Dowsing: A Tool Framework for Domain-Oriented Browsing of Software Artifacts

Richard Clayton, Spencer Rugaber, and Linda Wills
Georgia Institute of Technology

Abstract

Program understanding relates a computer program to the goals and requirements it is designed to accomplish. Application-domain analysis is a source of information that can aid program understanding by guiding the source-code analysis and providing structure to its results. We use the term "dowsing" to describe the process of exploring software and the related documentation from an application-domain point of view. We have designed a tool framework to support dowsing and have populated it with a variety of commercial and research tools.

Keywords: reverse engineering, domain analysis, program understanding, software tools

1. Introduction: Dowsing

Software maintenance and enhancement activities are the largest part of total lifecycle costs. Moreover, understanding source code and change requirements dominate maintenance and enhancement effort. Consequently, methods and techniques for creating and browsing software artifacts that improve their understandability can significantly aid the overall software development process. It is the thesis of our work that effective access to software artifacts is enabled when that access is organized around the structure of the problem that the software is solving: its application domain.

The term *browsing* is often used when a software engineer visually examines existing code and documentation artifacts. Browsing can be supported by software tools such as text editors, screen paginators, or text searching tools. Even in the case of advanced source code browsers, however, the browsed material is organized around its syntactic structure; that is, how the programming language statements are nested and parsed. This tends to obscure details of how the program actually solves an application-domain problem.

We have introduced the term *dowsing* for the process of exploring software artifacts based on the structure of the software's application domain [3][4]. We are investigating how to use application-domain information to generate, organize, and present artifacts to software maintainers.

A *domain* [10] is a problem area which can be characterized by its vocabulary, common assumptions, architectural solution approaches, and literature. *Domain analysis* "is an attempt to identify the objects, operators, and relationships between what domain experts perceive to be important about the domain [9]."

To facilitate dowsing, various forms of automated support are needed, including the following:

- queries posed in terms of the domain vocabulary and concepts;
- queries about relationships between concepts or actors in the domain;
- identification of the use of typical programming solutions used in the domain, such architectural styles, patterns, and cliches;
- data type analysis linked with *concept assignment* [2] to connect programmer-defined types with domain concepts.

We have developed a domain-oriented tool framework for dowsing software artifacts. Artifacts include informal textual documents, structured documents, graphical depictions, and source code. Tools include those for analyzing and organizing textual documents and generating domain models, for analyzing source code to locate domain concepts, for constructing various visualizations of application domains and source code, and for automatically generating domain-based hypertexts.

2. A tool framework

Software artifacts are normally organized using the directory structure of a computer's file system and the syntactic structure mandated by a system's programming language. Of course, non-source-code documentation can provide domain-centric pointers to source-code constructs. Unfortunately, such pointers typically suffer from several difficulties: being out-of-date with respect to the source code, indicating specific code without specifying the nature of the connection between the documentation and the code, and not supporting *delocalized* (non-contiguous) mappings between documents and code [13]. Our approach is to provide semi-automated support for deriving domain and code models from the actual software artifacts and for exploring the software that is organized around the domain model.

To explore the efficacy of dowsing, we have designed a tool framework, called a *Dowser*, and have populated it with a variety of commercial and research tools. The software maintainer uses the Dowser primarily as an exploration environment which provides access to source code, textual documents, and tool-specific structured reports and diagrams. Besides providing access to existing artifacts, the Dowser is also capable of controlling a variety of analysis tools to generate further artifacts in the form of reports and graphical depictions.

As illustrated in Figure 1, there are several components in the Dowser tool framework.

- source-code-based tools for analyzing programs;
- domain-based tools for constructing application domain models from textual and diagrammatic documents;
- repository tools for maintaining the conceptual and logical models constructed by the domain-modeling and program-analysis tools;
- hypermedia tools for constructing linkages between the domain and program models and generating presentations, both graphical and textual, of them;
- a user interface with which the software maintainer explores the software artifacts.

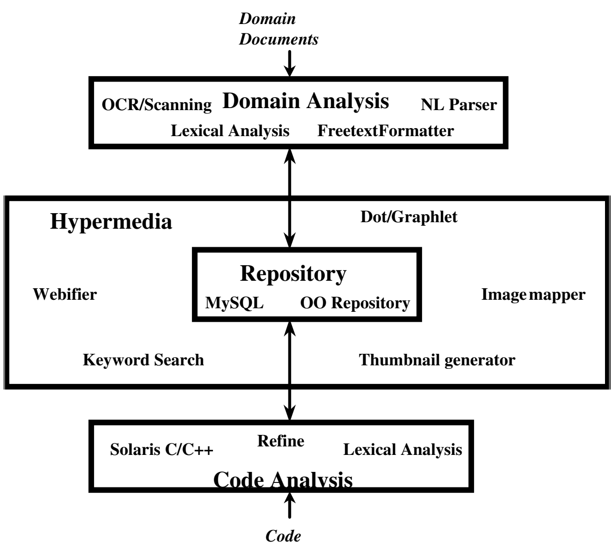

Figure 1: Dowser Tool Framework

3. Using the Dowser

3.1 Domain modeling

The core of our approach to program exploration is an application-domain model of the software. We have provided tools to help derive the domain model from software design artifacts and documents written about an application problem. An important activity in comprehending the problem is being able to view the domain model and to explore the relationships among the domain concepts. We create an application-domain model in two steps. First, various descriptions of the domain are analyzed using word-frequency analysis and noun-verb analysis to extract possible object classes and associations, as suggested by Abbot [1]. Second, the extracted information is organized into a coherent application-domain model by applying the Object Modeling Technique (OMT) [12].

3.2 Program analysis

We used SUN Microsystems' Source Browser Facility to build a procedural-level model of the code We stored the Source Browser data in a relational database and performed our analyses using SQL queries. SQL allowed us to quickly and easily formulate code analyses with a minimum of programming.

3.3 Architectural bridging

Code models and domain models must be interlinked to enable effective domain-based program understanding. Direct linkages are difficult to make because of the conceptual distance between the code and domain models. Consequently, we use software architecture as a stepping stone between the two. To connect the domain model with source code, we need an intermediate architectural model. We performed three primary architectural analyses to construct one.

Invocation analysis. Invocation analysis explores relationships between functional components and evaluates module coherence. It can also represent intercomponent communication if the modules denote architectural components. From a domain-model perspective, the inter-object communication suggests associations among objects.

Type analysis. Type analysis connects programmer-defined types with domain concepts. Analysis of source code data types can be used to develop an object-oriented representation of the code, even if the source language does not natively support object-oriented constructs. *Part-of* associations (aggregations) may be derived from analysis of the hierarchical structure of the types.

Coupling analysis. Coupling analysis enables coarse-grain relationships between functional domain concepts to be understood. The amount of communication between modules can be used as a measure of the strength of the association between them, which in turn provides a measure of the coupling between two modules or objects.

3.4 Dynamically generated hypermedia

The terminology established by specification and design documents is often carried over into later portions of a software development project; these early-occurring documents set the domain of discourse for the project. Conversely, words appearing in the program text may have come about due to their appearance in preliminary project documents, and understanding the context in which these words occur may help explain their appearance in the program text. To provide the associated linkage tools that connect the domain and code models, we have built a hypermedia tool that dynamically generates web pages linking the evolving domain model, code analyses, and existing documentation. The tool supports keyword searches and provides annotated image maps relating common terminology across these models and documents.

The user can submit a keyword-search request by entering the appropriate keywords into an HTML form. The keyword query is parsed by a CGI script associated with the form and applied to a cross-document concordance. The result is a set of pages containing the words matching the keyword query. These pages are then presented in a response page of "thumbnail" representations of the retrieved pages. Each keyword on the thumbnail pages is color coded to provide a visual depiction of the pages' overall match density. Clicking on a thumbnail page retrieves the associated full-sized pages with matching terms highlighted. By using the browser's forward and backward buttons, as well as by spawning new pages, the maintainer can create a detailed set of connections between the terminology appearing in program text and in preliminary project documents.

3.5 Presentation of high-level graphical models

In cases where tabular arrangements of Dowser-generated data are appropriate, relational databases queries produce acceptable output. More complicated data presentations, such as the call graphs produced by invocation analysis, require further processing by external programs. For example, the call-graph generator uses an **awk** script to read the results of the call-graph analysis and write a description of the call graph in the language used by the **dot** graph-drawing program [8].

4. Comparison with other work

Recognizing and exploiting structure in domain, architecture, and code is the central problem in supporting program understanding. Our work draws on and benefits from a set of approaches differing at the level of abstraction at which they address the problem.

LaSSIE [5] explicitly embodies domain, architecture, and code knowledge about a system. LaSSIE gets domain knowledge through reverse domain-engineering, while our work gets its domain knowledge through domain analysis. LaSSIE supports maintenance and enhancements of a single, existing system, while our work develops a framework which can be instantiated to create several different but related systems. LaSSIE is interested in promoting reuse at module and subprogram level, while our work is interested in promoting reuse at the architectural level.

DESIRE [2] makes use of a rich domain model and neural-network learning to establish connections between the domain model and the code. Our work supports the reverse engineer in acquiring and refining the domain model as well as its connection to the code being explored.

Harris, Rubenstein, and Yeh [6] extract architectural-level features from source code using pattern-matching techniques. Architectural extraction and our work are mutually supportive. An architectural description of the software provides an important midpoint between the domain description and the code, and domain knowledge can direct and sharpen pattern matching.

DECODE [11] uses cooperative, bottom-up code analysis to create object-oriented descriptions of code. The reverse engineer cooperates with DECODE to create and organize objects. To a certain extent domain concepts are embodied in the code analysis, but when these reach their limits, the reverse engineer is thrown back on whatever domain resources are available outside of DECODE.

I-DOC's research on interactive software explanation [7] influenced our work on dowsing and linkage tools. We share the goals of supporting interactive querying to explore results of code analyses and of using hypermedia to support browsing annotated software artifacts. I-DOC supports task-oriented software understanding, our work supports the complementary process of domain-oriented exploration.

5. Future work

Using domain knowledge to support automated program understanding. One of our goals is to explore how domain knowledge can be used to further the goal of program understanding. One way of accomplishing this is to have the domain model expressed in a form easily accessible to the tools that undertake the source code analyses. The domain model should guide the exploration process by generating a set of expectations, and, when the code analyses detect constructs satisfying the expectations, the object-oriented domain model can be "instantiated" with the results.

Dynamic view generation. We need to extend our document generation tools to handle document diagrams, automatically generating HTML image maps from diagrams.

Architectural analysis and visualization. An architectural abstraction of the program can bridge the gap between the domain model and the results of code analysis. This raises several issues, including the relation between the application-domain model and the architecture, how architectural styles can be connected to program descriptions, and what annotation mechanisms should be used to make these relationships explicit and allow them to evolve as the domain model is refined.

Knowledge sources. We need to extend our ability to capture, structure and access external knowledge. One possibility is to extend the scope of lexical analysis. Another is to encode programming knowledge as a set of program plans and use cliche recognition to find them in the code.

6. Conclusions

Our research hypothesis is that domain-based browsing, dowsing, is a more effective way to access software artifacts for the purpose of gaining understanding than is traditional, source-code-based browsing. To test this hypothesis we have designed a tool framework, the Dowser, and populated it with a variety of commercial and research tools.

We then used the Dowser to look at two applications. The first study examines the Mosaic web browser. Mosaic is a mature, publicly available browser whose 100,000 lines of source code are written in the C language, and for which some documentation is available. We have built a domain model for web browsing and used it to organize various artifacts related to Mosaic. The second case study examines the domain of software loader-verifiers (SL/Vs) which download binary executables into mobile, embedded systems such as those in tanks or airplanes. We examined an SL/V program written in 10,000 lines of Ada.

We have informally found that the dowsing framework is a comprehensive and integrated way to deal with the disparate forms of loosely related software artifacts. Nevertheless, our underlying hypothesis that domain-oriented browsing is a more effective approach to program understanding than traditional browsing needs to be validated through actual use in performing maintenance tasks, which we can now use the Dowser to test.

Acknowledgments

This effort was sponsored in part by the Army Research Laboratory under contract DAKF11-91-D-0004-0055 and by the National Science Foundation under grant CCR-9708913. It was also sponsored by the Defense Advanced Research Projects Agency, and the United States Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-2-0229. We would also like to thank the U. S. Army Tank and Automotive Command for supplying the SL/V software to us, and Lyman Taylor for helping with the code analysis.

References

[1] Russell J. Abbott. “Program Design by Informal English Descriptions.” *Communications of the ACM*, 12(11): 882-894, November 1983.

[2] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. “Program Understanding and the Concept Assignment Problem.” *Communications of the ACM*, 37(5):72-83, May 1994.

[3] Richard Clayton, Spencer Rugaber, Lyman Taylor, Linda Wills, A Case Study of Domain-based Program Understanding, *5th International Workshop on Program Comprehension*, Dearborn, Michigan, May 28-30, 1997.

[4] J-M. DeBaud, B. Moopen, and S. Rugaber. “Domain Analysis and Reverse Engineering.” *Proceedings of the 1994 International Conference on Software Maintenance*. Victoria, British Columbia, Canada, September 19-23, 1994, 326-335.

[5] P. Devanbu, R. Brachman, P. Selfridge, and B. Ballard. “LaSSIE: A Knowledge-Based Software Information System,” *Communications of the ACM*, 34(5):35-49, May 1991.

[6] David R. Harris, Howard B. Reubenstein, and Alexander S. Yeh. “Recognizers for Extracting Architectural Features from Source Code.” *Second Working Conference on Reverse Engineering*, pp. 252-261, July 1995.

[7] W. Lewis Johnson and Ali Erdem. “Interactive Explanation of Software Systems.” *Automated Software Engineering*, Volume 2, 1996.

[8] E. Koutsofios and S. C. North. “Drawing Graphs with dot.” AT&T.

[9] James M. Neighbors. “Draco: A Method for Engineering Reusable Software Components.” *Software Reusability / Concepts and Models,* volume 1, Ted J. Biggerstaff and Alan J. Perlis, editors, Addison Wesley, 1989.

[10] Rubén Prieto-Díaz and Guillermo Arango. *Domain Analysis and Software Systems Modeling*. IEEE Computer Society Press, Los Alamitos, California, 1991.

[11] Alex Quilici and David N. Chin. “DECODE: A Cooperative Environment for Reverse-Engineering Legacy Software.” *Second Working Conference on Reverse Engineering*, pp. 156-165, IEEE Computer Society Press, July 1995.

[12] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[13] E. Soloway, J. Pinto, S. Letovsky, D. Littman, and R. Lampert. “Designing Documentation to Compensate for Delocalized Plans.” *Communications of the ACM,* 31(11): 1259-1267, November 1988.