Design Maintenance: Unexpected Architectural Interactions (Experience Report)

Ian Carmichael, Vassilios Tzerpos and R.C. Holt
University of Toronto
Toronto, Ontario, Canada
{ihc,vtzer,holt}@cs.toronto.edu

Abstract

There have been many systems developed that attempt to recover design and structure from code. In this paper, we present our experience with using one such tool, SoFi, to extract design structure from a large industrial system written in C. We compare the extracted structure to that which was intended by the designers of the system. We observe and categorize, for our system, the reasons why these two views differ. We observe that seemingly minor decisions in implementation, can have a large impact on the extracted design, and draw some conclusions about the practicality of trying to recover "intended designs" from source code.

1 Introduction

Many tools have been developed with the goal of extracting structure from code. These include tools such as Rigi [Muller 88], Star[Mancoridis 94], CIA [Ramamoorthy 90], ARCH [Schwanke 89] and CARE [Linos 92].

We present our experience with using this kind of a tool on a large, recently-developed industrial system.

The tool we used to perform source code analysis is SoFi [Tzerpos 95] (for Source File), developed as an extension to the Star Tool. SoFi comprises facilities for source code information extraction from C programs. There are 3 basic components to our design-extraction tool.

1. Information on source inclusion relation between source files can be extracted. (For this project, each c file, had a corresponding header file describing the interface to that file, thus the source inclusion does describes dependencies. This is not necessarily the case in any 'C' program.)
2. Source files can be grouped into subsystems based on various criteria such as directory structure or naming similarities.
3. A tool is provided for viewing and navigating the resulting "software landscape" [Penny 92].

We were interested in seeing how views extracted from such tools, and the informal diagrams developers use to describe their systems differ. We wanted to see what kind of insight the visualizations provided by the tools might offer to the developers of such systems.

The system we analyzed was a commercial compiler recently developed at IBM. In size, it was about 300,000 lines of new code in about 200 source files. It was thought of as a reasonably well designed and well constructed piece of software.

2 Experience with the Tool

From our experience, we have concluded that visualizing the source file structure of nontrivial systems without employing some clustering of files is of little value. The resulting "picture" (figure 1) has become familiar.

The file naming convention of the compiler project gave us a simple, yet useful way to approach clustering. The major subsystems were identifiable by the file-name prefixes.

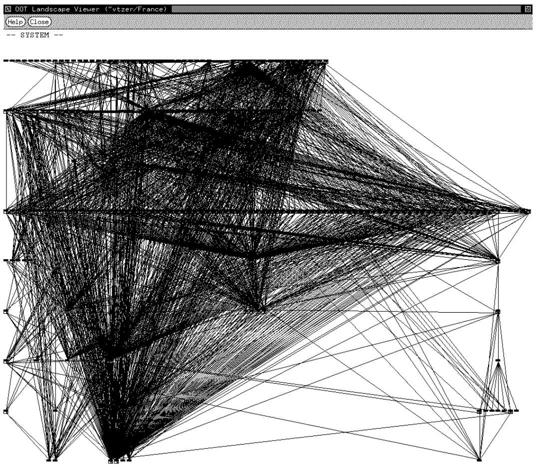

Figure 1: The unclustered structure of the system.

Utilizing this naming convention, SoFi was able to produce a much better visualization of the source file structure of the system, as shown in figure 2. This picture begins to show a meaningful structure, identifying interconnections between major components.

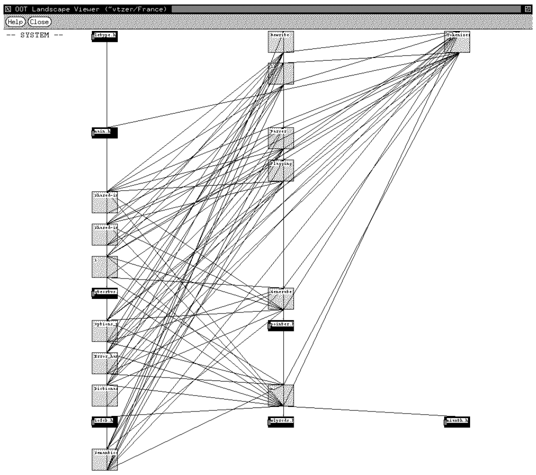

Figure 2: Layout produced by SoFi after source files were grouped into subsystems.

2.1 Interacting with the system designer

At this point, we enlisted the help of the system designer to help us lay out the diagram in a more meaningful fashion. The designer was also able to identify some low-level shared subsystems, whose connections with the overall system were elided.

In figure 3, we see the view of the system, after consultation with the system designer. The layout was rearranged to more closely match the design of the system.

This view, however, was still quite different from the original design of the system. We were most interested in discovering the underlying reasons for these discrepancies.

2.2 Analysis

We looked, in detail, at the unexpected interconnections between those subsystems which formed the core of the compiler (the Tokenizer, Parser, Semantic Analysis, Rewrite, and Code Generator). The dependency graph, in the original design, looked like that shown in figure 4. (This picture appears in early design documentation for the system.)

In sharp contrast, the graph, as extracted from the source code by our tool, was two edges short of a complete graph (figure 5). The designer did not expect these views of the system to be so different.

Among these 5 subsystems, there were 14 subsystem to subsystem include dependencies which the designer did not expect to see. These 14 unexpected dependencies were the result of over 450 file to file include depencencies which crossed the boundaries of these subsystems.

We had the system designer sit down and look at each one of these, and decide exactly why the include dependency was present.

We found, to our surprise, that all of these unexpected dependencies were the symptoms of a small handful of underlying decisions in implementation. This was not what we expected, given the rather large number of dependencies. We have categorized the implementation decisions we found as follows:

- ARCHITECTURAL EVOLUTION. One of the edges in this graph (Code Generation back to Rewrite) was decided by the designer to belong as part of the system, even though not part of the original design. It was considered a natural evolution of the original design.

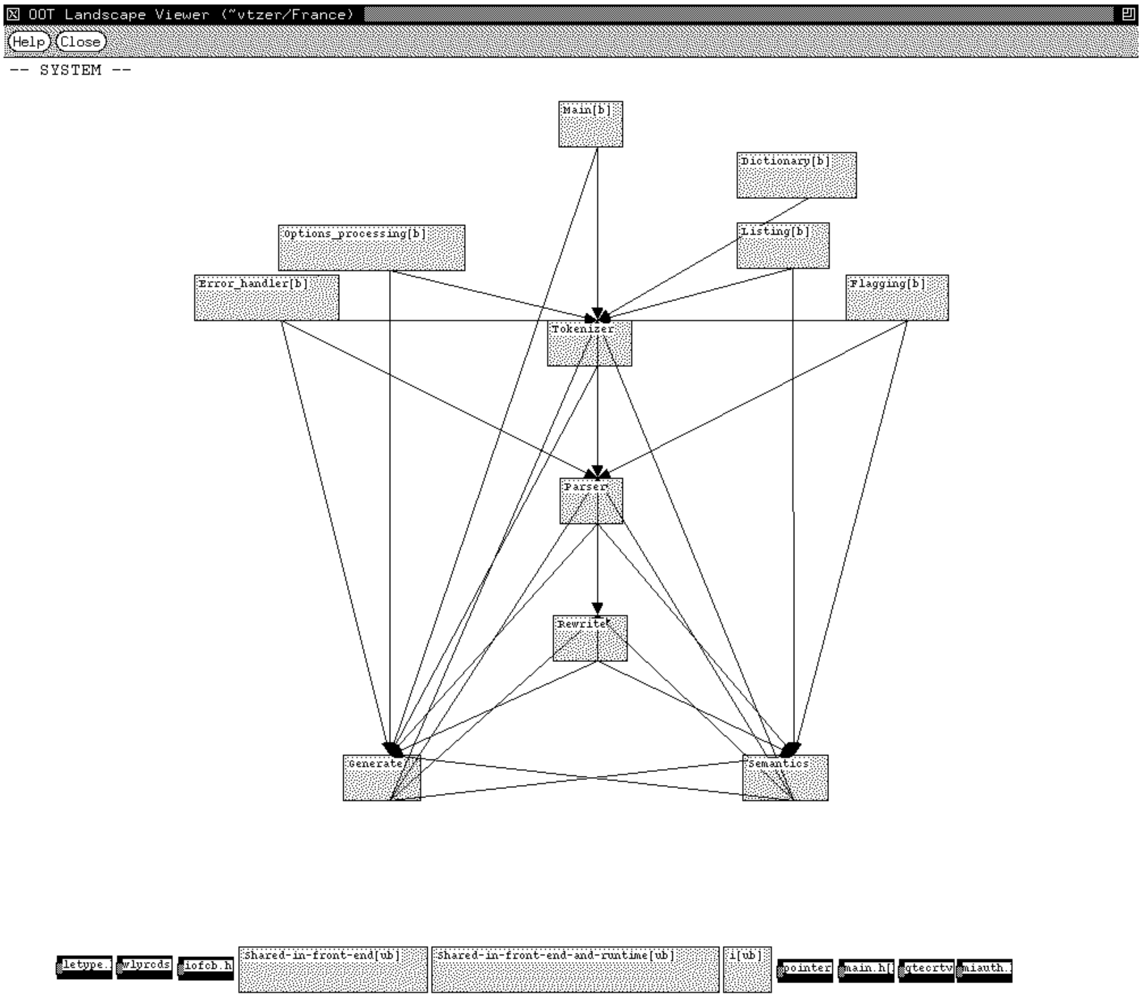

Figure 3: Redrawn by system designer.

- CONTAINER INVENTION. When a developer adds a new item to the system, a decision has to be made on where to put the item. (For example, variables and functions need to be placed in appropriate files, files need to be placed in appropriate subsystems.) When an appropriate container was present, programmers generally did this rather well. However, it seems that when a container did not already exist, rather than create a new one, programmers would sometimes add the item to a container which was not entirely appropriate. This item existing where it doesn't really belong creates unusual dependencies.

Consider the following two examples.

1. During development, a number of variables were added to capture global information about the state of a compile (had there been a severe error, etc.). Ideally, these variables would have been collected, and placed together in their own file, part of the shared utilities. However, no such part existed, and these variables were scattered through a number of different files.
2. The early architectural diagrams did not distinguish a separate subsystem for the abstract data-type for parse

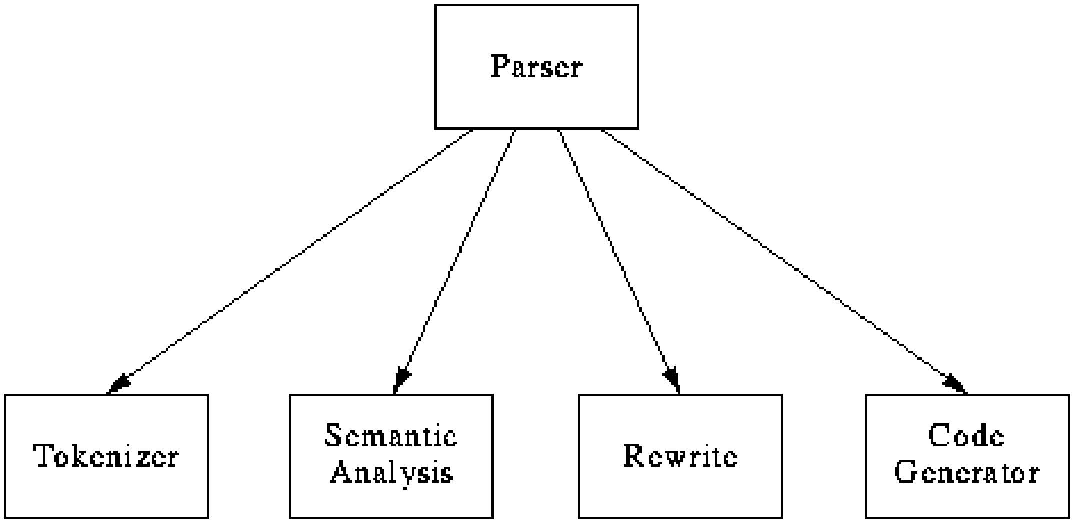

Figure 4: The intended structure of the five main subsystems of the compiler.

trees. As a result, both the parser and the parse trees ended up as part of the same subsystem. While this is plausible on the surface, the parse tree data-type is shared throughout the compiler, while the parser is not. This results in unnecessary dependencies on the parser.

- FUNCTION MIGRATION. As the code grows, functions that are originally written for one purpose become needed elsewhere later. Programmers do not always move such functions to appropriate locations. They are occasionally left where they first appear, creating unexpected dependencies.
- IMPLEMENTATION EXPEDIENCY. There were two cases, where a developer had deliberately ignored the "architecture". In both cases, the overriding concern was minimizing development time.
- GRATUITOUS INCLUDES. These occur when a file includes a header file on which there are no dependencies. Typically, these would appear because when creating a new file, an older file is copied and used as a boiler-plate. Includes which were necessary in the original file, but are not necessary in the new file are not always removed by the developers.

It was the designer's opinion that, with the exception of the one edge described above as an architectural evolution, these differences were not necessary, could be fixed with little effort, and were well worth correcting.

3 Conclusions

We have observed some of the ways in which the concrete structure of a software system evolves (and de-evolves) during development; and have categorized some of the ways in which unexpected interactions between architectural components can arise.

A small number of seemingly minor implementation decisions had a profound impact on inter-component dependencies, and hence the extracted structure of the system. Conversely, correcting these implementation decisions can profoundly reduce the number of inter-component dependencies.

The intended design did not correspond to the structure embedded in the source code

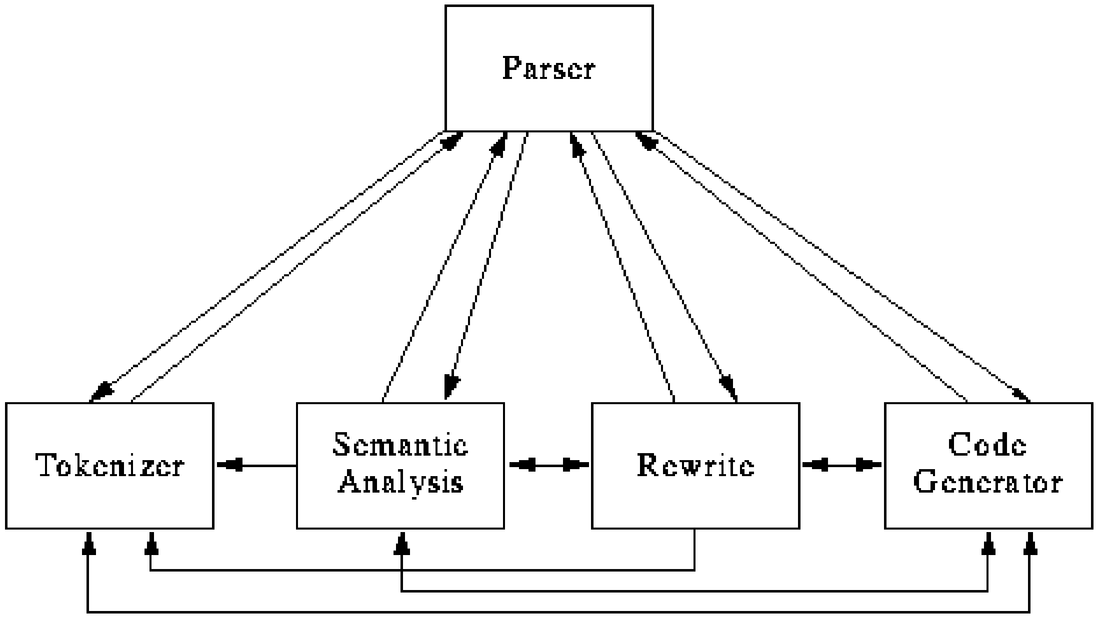

Figure 5: The extracted structure of the five main subsystems of the compiler.

(even though our system had not suffered years of maintenance). This suggests that one may not be able to directly recover "the intended design" from a software system.

Some degree of restructuring was appropriate, even for a relatively new and well-understood project. This is similar to our experience with other projects [Tzerpos 95]. When we have shown the extracted structure of a system to a system designer, there are always discrepancies with the way the designer believes the system should work. It is certainly easier, faster, and cheaper to restructure a system while the people who understand the system in detail are still involved with it.

4 Acknowledgements

References

[Mancoridis 94] Mancoridis S., Holt R.C., Godfrey M.W. A Program Understanding Environment Based on the "Star" Approach to Tool Integration, ACM CSC 1994.

[Schwanke 89] Schwanke, R. W., Altucher, R.Z., Platoff, M.A., Discovering, Visualizing, and Controlling Software Structure, Siemens Corp. Research, Inc., 755 College Rd East, Princeton, NJ 08540

[Muller 88] Muller, Hausi A. and Klashinsky, Karl, Rigi: A System for Programming-in-the-Large, Proc. of 10th International Conference on Software Engineering, Singapore, April 11-15, 1988.

[Tzerpos 95] Vassilios Tzerpos, Visualizing the source file structure of software written in C. Master's thesis, Department of Computer Science, University of Toronto, 1995.

[Penny 92] Penny, D.A. The Software Landscape: A Visual Formalism for Programming-in-the-Large. Ph.D. Thesis, Department of Computer Science, University of Toronto.

[Ramamoorthy 90] Ramamoorthy V., Chen F., Nishimoto M., The C Information Abstraction System, IEEE Transactions on Software Engineering, vol. 16(3), March 1990, pp. 325-334.

[Linos 92] Panagiotis Linos, Philippe Aubet and Laurent Dumas, "CARE : An Environment for Understanding and Re-Engineering C Programs", 5th ACM SIGSOFT Symposium on Software Development Environments, Washington D.C., December 9-11, 1992.