Finding Reusable Software Components in Large Systems

James M. Neighbors
Bayfront Technologies, Inc.

Abstract

The extraction of reusable software components from existing systems is an attractive idea. The goal of the work in this paper is not to extract a component automatically, but to identify its tightly coupled region (subsystem) for extraction by hand or knowledge-based system. Much of our experience is anecdotal. Our experience with scientific systems differs from much of the work in reverse engineering that focuses on COBOL systems.

Module and data interconnection was collected from three large scientific systems over a 12 year period from 1980 to 1992. The interconnection data was analyzed in an attempt to identify subsystems that correspond to domain-specific components. The difficulties of dealing with large scientific systems and their organizations are discussed. The failures and successes of various subsystem analysis methods is discussed. A simple algorithm for the identification of subsystems is presented. A pattern of object hierarchies of subsystems is briefly mentioned.

The average subsystem is surprisingly large at 17,000 source lines and 35 modules. The concept of a subsystem is informally validated by developers from subsystem interconnection diagrams. The actual reusability of these identified components is not assessed.

Keywords: *reuse, component, subsystem*

1. Introduction

Our motivation in reverse engineering systems was to discover reusable software components that could be encapsulated for Draco, a knowledge-based forward engineering system [9]. Our focus in this work is not the extraction the individual components, but on general techniques for finding the locus of a component in an existing large system. We presume that once found the individual components could be extracted either by hand or by an advanced knowledge-based system.

1.1 Related Work

Much of the existing reverse engineering literature applies to COBOL programs [8] where the data model and program flow seem more explicit than in scientific programs. We collect and graph similar interconnection data. However, some techniques that work for COBOL such as data flow analysis slices [11] failed to work as well for our purposes as discussed in section 4.4.2. Perhaps it is because the number of paths through a large scientific application is much larger than through a large COBOL application. This large number of paths makes it much more difficult to recognize the plans of a technique such as [15]. Further compounding the problem is the richness of the underlying problem domain. As succinctly stated in [13] "domain-specific operations and objects form a domain model absolutely critical for understanding the programs built on top of them." Unfortunately, as we discuss in section 4.1, even the developers of a large scientific application may not have this domain model.

Some of the problems we have encountered may be due to the large systems we examined. Our subsystem analysis work is similar to the patterns of relationships in the DESIRE system [2]. However, DESIRE was able to use "suggestive data names" as functional clues; but in very large systems global names tend to no longer be functional but architectural as discussed in section 4.4.3.

The goal of this work is not to displace knowledge-based program understanding. Instead it is to place a bounding scope on recognizing "plan calculus" [14], "design patterns" [4] and "typical architectures" [5] in the millions of source lines that make up general systems.

2. Hypotheses

The basic beliefs that lead us to use a structure-based approach rather than a knowledge-based approach during continuing system development are discussed and rationalized in the following sections.

2.1 Architecture of Large Systems

Classical Software Engineering teaches that the architecture of a system is a tradeoff between top-down functional decomposition (Stepwise Refinement) [16] and bottom-up support of layers of Application Programming Interfaces (API's or Virtual Machines) [3]. Functional decomposition partitions the architecture vertically while

virtual machine layers partition the architecture horizontally.

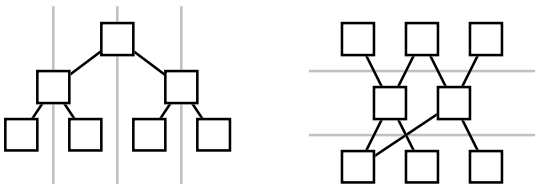

Figure 1. Functional Decomposition and API Decomposition

These two partitioning forces are in constant opposition. The decision as to which of these approaches to use is the policy of maximal information hiding [12].

Since real systems are constructed as a trade-off of these two decomposition techniques we cannot expect the architecture of a large system to predominately follow one technique. They are neither top-down decompositions nor bottom-up layer by layer constructions but a combination of the two processes. Experience with system construction leads us to believe in the informal concept of subsystems. Subsystems are individual regions of a system that are tightly coupled in data and function. They are the large system cells that result from the architecture partitioning tradeoffs. Subsystems represent encapsulations convenient to system designers, maintainers and managers. Thus, we would expect subsystems to be present even in old, heavily maintained systems.

From these reasons we decided not to stress either top-down decomposition or bottom-up layer construction in our attempt to determine system architecture. Instead we stressed determining the existence, structure, size, scope and function of subsystems resulting from tradeoffs in these partitioning approaches. We would expect these subsystems to be the reusable software components we seek. After all they are the distillate of years of maintenance.

2.2 Economics of Large Systems

The complete reverse engineering of old, large systems followed by the subsequent forward engineering of replacement systems will be very rare for economic and business reasons. If we assume a low $20/SLOC (source line of code) evaluation, then a million line system is a $20 million business asset. These assets maintain themselves by producing continuous sales. These sales support the 20-100 people needed to maintain the system. In short, the system is a cash generator and an evolutionary survivor. It takes a long time for a system to grow to a million lines. Usually, it has had to compete in its particular area of expertise with similar systems. Technical staff that suggest a reverse and forward engineering approach put the management in the position of effectively scrapping a $20 million asset and starting afresh with some scavenged parts on the extremely perilous process of large software development. This is not an attractive decision to make. If the cash flow is to be maintained then the development team must be expanded and forked into two teams. The "old" system team maintains the cash flow while the "new" team forward engineers the replacement system. Everyone on the "old" system team will know that at some time in the future their system knowledge will be obsolete and their employment will be in jeopardy. They will be unhappy and they are maintaining the cash flow. This is not a good business management situation.

Studies of software maintenance [7] have shown that maintenance and enhancement of existing function account for 75-80% of resources. In effect the system is continually rewritten. The management trick of large system maintenance is to harness all that change and channel it towards reverse/forward reengineering that creates flexibility for the system to meet new challenges. This points once again towards architectural encapsulations.

For these reasons we decided that for large systems the process of reverse engineering and forward engineering could only be viewed as the ongoing process of system reengineering (evolution and maintenance).

2.3 Problem Domains of Large Systems

Large systems address problems in a general problem area, such as banking, accounting, CAD, CAM, CAE, command/control, and inventory. Underlying all these problem domain specific applications are more general systems treated as subsystems such as operating, networking, graphics, and database systems. The large system represents a very rich set of knowledge ranging from the very domain specific to the very general. The problem of reverse engineering is that all of this knowledge is spread and intermingled in the context of programming language code throughout the system source code [6]. Because of this we would not expect knowledge based reverse engineering to be very successful. To successfully reverse engineer an arbitrary system we would need a knowledge base at least capable of forward engineering the system.

Experiments with this approach must be carefully examined. Any small experimental program fragment can be reverse engineered given a knowledge base for that fragment. It is not a question of capability - it can be done. It is a question of practicality for systems of size to be of interest. Usually researchers in this area have suggested that "people in the loop" be the ultimate backup to a failure of the knowledge base to understand a source

code fragment. In the limit this is the "people in the loop" looking at a million lines of code when the knowledge base fails and then explaining to the reverse engineering system what that code means in some semantic notation that meshes with the semantic understanding already constructed by the reverse engineering system. We doubt people can do this in a large problem domain specific system.

From the previous discussion we decided that knowledge-based understanding of large system semantics was currently too difficult for three reasons: absence of a robust semantic theory, lack of problem domain specific semantics, and knowledge spreading in the source code.

3. Experimental Method

Over a 12 year period from 1980 to 1992 we participated and collected data during the reengineering of three large systems. By this we do not mean the systems were completely rewritten. In our cases the reverse and forward process led to a 10% to 20% reduction in the number of modules and with an upwardly compatible increase in function. Table 1 presents the characteristics of the systems at the end of the process. The approximate source lines of code measurements (SLOC) does not include comment lines.

The CAD/CAM and CAE/CAM systems were both started around 1968 and so were over 10 years old at the start of the reverse engineering process. Reverse/forward engineering for these systems was motivated by the desire to move from mainframes to workstations and support new graphic hardware. The switching system was started in 1980 and was grown very quickly over 3 years. Reverse/forward engineering for this system was motivated by management, performance and flexibility requirements.

system	role	size	source
Telcom / Datacom switch	Software Architect (full time)	4M SLOC 3,800 modules	Pascal, C, assembly
CAD/CAM	Consultant (part-time)	2M SLOC 3,394 modules	FORTRAN, C
CAE/CAD	Manager (full-time)	4M SLOC 7,089 modules	FORTRAN, C

Table 1. Systems Examined

3.1 Interconnection Data

With each system we collected interconnection data between the modules of the system (globally linkable, top lexical level procedures and functions) and global data stores (unnamed COMMON, named COMMON and global storage variables). This data was only taken on the FORTRAN and Pascal that constituted the bulk of the systems in each case. In most cases the C and assembly modules were software drivers for hardware. Connections from the FORTRAN and Pascal to C entry points were taken. The data was taken on the release directly after the forward engineering phase.

We used three techniques to obtain interconnection data. First, parsers for each language were built and the source code was scanned for static interconnections. Second, link time interconnections were found by scanning linker reports and object modules. The runtime libraries for the various languages were removed from this data. Finally, execution monitoring was used to determine dynamic interconnections. The dynamic interconnections, of course, are biased by the input data to the system during execution monitoring. In all cases we used the quality control regression tests that exercise as many system functions as possible under varying loads.

We only collected module reference interconnection data. Global data stores are treated as modules that make no references. We did not collect the SLOC size of individual modules and in retrospect wish that we had.

3.2 Subsystem Analysis

With each system we developed tools proprietary to each system owner that produce interconnection diagrams and reports. These tools were easy to build and earned us the trust of the developers and managers. We used this trust to ask endless questions about subsystem scope. Researchers in this area must understand that without this trust the developers will not take them seriously since they are just another person here to study something in the way of finishing the next system release. The developers will not care who they are or who sent them. Researchers need to help developers if they want their help.

The simplest interconnection report is a cross reference tool that lets the individual developers easily discover how the system works and how the part they maintain fits into the overall architecture. The hardest reports are inter and intra source code module quality compliance reports from the source code parsers. The simplest diagrams are top-down decomposition views similar to the one show in Figure 1. The hardest diagrams are policy-driven subsystem diagrams similar to the one shown in Figure 4.

With each system we went through cycles of proposing subsystems and asking the developers responsible for each area whether it was a tightly coupled region. The focus of the discussion was always on what modules could we add and why and what modules could we remove and why. The goal was to determine a method to identify a

subsystem and thus identify at least a starting point for extracting reusable software components from existing large systems.

4. Results

We admit that we started naively. Armed with a research-level knowledge of Software Engineering and a background of 10 years experience with many systems up to 250K SLOC we decided to study large systems. Large systems seem to be where most Software Engineering problems are found. Large systems should also contain large numbers of reusable software components in the form of subsystems. While the particular components we would find would belong to the owners of the systems, we reasoned it was worthwhile to determine if the locus of modules that form a component could be determined from the explicit structure in the system. The extraction of the component would be an effort past this.

4.1 Large System Development Issues

Many issues related to large system development needed to be addressed before any data could be collected.

In general the developers and managers of large systems hold that the statements enumerated below are true of their system. We found that most of the time they are right. However for a system of 4,000 modules if they are not true only 5% of the time, then 200 modules are affected. This can hamper reverse engineering efforts that focus on interconnection. We found the following issues in *all* of the systems we examined. Each generally true statement is accompanied by one of many sensible scenarios of how it can become false in large systems.

1. **We know what the system does**. All large systems have special customers or configurations that occasionally require special versions. These custom solutions become part of the common system to ensure their maintenance. Usually only the developers that added these custom solutions know what they do.
2. **The system is one big program.** All of the systems we examined are a complex array of versions, configurations, support utility programs, and support data files. In reverse engineering all of these items need to be examined. It is difficult to find them all.
3. **We know where all of the source code is.** There are many good reasons why a source code is not under source code control. Security codes, proprietary codes and codes that no longer compile under the current tools are routinely removed from the source base. A poor reason is that certain programmers just refuse to check in their codes.
4. **We know the version space of the system.** This addresses the feature space of the system. All large systems are linked together many ways to provide different features to the users. There is not one set of compiled source modules in every system sold. A series of system releases over time usually enlarges the version space. Marketing usually breaks down its product list by the available versions. The developers seldom have a list of the modules in each version even in Make files. This space is compounded by the configuration space below.
5. **We know the configuration space of the system.** This addresses the hardware (and driver software) space of the system. This includes the computer, operating system, and all attachable I/O devices. As with the version space there is not one set of compiled source modules in every system sold. The configuration space is compounded by the version space. Make files in theory could characterize the modules in each system of version by configuration; but in practice it is seldom done because the space is so large.
6. **The system is written in a standard programming language.** Usually error trap to stack unwind, fast cross system branch, memory allocation, shared data, I/O needs and message passing are implemented knowing the runtime structure of the compilers in the configuration space.
7. **The design documentation represents what the system does.** The only internal design documentation that is maintained is that which helps the maintainers. Usually this includes code headers, global data descriptions, and end-user interface kit documentation. Other design documentation away from the code is not maintained.

It is a mistake to assume that the above problems are somehow a consequence of mismanagement. Better processes and tools would have very little impact. These problems are just part of building large systems. All large systems have these problems.

4.2 Interconnection Data

Figures 2 and 3 present the module interconnection data for the two FORTRAN-based systems of Table 1. In each case we used the largest configuration and version. This data represents 10,483 modules and approximately 6M SLOC. The average module size is about 570 SLOC. Global data stores are treated as modules that make no references. If one module references another module multiple times, it is counted as a single reference. The vertical lines in the graph represent the range in values between the two systems. The horizontal bar represents

the average value between the two systems. Some data has been eliminated in both figures by limiting the References axis. In Figure 2 we eliminated a total of 210 modules that were referenced by many modules (maximum value 1455). In Figure 3 we eliminated a total of 64 modules that referenced many modules (maximum value 144). With the full set of data the References axis of both figures would be long and sparse at the higher values.

The data is surprisingly similar considering that the two systems, while having a similar function, have radically different architectures. One system has over 980 named COMMON global data areas while the other only has 20 to 30. One system is twice the size of the other in SLOC.

These systems are very carefully partitioned. Figure 2 shows that fully 60% of the modules of both systems are referred to by only one other system module. Notice that in the strict top-down decomposition approach of Figure 1 each module is only referred to by one module. These modules exist simply as functional decompositions.

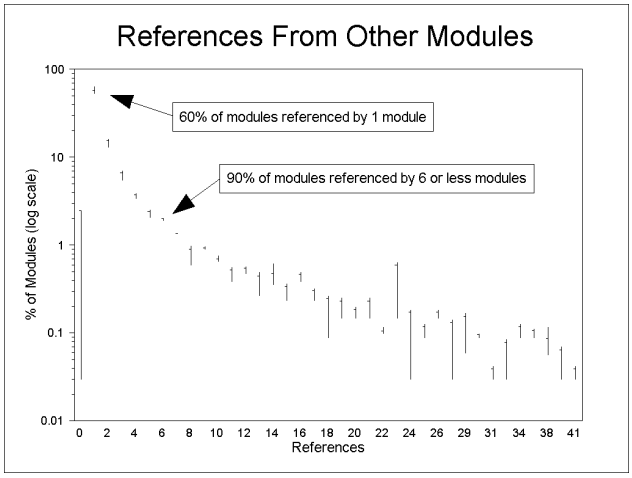

Figure 2. References From Modules

The wide difference in the number of modules referenced by no other modules in Figure 2 stems from the handling of environment callbacks by the two FORTRAN systems. One system uses a central dispatch while the other uses a separate module for each needed callback. All of the callback modules look as if they are not referenced.

Figure 3 shows that even though there are many modules in the system, an individual module refers to very few of them. In fact 50% of the modules refer to only 3 other modules. Remember the average size of a module is over 500 SLOC. This is much more highly partitioned that we would expect to see in a small 10K SLOC program.

Notice that Figure 3 could be fit by a straight line against a logarithmic scale. We do not know why. It seems to suggest a constant module embedding. Comparing the ratio of Figure 2 with Figure 3 does show that the reference of modules for support is slower to decline. Figure 2 shows that 60% of the modules are only referenced by one other module but Figure 3 shows that 50% of the modules refer to 3 or less other modules. This hints at the horizontal partitioning of Figure 1 in action.

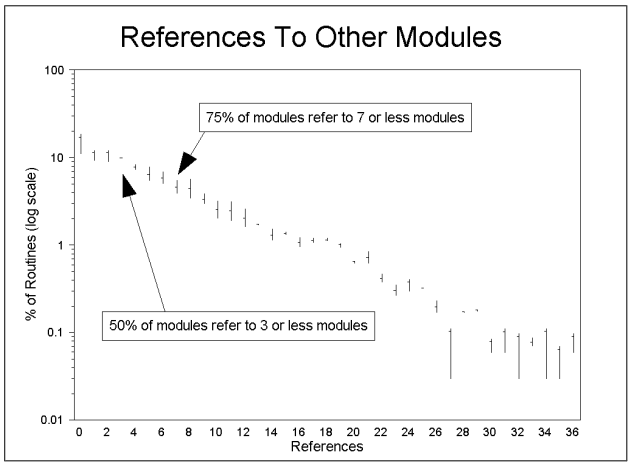

Figure 3. References To Modules

4.3 Maintenance Programming Style

Some of the effects seen in the interconnection data can be explained by years of maintenance programming. Maintenance programmers are very conservative in the way they approach changing a system. Common maintenance techniques are:

1. Surround all small local changes with IF-THEN protections so that the system exhibits the new behavior only when a specific predicate is true.
2. Try to combine larger changes into an existing module so that system-wide problems of defining a new module (e.g., naming the module and finding all references to the old module and determining which should call the new module) can be avoided. Use passed control variables to identify the functionality desired in each case.
3. For changes that require modification to a complete suite of modules (subsystem) make a copy of the complete subsystem with suitable translation naming and modify the copy of the subsystem.

In forward engineering these are not good programming practices. The first leads to complex, unnecessary and brittle predicates all over the system. The second is poor module cohesion usually with passed control coupling. The third leads to a dramatic rise in the

number of lines of code in the system. All of these techniques lead to dead code. The maintenance programmers do not use this style because they are foolish or not very good. They program this way because it works for them. Their concern in the short term is system stability and maintaining a common understanding of system function among the developers. It would be chaos to allow any developer to arbitrarily add or delete individual modules in the system at will. In the long term, without cycles of architecture revision, these techniques lead to hard to maintain and understand systems. These are the systems we will have to reverse engineer to find reusable software components.

4.4 Large System Structure

Our hypothesis is that large system architecture is a collection of subsystems, each piece of which is another embedded subsystem. The interconnection data of Figures 2 and 3 hint at this by affirming that the systems are very highly partitioned. This affirms that the tradeoffs of Figure 1 were made. These tradeoffs resulted in system architecture cells or subsystems. The questions we are trying to answer is what constitutes a subsystem and how do we find them.

Our basic approach to finding these subsystems was to propose them and ask the developers what was missing. Proposing the subsystems based on the interconnection data took many wrong turns. Some unsuccessful and successful approaches are described below in chronological order. Each approach was implemented and given to developers for evaluation.

4.4.1 Failure: subsystems based on decomposition: Given a module, the subsystem includes all of the modules it references and include the transitive closure of those modules. To our surprise this ended up including every module in the system almost every time. Even more surprising this turned out to be the case even for the systems written in FORTRAN, a non-recursive language. Yet, in execution the systems never recur or they fail. This behavior is a consequence of maintenance programming with passed control variables discussed in the previous section.

The developers liked the top levels of these reference trees. We added a depth limit to the traversal but it was only a rough fix to get the top level decomposition diagrams out.

4.4.2 Failure: subsystems based on intermodule data flow analysis: Given a module, the subsystem includes all modules in that module's interval as determined by interval analysis [1]. Informally a module's interval is a local set of modules bound by data production, consumption and pass-through dependencies. Full data flow analysis on a large system with large amounts of global data is virtually impossible since it is related to the number of paths through the source code. In our case we applied the technique to the reference structure of the systems that includes a restricted view of global data. The subsystems proposed were the individual interval graphs that were embedded into higher level data flow graphs that represented higher-level subsystems.

The developers liked the interval groupings because it pared away context of the surrounding system. Many times too many modules were included because as can be seen in Figures 2 and 3 the system is very vertically partitioned. Because of high module specialty the context of a special module is highly specialized (i.e., few modules reference it). This leads to the data of the module being highly dominated by a specialized line of reference modules. The developers expressed the opinion that this long vertical string of referencing modules crossed many subsystem boundaries. This hints at embedding but is as yet unclear.

4.4.3 Success: subsystems based on module name matching: Given a string matching pattern on module names, produce a subsystem diagram based on the reference structure of modules whose names match the pattern. The name space in large systems is very important. Naming conventions are rigorously followed. The module names are not functional descriptions but architectural markers. In large systems it is more important to determine the owning subsystem of a module rather than its function. So this approach is not as odd as it might seem at first blush.

The developers motivated this approach because it is one of the basic approaches they use. Also from earlier diagrams we had heard "This has a lot of the modules but it is missing some that are named similarly to these." The problem with this approach is that to get a particular subsystem diagram acceptable to its associated developer team, we would start to enlarge the regular expression. In effect we were explicitly stating the modules of a subsystem with a large regular expression. The expressions were not general or time invariant. The benefit of this approach was that we had given the individual developers a simple tool they could use. They started to identify and draw the subsystems for which they were responsible. They became enthusiastic and we got examples of subsystems. This strengthened our belief in the subsystem concept.

4.4.4 Success: subsystems based on reference context:. Given a module that determines the subsystem of interest and parameters M and N, add a module to the

diagram only if N percent of the modules that refer to it are already included or M percent of the modules it refers to are already included. Repeat until no module fits the policy.

This approach was very successful with the developers. In fact, with minor parameter variation, we were able to duplicate and exceed the developers efforts based on the name matching technique of section 4.4.3. The technique seems best suited to the lowest-level subsystems where the parameters are clear. To examine subsystem embedding we would need to collapse the low-level systems and treat it and all of its references as a single module. This is the same embedding idea used in the intermodule data flow approach. However, this basic approach is capable of producing a diagram of related composed subsystems. Figure 4 is an example of such a diagram.

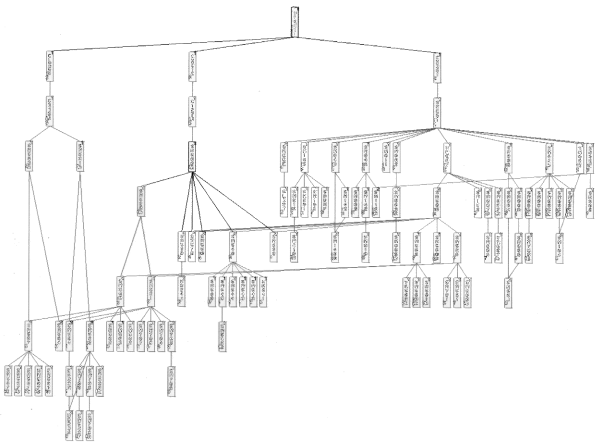

Figure 4. An Example Diagram of Composed Subsystems

Figure 4 shows a collection of three related subsystems drawn using the technique given in this section. Each box represents a module averaging over 500 SLOC in size. The diagrams are usually drawn in color where the color represents the time from the last change. The colors range from recent changes within 5 days in red through older changes past 2 years in black. Both the modules and the reference lines indicate time of last change. This way a developer can determine quickly determine possible sources of an error. Once an error has been localized to a subsystem the best places to look for the problem are the red modules and references.

The particular story of the subsystems in Figure 4 is an interesting one for reverse engineering. The oldest subsystem is the one on the left. It is rather small at 19 modules and about 9,000 SLOC. The second subsystem in the middle was grafted onto it at a later time. This second subsystem contains 27 modules and about 13,000 SLOC. The rightmost and newest subsystem is about average size for the subsystems we found at 54 modules and 27,000 SLOC. It was grafted onto the middle subsystem at a later time. An appropriate structure for the reverse engineering of these three related structures is as objects. The leftmost subsystem is the base object. The middle subsystem descends from the base object. The rightmost subsystem descends from the middle one. These are much larger objects than usually found in object-oriented programming discussions. Perhaps, this is how objects should be used in large systems.

4.4.5 Subsystem Structure: After proposing subsystems to the developers and getting them approved as the highly cohesive concept units of the system, we determined that in general subsystems have the basic structure shown in Figure 5.

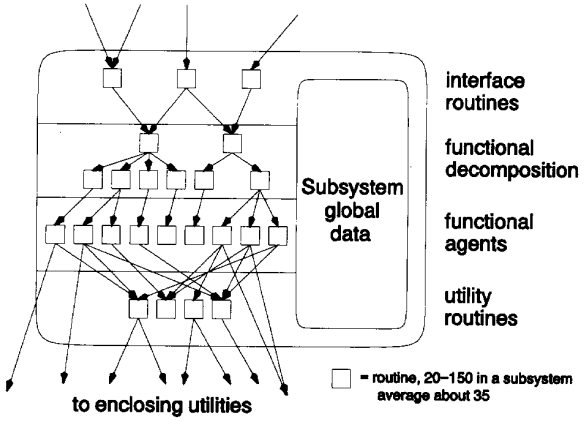

Figure 5. Basic Subsystem Structure

The basic subsystem structure is not a surprise. It is similar to the structure of a small programs and programming language level objects. Both of these concepts represent encapsulations convenient to system designers and maintainers. We found approximately 20 to 150 modules in a subsystem with the average size of about 35 modules or 17,000 source lines. The reader should be able to validate that the structure of Figure 5 is somewhat the structure of the three subsystems in Figure 4.

5. Conclusions

We make no pretense that this was a scientific investigation. Rather it was an attempt to make natural observations on actual systems. We reported techniques that worked on only three systems, but with thousands of modules. It is not known whether these same techniques will work on other systems.

Section 4.1 reports the issues, problems and solutions in collecting data from real systems. We hope that our techniques and solutions will aid other researchers in their observations. We have presented a simple technique in section 4.4.4 that seems to work to identify subsystems. The use of subsystems as manpower assignment technique to clean up the systems was very successful.

We believe that the hypothesis of subsystem existence was validated. We believe that subsystems would be a good start for the extraction of domain-specific software components. The best validation of the subsystem concept was that on their own volition developers made subsystem diagrams; put them on their office walls; and used them for reference, planning, and training. The architectural concept of subsystems is an important one for forward synthesis systems such as Draco that purport to address the building large systems [10]. We also believe that the subsystems can be a basis for converting an old system to object-oriented programming as discussed in section 4.4.4.

We were very much surprised that the curves for the two FORTRAN systems in Figure 2 and 3 were similar. The Pascal system curves show the same characteristic shapes but with at different values. We believe the curve shapes are an aspect of large system development. We believe the particular values are characteristic of the encapsulation mechanisms of the underlying programming language.

The hypothesis of subsystem embedding seems used, but it is not clearly validated. We are led to this conclusion by examples such as Figure 4 that could be viewed as an embedding. The M and N parameters in the subsystem determination policy of section 4.4.4 seem to change the view from enclosing subsystem to enclosed subsystem. Also, the developers and managers talk about subsystem embedding. Perhaps even for large systems the degree of embedding is small. We suspect that the ratio of the data in Figures 2 and 3 says something about average subsystem formation and embedding; but it is still unclear. Embedding needs a better determination and the straight line fit of Figure 3 seems to call for some explanation.

We believe that subsystems represent the domain-specific building blocks of large systems. At an estimated average size (in FORTRAN) of 17,000 source lines and 35 modules they are much larger than the loops, data structures, procedures, functions, and programming language-level objects that implement them. We believe that these subsystems are where knowledge-based program understanding systems should focus their attention. In the absence of such systems for the reasons discussed in section 2.3, these subsystems can serve as the basis for the manual extraction of reusable software components.

We would like to thank the companies, managers, and developers that took their valuable time from the ordered chaos of large system development to help us.

References

[1] Aho, A., and Ullman, J., "Data Flow Analysis", Section 11.4, in *The Theory of Parsing, Translation, and Compiling, Volume II: Compiling*, Prentice-Hall, 1973.

[2] Biggerstaff, T., Mitbander, G., and Webster, D., "Program Understanding and the Concept Assignment Problem", *Communications of ACM*, **37**, 5(1994), pp 72-82.

[3] Dijkstra, E., "Complexity Controlled by Hierarchical Ordering of Function and Variability", in *Software Engineering*, P. Naur and B Randell eds., NATO Science Committee Report, Germany (1968) pp. 181-185.

[4] Gamma, E. et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[5] Garland, D., and Shaw, M., "An Introduction to Software Architecture", *In Advances in Software Engineering and Knowledge Engineering, Vol. 1.*, World Scientific Publishing.

[6] Letovsky, S., and Soloway, E., "Delocalized Plans and Program Comprehension", IEEE Software, May, 1986, pp 41-49.

[7] Lientz, B.P., and Swanson, E.B., "Characteristics of Application Software Maintenance", *Communications of ACM*, **21**, 6(1978) pp. 466-471.

[8] Markosian, L., et al., "Using an Enabling Technology to Reengineer Legacy Systems", *Communications of ACM*, **37**, 5(1994), pp 58-70.

[9] Neighbors, J., "The Draco Approach to Constructing Software from Components", *IEEE Trans. on Software Engineering*, **SE-10**, 5 (1984) pp. 564-574.

[10] Neighbors, J. "An Assessment of Reuse Technology after Ten Years", in *3rd International Conference on Software Reuse*, IEEE Computer Society Press, Nov. 1994, pp. 6-13.

[11] Ning, J.,Engberts, A., and Kozaczynski, W., "Automated Support for Legacy Code Understanding", *Communications of ACM*, **37**, 5(1994), pp 50-57.

[12] Parnas, D., "On the Criteria to be Used in Decomposing Systems into Modules", *Communications of ACM*, **15**, 12(1971) pp. 1053-1058.

[13] Quilici, A., "A Memory-Based Approach to Recognizing Programming Plans", *Communications of ACM*, **37**, 5(1994), pp 50-57.

[14] Rich, C., and Waters, R., "Automatic Programming: Myths and Prospects", *IEEE Computer*, August, 1988, pp 40-51.

[15] Waters, R., "A Method for Analyzing Loop Programs", *IEEE Trans. Sftw. Eng.*, **SE-3**, 3(1979) pp 237-247.

[16] Wirth, N., "Program Development by Stepwise Refinement", *Communications of ACM*, **14**, 4(1971) pp. 221-227.