition) and low coupling (dependency between the modules of different partitions) as the main partitioning selection criterion, which they expressed in the form of an objective function referred to as the Modularization Quality (MQ) function. An important contribution of the authors' approach is the ability to preprocess the entities of the system prior to performing the clustering process by allowing the users of their tool to filter out utility modules from the clustering process. Their rationale behind this is that utility components tend to encumber the component dependency graph and may affect the effectiveness of the clustering process.

The same observation was made by Müller et al. who also argued that utility components, referred to as omnipresent components in their study, encumber the internal structure of a system without adding much value to its understandability [9]. They proposed removing them to help understand how parts of the code map to other system artifacts such as the system architecture.

In [2], Tzerpos and Holt presented an algorithm called ACDC (Algorithm for Comprehension-Driven Clustering) where the authors introduced the concept of incremental clustering. Their clustering process consists of two phases. During the first phase, they built a skeleton decomposition of the system, which contains entities of the system that are identified by users as core entities of the system (i.e., the ones that implement important concepts). In the second phase, they cluster non-core entities by adding them to the already formed clusters. An interesting aspect of their work is that they did not use the source code to build the skeleton decomposition. Instead, they built an algorithm that simulates the way software engineers group entities into subsystems. One of their main contributions is that they observed that software engineers tend to group components with large fan-in into one cluster that they call the support library cluster, which represents the set of utilities of the system.

In [5], Wen and Tzerpos proposed a metric, called MoJoFM, to measure the similarity between two given decompositions of the same system. MoJoFM can be used for example to validate the effectiveness of an architecture recovery technique by comparing the recovered decomposition with the one provided by the software designers. The authors experimented with MoJoFM on many systems using various clustering algorithms. One of their findings is that removing utility components before the clustering process can significantly improve the process of recovering high-level views of a system from source code.

Chirag et al. proposed an architecture recovery technique that relies on a component dependency graph, extracted by static analysis of the source code, to measure the degree to which a system's components can be clustered together [10]. After applying their approach to several software systems, they observed that best results were obtained when they were able to identify utility components and exclude them from the clustering process.

Hamou-Lhadj et al. showed how the recovery of behavioral design models (such as UML sequence diagrams) from execution traces can be achieved if one can distinguish between utility components and the ones that are close to domain concepts [11]. They proposed a trace filtering based on successive filtering of utilities. They conducted an experiment with a real-world system where they asked the designers of the system to validate the extracted high-level design views. Most participants agreed that the end result was very similar to the design models they initially created for the system.

Rohatgi et al. proposed an approach for solving the feature location problem, which consists of locating parts of the source code that implement a specific software feature [12]. The objective is to allow software maintainers to focus only on these parts of the source code, most relevant to the feature or source code to be modified. In their work, they proposed a ranking approach that ranks the system components (they used system classes in their study) according to their relevance to the feature under study. One of their main findings is that components involved in the implementation of many software features are mere utility components. They, therefore, assigned a lower weight to these components. The results of applying their approach to two software systems are very promising.

3. Automatic Detection of Utilities

Despite the fact that utilities, if detected and removed effectively, can improve significantly the result of a design and architecture recovery technique, there has not been a lot of work in developing algorithms for automatic detection of such components. These algorithms are needed for systems that have undergone several ad-hoc maintenance tasks and in which utilities and non-utilities are intermingled. There is usually no programming construct that distinguish a utility from a non-utility. In this section, we present an overview of the modest number of studies that aim to automatic detection of utilities along with the challenges and key research issues that remain unaddressed.

3.1. Utility Detection Using Fan-in Analysis

Fan-in analysis techniques are based on the exploration of the component dependency graph built from static analysis of the system. There are several types of static dependencies that may exist between two given components including method calls, generalization, realization, etc. Additionally, the edges might be weighted to represent the number of dependencies that exist between two given components.

Fan-in analysis has been used in a variety of studies for detecting utilities (e.g., [9, 10, 11, 13]) to measure the extent to which a component can be deemed a utility. The rationale behind this is as follows: the more calls a component has from different places (i.e., the more incoming edges in the static component graph), then the more purposes it likely has, and hence the more likely it is to be a utility according to the discussion presented in the previous section. Conversely, if a component has many outgoing edges in the component dependency graph, this is evidence that it is less likely to be considered a utility.

Müller et al. computed fan-in of the system components and proposed a utility threshold above which a component is deemed to be a utility [9]. Hamou-Lhadj and Lethbridge used a combination of fan-in and fan-out to create a "utilityhood" metric that measures the extent to which a system's component can be considered a utility [11]. They experimented with the metric and a large system and showed promising results. However, their metrics was only able to detect system-scope utilities. One of their main findings is that utilities can also appear at narrower scopes than the entire system such as in local subsystems. They concluded that detecting local utilities (i.e., package-level utilities) may need further adjustments to the utilityhood metric they proposed.

A natural extension to fan-in analysis is to consider the impact of a component modification of the system rather that its mere fan-in. Impact analysis is the process of identifying parts of a program that are potentially affected by a program change. It has been shown to be useful for planning changes, making changes, and tracing through the effects of changes [14].

Rohatgi et al. used impact analysis to detect the components that implement a specific set of components (more precisely system classes) that implement a particular software feature [12]. In their approach, they consider not only the direct impact associated with a component change but also the ones that are indirectly affected by this component change. This allows measuring the fact that the impact of a component can be very high without necessarily having a high fan-in. In Figure 1, we show an example of a dependency graph where the component *C2* has a very low fan-in (one incoming edge) but a high afferent impact value (five components are affected by a change to *C2*).

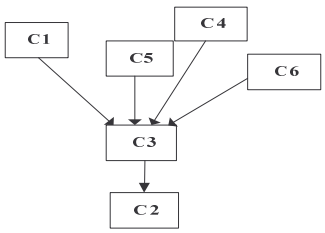

Figure 1. An example of a component with low fan-in and high impact

For the measurement of component impact on the remaining parts of a system, they used a static component dependency graph. They defined the impact set of modifying a component C as the set of components that depend directly or indirectly on C. They developed a metric that allows detecting the most important components of the system by measuring the afferent and efferent impact of components. The afferent impact of a component C consists of the number of components that are affected (directly or indirectly) when C is modified. The component efferent impact of C is the number of component that will affect (directly or indirectly) C if they change. These are the classes in the directed graph that can be reached through C.

Their approach was used successfully to detect the components that are most relevant to the implementation of specific features. As a complementary result, they were able to identify utility components as being the ones that were involved in the implementation of almost all software features of a particular system.

One of the limitations of the aforementioned techniques is that they focus only on system-scope utilities. However, we believe that utilities can have different scopes. For example, there might be utilities that belong to a particular subsystem, used to help implement the core functionality of only the elements of this subsystem. However, it is hard to determine the system boundaries for many systems that suffer from poor architecture. The main challenge is then to assess, for any candidate utility, whether it is being called from a variety of different places within the system, or else from some very specific parts of the system.

Another challenge is with respect to determining a threshold above which a component can be considered a utility. The problem is that what is considered a utility for one software engineer might be something important for another person. Therefore, any tool that supports utility detection techniques must allow enough flexibility to software engineers to dynamically vary the thresholds.

In addition, it is important to note that not all utilities are grouped into some sort of utility containers. For example, being able to detect all utility classes of a particular system may not be sufficient since there might be many utility methods that belong to non-utility classes. For example, accessing methods in most classes can be considered as utilities.

3.2. Naming Conventions

Another technique used to identify utilities consists of using design conventions including naming conventions, comments, etc. For example, Chirag et al. [10] found that many system-scope utility packages are given names that are variation of the term “Utils”.

However, naming conventions tend to be informal and rely on existing design conventions, which make them impractical if design conventions are not followed by software engineers. In addition, extracting knowledge from informal sources of information is known to be a difficult task due to the existence of noise in the data. One possible solution for this is to assess the extent to which naming conventions have been used consistently throughout the life cycle of the system. One naming convention evaluation framework was proposed by Anquetil and Lethbridge [14]. Their framework was used to evaluate the names of structured types (e.g., C structure). The idea is that if two structured types have similar names then they should represent similar concepts. They conducted an experiment on a large telecommunication system and the results are promising. In our case, the challenge is to apply this framework to evaluate the names of different components.

3.3. Utility Categorization

In [7], Hamou-Lhadj and Lethbridge propose that utility components be categorized depending on the purpose they serve. Knowing that a component belongs to one of these categories will ease the utility detection process. Examples of categories they proposed include:

- Utilities derived from the usage of a particular programming language. For example, in Java, the method toString() has the same meaning in all classes that override it, which consists of returning information about objects. This function is clearly a utility. Knowing that a system is Java-based suggests that all system components that override or implement Java elements are candidate utilities.
- Utilities derived from the usage of a particular programming paradigm. An example of this would be accessing methods used in object-oriented systems, and are used only to enforce information hiding. They are therefore not needed in higher level views of a system.
- Utilities that implement data structures. Data structures and the methods that operate on them are typical implementation details, which can also be ignored in a recovery process.
- Similar to data structure, input/output operations can also be considered as utilities since they only deal with low level data storage and retrieval.

Although classifying utilities into categories can facilitate their detection, some categories may still be hard to detect unless some sort of design convention techniques are followed. For example, in some of the systems that we studied, we found that accessing methods do not always start with the conventional get and set suffices. In addition, the above categories need to be validated. New categories may be added by studying utilities that appear in systems of a particular domain.

4. Conclusion

In this paper, we discussed the concept of utility components, which we described as any element of a program designed for the convenience of the designer and implementer and intended to be accessed from multiple places within a certain scope of the program. We also discussed how utility components, if detected effectively, can help develop effective design and architecture recovery techniques, which are reverse engineering techniques that aim to reduce the gap between implementation and design.

In addition, we surveyed exiting techniques for detecting automatically utility components.

We intend in the future to continue the work presented in this paper by further investigating ways of automatically detecting utility components. We also intend to investigate how the detection of utilities can be best integrated with tools that support design and architecture recovery techniques.

Acknowledgements: This work has been partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

5. References

[1] T. C. Lethbridge, J. Singer, and A. Forward "How software engineers use documentation: the state of the practice", *IEEE Software special issue: The State of the Practice of Software Engineering*, pp 35-39, 2003.

[2] V. Tzerpos and R. C. Holt, "ACDC: An algorithm for comprehension-driven clustering", *In Proc. of the 7th Working Conference on Reverse Engineering*, pp. 258–267, 2000.

[3] T. A. Wiggerts, "Using Clustering Algorithms in Legacy Systems Remodularization", *In Proc. of the 6th Working Conference on Reverse Engineering*, pp. 33-43, 1997.

[4] N. Anquetil, C. Fourrier, T. C. Lethbridge, "Experiments with Clustering as a Software Remodularization Method", *In Proc. of the 6th Working Conference and Reverse Engineering*, pp. 235-255, 1999.

[5] Z. Wen, V. Tzerpos, "Software Clustering based on Omnipresent Object Detection", *In Proc. of the 13th International Workshop on Program Comprehension*, pp. 269-278, 2005.

[6] V. Tzerpos, R. C. Holt, "ACDC: An algorithm for comprehension-driven clustering", *In Proc. of the 7th Working Conference on Reverse Engineering*, pp. 258 – 267, 2000.

[7] A. Hamou-Lhadj, and T. Lethbridge, "Reasoning about the Concept of Utilities", *In Proc. of the ECOOP International Workshop on Practical Problems of Programming in the Large, LNCS, Vol 3344*, Springer-Verlag, pp. 10-22, 2004.

[8] S. Mancoridis, B. Mitchell, Y. Chen, and E. Gansner, "Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures," *In Proc. of the International Conference on Software Maintenance*, pp. 50-62, 1999.

[9] H. A. Müller and J. S. Uhl, "Composing Subsystem Structures using (k, 2)-Partite Graphs", *In Proc. of the International Conference on Software Maintenance*, pp. 12-19, 1990.

[10] Chirag Patel, Abdelwahab Hamou-Lhadj, Juergen Rilling, "Software Clustering Using Dynamic Analysis and Static Dependencies", *In Proc. of the 13th European Conference on Software Maintenance and Reengineering (CSMR'09), Architecture-Centric Maintenance of Large-Scale Software Systems*, 2009.

[11] A. Hamou-Lhadj, and T. Lethbridge, "Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System", *In Proc. of the IEEE International Conference on Program Comprehension,* pp. 181-190, 2006.

[12] A. Rohatgi, A. Hamou-Lhadj, J. Rilling, "An Approach for Mapping Features to Code Based on Static and Dynamic Analysis", *In Proc. of the 16th IEEE International Conference on Program Comprehension (ICPC)*, 2008.

[13] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge, "Recovering Behavioral Design Models from Execution Traces", *In Proc. of the IEEE European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 112-121, 2005.

[14] J. Law , G. Rothermel, "Whole program Path-Based dynamic impact analysis", *In Proc. of the 25th Int. Conf. on Software Engineering,* pp. 308-318, 2003.

[15] N. Anquetil, T. C. Lethbridge, "Assessing the Relevance of Identifier Names in a Legacy Software System", *CASCON*, pp. 213-222, 1998.

[16] T. A. Wiggerts, "Using Clustering Algorithms in Legacy Systems Remodularization", *In Proc. of the 4th Working Conference on Reverse Engineering,* pp. 33-43, 1997.