Reasoning about the Concept of Utilities[*]

Abdelwahab Hamou-Lhadj
University of Ottawa
800 King Edward Avenue
Ottawa, Ontario, Canada
ahamou@site.uottawa.ca

Timothy C. Lethbridge
University of Ottawa
800 King Edward Avenue
Ottawa, Ontario, Canada
tcl@site.uottawa.ca

Abstract

Understanding large software systems is a hard task if effective tool support is not provided. We hypothesize that efficient detection of utility components can help reduce the space of components that need to be analysed. This can be applied to several software engineering areas such as software visualisation, architecture recovery and dynamic analysis. However, to achieve this goal, the concept of utility components needs to be better understood. In our early work, we organized a brainstorming session with twenty software engineers of the company that supports our research to discuss several research questions including the concept of utilities. This paper discusses the ideas that came out of this session with respect to various research questions surrounding what constitutes a utility, and how to detect them. In particular, we consider heuristics besides simple fan-in and fan-out.

1. Introduction

Without efficient tool support, the analysis of a large system becomes extremely very difficult. This is because there is a tendency to lose track of the system's artifacts and the relationships among them. This situation may be worse in the case of dynamic analysis; in our experience, the analysis of run-time information from even a small object-oriented system can be extremely complex.

Our research focuses on investigating techniques for filtering large execution traces to simplify their analysis. For this purpose, we implemented several pattern matching algorithms to compress the traces to the level where humans can understand important aspects of their structure [3]. However, these algorithms tend to be *conservative*. That is, they consider that all the trace components have the same level of importance, which is not always true in practice. Zayour, for example, experimented with routine-call execution traces of a large telecommunication system and showed that some routines can be considered as low-level implementation details and can be removed from the traces without affecting their content from the program comprehension perspective [10].

Developing efficient techniques for detecting utility components can also be useful in other areas in software engineering. The common practice is to use heuristics that are based on computing a component's fan-in and fan-out: The rationale behind this is that something that is called from a lot of places is likely to be a utility, whereas something that itself makes many calls to other components is likely to be too complex and too highly coupled to be considered a utility. However, utilities can be of different types and have different scopes. Some utilities may be global to the whole system whereas others can be specific to a particular subsystem, which make these heuristics hard to generalize.

In this paper, we discuss the concept of utility components, and investigate answers to the following questions:

1. What do software engineers consider to be a utility?
2. What types of utilities exist in a large software system?
3. What other techniques can be used to detect them besides fan-in and fan-out?
4. What are the research questions that need to be addressed for efficient detection of utilities?

The aim of the next section is to investigate the first three questions. Section 3 presents the research questions that need to be addressed for detecting utilities.

Much of the data we have used to develop and validate the ideas in this paper comes from our interaction with QNX Software Systems. We have studied traces of their software systems, and, perhaps more importantly, we have discussed the concept of utilities with software engineers at that company. Many of the points in the next section

[*] This research is supported by NSERC, NCIT and QNX Software Systems

were raised at a two-hour brainstorming session which involved 20 QNX software engineers.

2. What Constitutes a Utility?

In the UML 1.5 specification, a *utility class* is defined as “a grouping of global variables and procedures in the form of a class declaration. This is not a fundamental construct, but a programming convenience.” [7]

This definition is too specific to be able to derive a complete understanding of what a utility is, since there are other kinds of utilities, such as utility methods or packages. However, it points towards three interesting aspects of utilities that deserve investigation: a utility's *scope*, *role* and *packaging*.

Scope of a utility: The UML definition cited above, suggests that utilities most often have a *global scope*. Along the same lines, Muller et al. [4] refer to components used by several other components of the system as *omnipresent nodes*. They explain that these components obscure the relationships between system artifacts and need to be filtered out. They compute fan-in and allow the user of their software engineering tools to filter out the components that exceed a certain ‘omnipresent threshold’.

In the QNX brainstorming session, the participants also said that a key aspect of utilities is that their usage of them tends to be distributed around the system.

However, we believe it is essential to realize that utilities do not always have a global scope. In our recent analysis of a medium size object-oriented system called WEKA [9], we found that while the system implements some utilities that are global to the whole system, others are specific to particular subsystems. For example, the class Utils is a global utility class and the class M5Utils is defined in the m5 package and is only referred to by a small set of classes. Obviously using the same omnipresent threshold for these two kinds of utilities would be ineffective.

To address this issue, we introduce the concept of a *utility context*. A utility context is the context within which the utility is defined and used. We believe that effective detection of utilities therefore requires determination and consideration of a system’s context boundaries. Any technique that is used to detect utilities (e.g. fan-in) should take into account these boundaries. In a well-structured software system, the packages, namespaces and other modularization constructs will form the natural utility contexts.

A component that has fan-in from throughout the system will be more likely to be a utility than a component that has fan-in only from a single package. But if the component is called from many different places in that package, then it is more likely to be a utility than something called from just a few specific places in that package. So to detect utilities we need to consider package boundaries and also the distribution of callers.

The utility context in a language like Java may well be a package. In a language like C, where there are no explicit packages, packages can be inferred from the patterns of inclusions. However a scope as narrow as a single class can be a utility context in some cases: There may be private utility methods that are called by many other methods in the class.

Although we have demonstrated above that utilities can have scopes narrower than the entire system, our brainstorming participants focused almost exclusively on system-scope utilities. This might be because the word utility is most often applied in the context of programming languages built-in utilities and libraries as well as in the context of a system’s architectural view.

It is important to notice that while some software engineering tasks, such as recovering system architecture, may only require detecting system-scope utilities, others, such as understanding a large execution trace, require consideration of all utilities.

Packaging of utilities: The QNX software engineers in our brainstorming session confirmed the intuitive idea that utilities tend to be grouped or packaged together in a class, a namespace, a library or some other construct.

This concept is raised in a number of contexts: A UML utility class for example, is a module that groups together utilities that would otherwise have no “home”. Tzerpos and Holt [5] observed that software engineers tend to group components with large fan-in into one cluster that they call the support library cluster.

It is important to differentiate these utility packages, from the individual utilities they contain. It is also important to note that not all utilities exist in groupings that contain only other utilities; for example, accessing methods in most classes can be considered utilities. In our previous experiments, we showed that the removal of accessing methods can result in a significant reduction of the trace size [2].

In our brainstorming session, another issue that was raised is that the utilities of a system are often designed and or maintained by specific groups of people. This knowledge can help to detect utilities in cases where there is little or no explicit packaging.

Role of a utility: In the previous two points, we have discussed the characteristic scope in which utilities are defined and accessed, as well as how they are packaged. However, scoping and packaging are applied to many things, not just utilities. We must dig deeper in order to consider the *role* of utilities as distinct from other entities.

Perhaps the clearest suggestion of a utility's role can be seen in the term "implementation convenience" used in the UML definition discussed earlier.

A utility seems to represent a detail that is needed to *implement* a general design element; it is at a lower level of abstraction than that design element, and the design element recurs in several places. Indeed, in many cases, utilities can be seen as logical extensions of programming languages, which all provide built-in facilities for manipulating data and interacting with operating systems. Indeed one can extend some programming languages to explicitly incorporate user-defined utilities; this is common practice in Smalltalk, for example: In that language you can readily add new control constructs to the system as a whole, new basic algorithms to many system classes, and even new syntactic elements to the language.

From our brainstorming session, it became clear that utilities are things that a programmer shouldn't have to worry about when trying to see the 'big picture'. A programmer should understand the details of a programming language before attempting to program in it. Similarly a skilled programmer will naturally understand 'at a glance' what a user-defined utility is doing without having to look at its details. The degree to which something is a utility can therefore be measured according to the extent to which it would be understood by a software engineer in terms of what it does, or how it works.

The above can help us identify different categories of utilities that we present below. Future work should validate these categories:

1) Utilities derived from the usage of a particular programming language:

These are utilities that are created due to the nature of the programming language that is used. For example, the method toString() is typical to Java classes and all what it does is to return some information about objects. Another example is a class that implements the Enumeration interface in Java. If asked to understand the methods of this class, the analyst can easily predict the role of the methods of the interface Enumeration.

2) Utilities derived from the usage of a particular programming paradigm:

Some utilities might be created due to the programming paradigm that is used. We saw the example of accessing methods in object-oriented systems. That is, to preserve information hiding, some utilities may need to be created. Another example is the use of initializing functions such as constructors or methods called by constructors to initialize large objects.

3) Utilities that implement data structures:

Data structures are typically implementation details. Although they might differ in the way they are implemented, they almost always operate the same way including inserting and removing members, sorting elements etc.

4) *Mathematical Functions:*

Depending on the domain that is represented, in some situations, mathematical functions can also be considered as utilities. This includes logical functions such as comparisons etc.

5) Input/Output Operations:

Numerous data storage, retrieval and transfer techniques are now known by several software engineers. The idea is that if they are implemented in a system then they might be considered as utilities.

These categories are just an example of different categories of utilities. There is definitely a need to future research in this area.

3. Detecting Utilities: Research Issues

We divide the research questions that need to be addressed for efficient detection of utilities into several categories that we list here and explain in more detail in what follows.

1. Efficient detection of the system context boundaries
2. Classification of utilities into categories
3. Detecting the extent to which a utility is called from 'different' places.
4. Using naming conventions for detecting utilities

The first research question addresses the scope of a utility. As we previously stated, utilities can have different scopes including the system level scope. To determine the utility scope attribute, there is a need to determine the system context boundaries. The system architecture can certainly be used if it is deemed valid. Otherwise, there is a need to recover it. For this purpose, there are several clustering techniques that can be used [6, 8]. However, software clustering is still an ongoing research topic with its own set of research challenges [6]. In fact, one can think of the utility detection problem as a subset of the system decomposition problem.

The second research question is related to classifying utilities into categories. We hypothesise that this is a step towards detecting them. In the previous sections, we discussed some categories that can be used. There is definitely a need for more research in this area.

The third research question that must be addressed is assessing, for any candidate utility, whether it is being called from a variety of different places (within its context), or else from some very specific places. In the latter case, the candidate is less likely to be a utility. The difficulty is determining whether the 'very specific places' are in fact just a narrower utility context.

Finally, design conventions including naming conventions, comments etc can also be used to detect utilities. For example, in many systems that we are studying, we found that they contain namespaces named using the word "Utils". However, design conventions are informal and there is a need to have a framework to validate them before using them. Anquetil and Lethbridge suggest a framework for evaluating naming conventions [1]. 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 (e.g. method, class ….).

Conclusions

This paper discusses the concept of utilities. The objective is being to understand the different dimensions of this concept so we can ultimately automate the process of detecting utilities.

We conclude that utilities can have different scopes unlike the old thought that they can be only at the system level. Also, utilities are often packaged together, but not necessarily.

Most fundamentally, however, we conclude that utilities implement general design concepts at a lower level of abstraction than those design concepts.

We also discussed how utilities are usually implementation details that can be divided into several categories. We presented some categories that need to be validated in future research.

Future work needs to focus on the research challenges that are discussed in Section 3. To start with, we intend to conduct a survey that will allow us to understand the concept of utilities better.

References

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

[2] A. Hamou-Lhadj, and T. C. Lethbridge, "Techniques for Reducing the Complexity of Object-Oriented Execution Traces", *In Proc. of the 2nd Annual "DESIGNFEST" on Visualizing Software for Understanding and Analysis,* Amsterdam, The Netherlands, 2003, pp. 35-40

[3] A. Hamou-Lhadj, and T. Lethbridge, "An Efficient Algorithm for Detecting Patterns in Traces of Procedure Calls", *In Proc. of the 1st ICSE International Workshop on Dynamic Analysis (WODA)*, Portland, Oregon, USA, May 2003

[4] H. Muller, M. A. Orgun, S. R. Tilley, and J. S. Uhl, "A Reverse Engineering Approach to Subsystem Structure Identification", *Journal of Software Maintenance: Research and Practice*, 5:181-204, December 1993.

[5] V. Tzerpos and R. C. Holt, "ACDC : An Algorithm for Comprehension-Driven Clustering", *In Proc. of the Working Conference on Reverse Engineering,* Brisbane, Australia, November 2000, pp. 258-267

[6] V. Tzerpos and R. C. Holt, "Software Botryology, Automatic Clustering of Software Systems", *In Proc. of the International Workshop on Large-Scale Software Composition*, Vienna, August 1998, pp. 811-818

[7] UML 1.5 Superstructure Specification. http://www.omg.org

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

[9] I. H. Witten, E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*, Morgan Kaufmann, 1999

[10] I. Zayour, "Reverse Engineering: A Cognitive Approach, a Case Study and a Tool", *Ph.D. dissertation, University of Ottawa*, 2002, www.site.uottawa.ca/~tcl/gradtheses/izayour