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

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

Abstract

Program understanding is an essential part of software maintenance and enhancement activities that relates a computer program to the goals and requirements it is designed to accomplish. Understanding techniques that rely only on source-code analysis are limited in their ability to derive this relationship. Application-domain analysis is another 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 tools infrastructure to support dowsing and populated it with a variety of commercial and research tools. We have used the infrastructure and tools to explore software in two application domains: web browsers and software loader/verifiers for embedded systems.

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 [4]. Moreover, understanding source code and change requirements dominate maintenance and enhancement effort [14]. Consequently, methods and techniques for creating and browsing software artifacts that improve their understandability can significantly aid the overall software development process. Yet most artifacts, whether textual documents or source code, are accessed either sequentially (in the case of documents) or hierarchically, based on programming language syntax rules (in the case of source code). 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 artifacts. Browsing can be supported by software tools such as text editors, screen paginators, or text searching tools. Recently, hypertext has become a popular way to peruse documents by providing non-sequential, cross-referenced access, both within and between documents. In the case of source code, more sophisticated tools are available, including those which perform extensive analyses and store the results in a database to support complex queries. 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 may not make obvious to the reader the 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 [9,11,29]. *Webster's Third New International Dictionary* defines "*dowsing*" as "to seek something with meticulous care especially with the aid of a mechanical device." We are also interested in mechanically supported search, in this case, within and among software artifacts. In particular, we are investigating how to use application-domain information to generate, organize, and present artifacts to software maintainers.

We have developed a domain-oriented tools 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 for domain concepts, for constructing various visualizations of application domains and source code, and for automatically generating domain-based hypertexts.

In this paper, we will first present a scenario of the dowsing activities we would like our tools to support and the role of domain analysis in intelligent exploration of software. Section 2 provides an overview of the dowsing tools framework while Section 3 shows how several dowsing activities from two case studies are supported by the framework. Section 4 summarizes related research. Section 5 describes areas where further research is required.

Scenario

Imagine a situation in which a software engineer is maintaining and enhancing a software simulator for hardware systems such as massively parallel processors, caches, pipelined systems, or superscalar architectures. Simulations of these systems are built by researchers and engineers as an integral part of designing and understanding new architectural innovations. Suppose the engineer plans to extend and apply an existing sequential simulator for a parallel processor. The following are some of the activities and questions that might arise.

- Identifying types of actors: The engineer is interested in domain-oriented types, such as whether the system clock is discrete or continuous and what types of parallel instructions are simulated, rather than focusing solely on low-level data types (e.g., integers or strings) or common abstract data types (e.g., linked lists or hash tables).
- Understanding the domain-specific software architecture: A common problem in simulating parallel systems is correctly modeling and coordinating concurrent events using a sequential process [21,35]. There are standard software architectures used to solve this problem and to ensure that events are not generated or processed out of order. Two of the most common are *event-driven* (in which a centralized event agenda keeps track of pending events and orders the event processing) and *synchronous* (in which event generators, such as processing nodes, are simulated in lock-step with a global clock). An engineer is interested in what style of architecture is used in the simulation at hand, because this affects how new events are added to the simulation.
- Checking domain-oriented constraints: In hardware simulation, there are constraints on the chronology of simulated events which are satisfied using standard mechanisms. It would be useful to answer a question about how a given domain-oriented constraint is satisfied by connecting it to the program mechanism that maintains the constraint. For example, there is a constraint in simulating parallel systems that events are processed in chronological order (that if several events are pending, no event with a later timestamp is processed before one with an earlier timestamp). If an event-driven simulation is being used, this constraint is often satisfied by using a priority-queue implementation of the event agenda, with arrival time as the priority. Understanding how the priority queue implementation relates to event handling is a key prerequisite to successfully modifying the simulator.

The scenario suggests various forms of automated support such as the following.
- queries posed in terms of the domain vocabulary and concepts;
- results of queries portrayed graphically to give an idea of the relative distribution of the concepts of interest;
- 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* [3] to connect programmer-defined types with domain concepts.

Domain Analysis

A *domain* is a problem area. Typically, many application programs exist to solve the problems in a single domain. Arango and Prieto Diaz [26] give the following prerequisites for the presence of a domain: the existence of comprehensive relationships among objects in the domain, a community interested in solutions to the problems in

the domain, a recognition that software solutions are appropriate to the problems in the domain, and a store of knowledge or collected wisdom to address the problems in the domain. Once recognized, a domain can be characterized by its vocabulary, common assumptions, architectural approach, and literature.

Domain analysis, according to Neighbors [24], "is an attempt to identify the objects, operators, and relationships between what domain experts perceive to be important about the domain." As such, it bears a close resemblance to traditional systems analysis, but at the level of a collection of problems rather than a single one. Domain analysis is an emerging research area in software engineering. It is primarily concerned with understanding domains in order to support initial software development and reuse, but its artifacts and approaches will prove useful in support of software maintenance and enhancement tasks as well.

In order for domain analysis to be useful for software development, reuse, or maintenance, the results of the analysis must be captured and expressed, preferably, in a systematic fashion. Among the aspects that might be included in such a representation are domain objects and their definitions, including both real-world objects like *global clock* and concepts such as *asynchronous event handling*; solution strategies, plans, and architectures like *priority queues;* and a description of the boundary and other limits to the domain like *byte vs. word oriented simulators*.

2. A Tools 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 [31]. 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 tools framework, called a *Dowser*, populated it with a variety of commercial and research tools, and performed semi-automated domain analyses on two example domains. The resulting analyses and associated source code serve as the raw materials upon which the tools act. The tools can accept requests in the form of keyword searches, SQL queries, hypertext link following, and graphical selections. Output consists of a variety of diagrams or textual accesses into the source code and documentation.

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 and generating further artifacts in the form of reports and graphical depictions. Of course, these derived documents should also promote further exploration and must therefore support the same kinds of access and queries as the original artifacts.

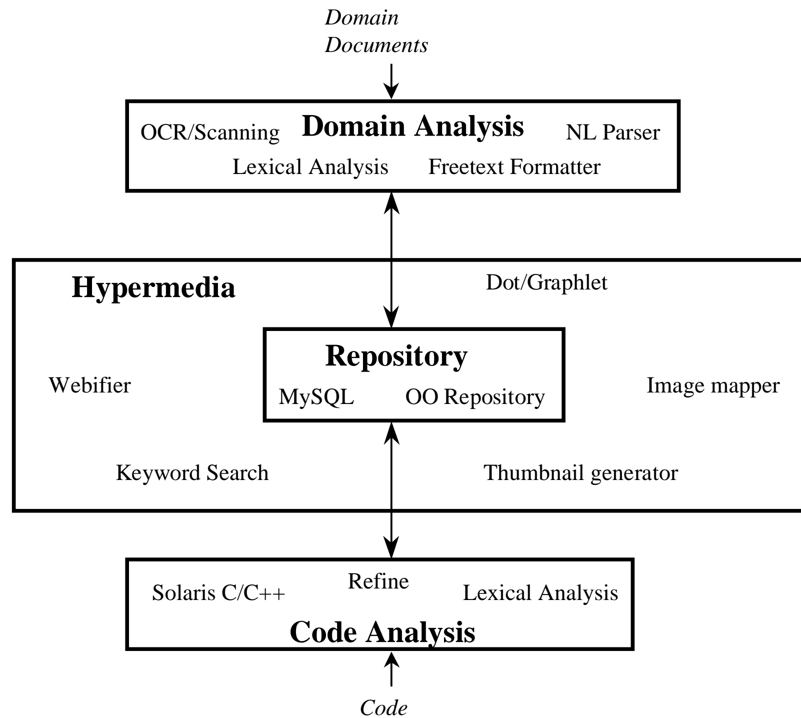

Figure 1: Dowser tools framework

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

- source-code-based tools for analyzing the program;
- domain-based tools for constructing a model of the application domain 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.

With the exception of the user interface, each of these components is represented by multiple tools. Hence, we use the phrase *tools framework* rather than merely *tool suite*.

Program Analyzers

There are two types of program analysis tools: traditional, language-based analyzers, available commercially, and value-added research prototypes that relate the program-based information to higher-level concepts, primarily architectural views of how the program realizes non-functional requirements such as performance or security. For the first type, we have used two commercial tools, the Software Refinery tool suite (Refinery) [28] and the Source Browser Facility (SBF) [33] that is provided by SUN Microsystems with its Solaris C and C++ compilers. For the second type, we have built tools to compute three relations: how components interact by invoking each other, how data types are defined in terms of each other, and how modules are coupled to each other.

Domain Analysis Tools

To support dowsing, a software maintenance tool must provide access to software artifacts in terms of domain concepts. Two ways of conveying such concepts are textually and graphically. Textual expression of domain concepts makes use of domain vocabulary. Graphical depictions convey relationships among domain concepts.

The key differentiator for the Dowser over traditional browsers is that the user can explore the artifact base in terms of application-domain concepts as well as traditional textual and source-code methods. Domain access is provided in terms of an explicit domain model comprising two parts: a vocabulary and an OMT static model [30]. The vocabulary consists of key domain concepts from which keyword searches can be composed. It also serves as the starting point for the construction of the OMT model. The OMT model consists of domain actors and objects and the associations among them. It can naturally be conveyed in graphical form and consequently supports visual exploration.

We assume that domain knowledge is encoded in various textual descriptions and diagrams such as might be found in a requirements document. Consequently, such descriptions must be converted into a formal model of the domain. Although this cannot be done entirely automatically, we have developed or made use of tools for scanning in text and converting it to ASCII characters [25], formatting the resulting text stream into structured HTML, and performing several types of lexical analyses. These lexical analyses include breaking text into words and filtering out uninteresting words, parsing it to determine parts of speech, detecting candidate objects, actions, and states, and computing word frequencies. Together, these provide the analyst with an initial approximation of the most important terms in the domain vocabulary.

Repository Tools

At the core of the Dowser is a repository for entering and managing the conceptual and logical models generated by the other tools. We have explored two such tools, the Refinery's object-oriented repository [28] and a public-domain, relational, record-management tool called MySQL [23]. The former has the natural advantages of being congruent to the domain model and integrated with one of the code analysis tools. The latter approach, similar to that taken by CIA and CIA++ [8,16], has the advantages of the relational algebra query language and a smaller conceptual footprint. Together these tools provide a powerful lever to the programmer trying to understand a software system.

Hypermedia-based Linkage Tools

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. For example, imagine the following situation related to an Internet browser. A software maintainer trying to understand the browser source code comes across a significant amount of code related to the browser's page caches. But there is no mention of caching in the any requirements document for web browsing. If anything is mentioned at all, it is that page retrieval should be rapid. This is a non-functional requirement, and caches are one way of satisfying it. An architectural description of the browser that views its modules in terms of how they support performance will feature the cache management software as a major component. The caching code can be related to the cache management component, which exists to satisfy a performance requirement from the application domain.

Constructing an architectural stepping stone requires that programming knowledge (such as how to provide increased performance using caching) be available. For the work we are describing in this paper, we assume that the knowledge exists in the mind of the software maintainer. Nevertheless, we need to provide some passive mechanisms for entering and maintaining the linkages within the computer. And we need a way of providing intelligent presentations of the information to the maintainer. By "intelligent" we mean that the presentations enable the user to follow the links. Examples of such presentations that we construct are dynamically generated web pages, image maps, and thumbnail sketches of the results of keyword-search requests.

User Interface

A software maintainer dowses a software system using a model of the program's application domain. The model gives a graphical depiction of the major entities and relationships in the domain. Not all of the entities and relationships are realized in any single program in the domain, so it is important for the Dowser to indicate how the particular program instantiates the domain model. This is done by giving the software maintainer access to the underlying program code and documents via domain concepts. Such access can take the form of keyword searches,

SQL queries, web-page links, or image maps selections. We use a front end consisting of HTML forms and CGI scripts to provide this interface.

3. Examples of Dowsing Activities

Now that we have presented an overview of the Dowser tools framework, it is illustrative to consider how dowsing activities are supported by this framework. This section looks at several dowsing examples in the context of two case studies of our approach on significant applications. The first study examines the Mosaic web browser [22]. A web browser is a tool for exploring and reading documents on the Internet. 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). An SL/V is a tool for downloading binary executables into mobile, embedded systems such as might be found in tanks or airplanes. We obtained an example SL/V and its related documents from the U.S. Army Tank and Automotive Command. The program is written in the Ada language and comprises 10,000 lines of code. The documentation is available only in hardcopy form, so we made use of scanning and character recognition tools for converting it to electronic form [25].

Domain Modeling Activities

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 the 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 three steps. First, various descriptions of the domain are analyzed to extract relevant information about the domain. Second, the extracted information is organized into a coherent application-domain model by applying the Object Modeling Technique (OMT) [30]. Third, the constructed OMT Object Model is instantiated in a machine-manipulatable form using the Software Refinery tool set.

In our Mosaic case study, we perform object analysis on textual descriptions of web browsers (such as Chapter 1 from [6]) by performing word-frequency analysis and object extraction. Frequency analysis is a first pass over the descriptions to determine important words. After eliminating noise words such as “the”, the remaining words are counted; high frequency words are assumed to be more important than low frequency words. For example, for Mosaic, the frequency analysis identifies the words “server,” “script,” “browser,” and “client” as the most salient. We then manually perform a noun-and-verb analysis on the textual description to extract possible object classes and associations. This step was suggested by the original approach to object-oriented analysis proposed by Abbot [1].

The second step in creating the application-domain model is to take the objects and associations found in the first step and organize them into a coherent model. We chose to organize around OMT because of its familiarity in the software engineering community. OMT includes several graphical notations for specifying models. Its Object Model expresses the important domain classes and the inter-class associations. For example, Figure 2 shows the OMT diagram we derived in the Mosaic case study.

Figure 2: OMT model of web browser domain.

The third step in creating an application-domain model is to reify the OMT Object Model in a machine-processible form. We chose Refine, part of the Software Refinery [28], as our modeling language. The Refinery is a large, Lisp-based program suite oriented towards program analysis and transformation. The Refinery is well suited to support dowsing because it can be used to represent both the application domain and the code under analysis. The domain model is represented using the Object Base, the Software Refinery subsystem responsible for holding the results of program analysis. In addition, we use its Dialect [13] tool to build parsers that translate between Refinery internal and external textual representations of the Object Model.

Program Analysis Activities

We have performed our program analyses using the Source Browser Facility (SBF) that is provided by SUN Microsystems with its C and C++ compilers [33]. This facility builds an initial procedural-level model of the code from which we access raw data necessary for our code analyses. After implementing a few analysis programs by hand using the Icon programming language [17], we stored our raw 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, but with a cost in naturalness in expression.

Architectural Bridging Activities

High-level models are useful when describing and comprehending complex systems such as programs. A model's utility comes from its ability to abstract away the details of the system it describes. In contrast, the source code of a program is full of such details. To connect the domain model with source code, we need an intermediate architectural model that abstracts away these details.

We performed three primary architectural analyses:

1. *invocation analysis* allows relationships between functional components to be explored;
2. *type analysis* connects programmer-defined types with domain concepts;
3. *coupling analysis* enables coarse-grain relationships between functional domain concepts to be understood.

Invocation Analysis.

Invocation (or call graph) analysis determines which subprograms invoke others in a program. If subprograms are grouped into modules, then the invocation between subprograms in different modules can be used to represent communication between the modules. Also, if a comparison between intra- and inter-module invocations is developed, module coherence [32] can be evaluated. If the modules denote architectural components, intercomponent communication is represented. From a domain-model perspective, the inter-object communication serves to suggest associations among objects.

Type Analysis

Analysis of source code data types may be used to develop an object-oriented representation of the code, even if the source language does not natively support object-oriented constructs [7]. *Part-of* associations (aggregation data structures) may be derived from analysis of the hierarchical structure of the types. For example, a datatype's subcomponent may refer to a collection. If a "member" subprogram associated with the datatype retrieves elements from the collection, that indicates a *part-of* association. This deduction can be made if the selector interface to the object (or abstract datatype) in question returns the elemental type of the collection instead of the collection itself and there are no selectors which do return the collection. In this case the elements of the collection have a relationship to the object. The collection is a manifestation of this relationship's many-to-one property. This analysis involves observing the subprograms' formal arguments and return types and associating subprograms to the abstraction interface of one particular type. Additionally, datatypes serving as collections have to be identified.

Coupling Analysis

The amount of communication between modules can be used as a measure of the strength of the association between them. A measure of the coupling [32] between two modules or objects is indicated by the strength of this association [5]. We interpret coupling as common, global-variable access; two subprograms accessing the same global are considered coupled. Determining the degree of coupling within a relational framework is somewhat complicated, particularly when the calculations involve more than simple counting and arithmetic. Fortunately, this problem is easily solved by either adding external subprograms to the SQL interpreter or dumping the data to a file and using an external analysis program.

Presentation Activities: Exploration using 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 in preliminary documents may help explain their appearance in the program text.

As an initial step toward providing the associated linkage tools that connect the domain and code models, we have built a prototype hypermedia tool that

- dynamically generates web pages linking the evolving domain model, code analyses, and existing documentation;
- supports keyword searches;
- provides annotated image maps relating common terminology across these models and documents.

This hypermedia tool performs lexical analyses on the documents associated with software and on the source code itself, creating a concordance of the documents. It then answers keyword queries with visual information on the frequency and relative location of the keywords across the documents. By requesting keyword searches, the

maintainer can examine document pages according to the words they contain, giving the maintainer a focused and oriented view into the documents.

The user can submit a keyword-search request by entering the appropriate keywords into an HTML form. In the example shown in Figure 3 (from the SL/V case study), the maintainer asked to see all documents currently associated with the SL/V software system that contain the words “software” and “download.”

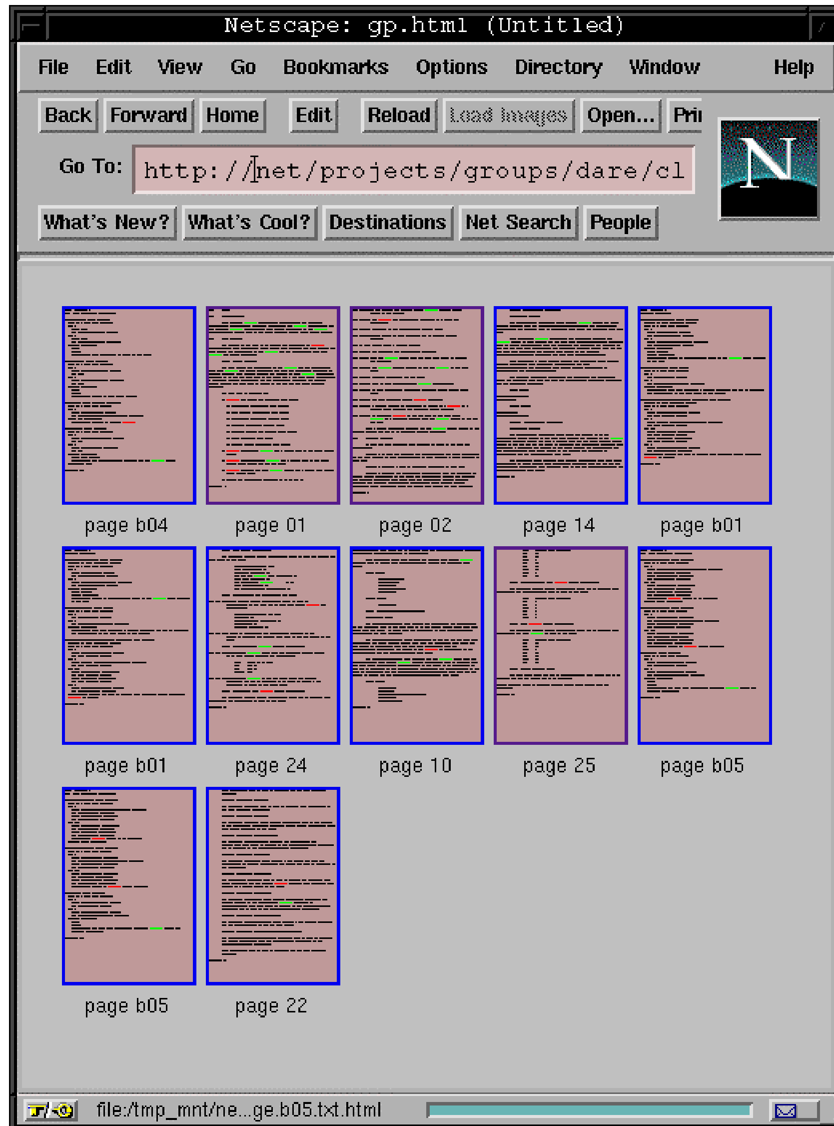

Figure 3: Thumbnail presentation of matching pages.

The keyword query is parsed by a CGI script associated with the form and applied to the 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 as shown in Figure 3. 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.

Presentation Activities: High-Level Graphical Models

Although generated data can be used as inputs for further analyses, usually the user of the Dowser also wants to view the results of an analysis. In cases where tabular arrangements are appropriate, relational databases queries are good enough to produce the output. However, more complicated arrangements, such as call graphs, exceed the output capabilities of most databases, and require further processing, such as external programs linked into the database. However, usually it is easier to have the database write the analysis results as text into a file, and then let external programs massage the data file. This is what was done, for example, by the call-graph generator, which uses a simple **awk** [2] 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 [20]. The results can be seen in Figure 4.

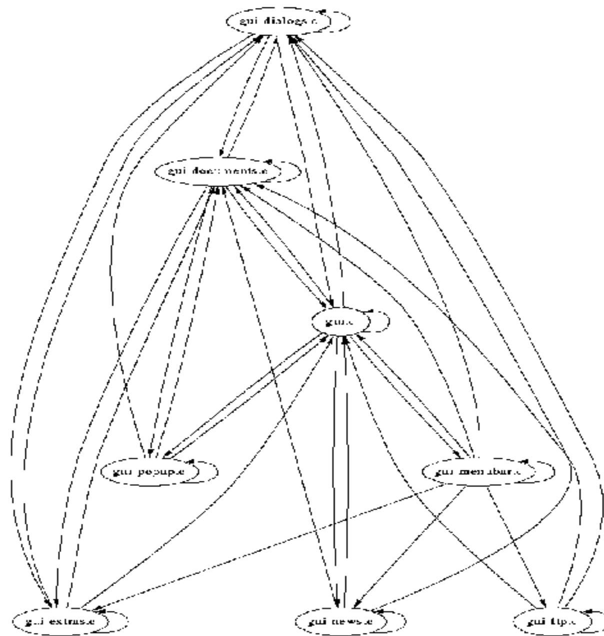

Figure 4: A call-graph that aggregates subprograms into their defining files.

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.

Our work is similar to approaches taken at the highest level of abstraction, such as LaSSIE [12]. LaSSIE is a knowledge-based information system designed to support maintenance and enhancements on a large software system. The LaSSIE knowledge base uses frames to represent information about the software's domain, architecture, and code. A formally defined inheritance relation between frames provides a semantics exploited by the knowledge base to produce useful answers to imprecise questions. LaSSIE and our work have similar means, while they differ as to their ends. Both projects recognize the importance of explicitly embodying domain, architecture, and code knowledge about a system. Differences in means are largely a result of process. For example, LaSSIE extracts its knowledge from a single, well-documented system, while our work extracts its knowledge from a set of smaller, functionally similar programs; thus, LaSSIE gets domain knowledge through reverse domain engineering, while our work gets its domain knowledge through domain analysis. The ends of the two projects are also different. 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. In one sense,

LaSSIE is interested in promoting reuse at the sub-architectural (e.g. module or subprogram) level, while our work is interested in promoting reuse at the architectural level.

A high-level approach which influenced our work is that taken by DESIRE [3]. DESIRE was the first software understanding systems to make use of a rich domain model. This model contains machine-processable forms of design expectations for a particular domain, as well as informal semantic concepts. It includes typical module organizational structures and the common terminology associated with programs in a particular application domain. Techniques for recognizing patterns of organization and linguistic idioms in the program are used to generate expectations of the concepts associated with these patterns. Linking the patterns found in the program to conceptual structures in the domain model is based on the correlation (experientially acquired) between the mnemonic names used in the program's procedures, variables, and comments and the terminology recorded in the domain model. While DESIRE uses neural-network-based learning techniques to draw and refine connections between the domain model and the code, our work supports the reverse engineer in acquiring and refining a model of the domain itself as well as its connection to the code being explored.

At a lower level than LaSSIE or DESIRE, Harris, Rubenstein, and Yeh [18,34] describe their pattern-matching techniques for extracting architectural-level features from source code. Patterns describe architectural entities and their interconnections; pattern matching is suitably fuzzy to allow for matches in the presence of obscure or missing features. The resulting set of matched entities and interconnections can then be combined, again under fuzzy pattern matching, to form architectural features. Our work and the work reported by Harris, Rubenstein, and Yeh are mutually supportive. An architectural description of the software provides an important midpoint between the domain description and the actual code. To the extent that this kind of pattern matching can construct such architectural descriptions, it could fill an important need in our work. In the other direction, the domain knowledge made available in our work could serve to direct and sharpen pattern matching.

Approaches still lower than those previously described are taken by such systems as DECODE [27], which uses cooperative, bottom-up code analysis to create object-oriented descriptions (that is, data plus operations) of existing code. The code is analyzed using pattern-matching techniques scalable over large bodies of code. DECODE automatically generates some objects from the concepts matched in the code, but the reverse engineer cooperates with DECODE to create and organize further objects. What is missing from DECODE is the explicit connection between a set of objects derived from code and concepts in the problem domain. This lack can be seen in the nature of the questions that DECODE can answer; it can do well on code-up questions (e.g., to what object does this code belong?) but less well on concept-down questions (e.g., does any input validation get done?). To a certain extent domain concepts are embodied in the patterns that can be recognized, but when these reach their limits, the reverse engineer is thrown back on whatever domain resources are available outside of DECODE.

Our work on dowsing and linkage tools has also been influenced by Johnson and Erdem's research on interactive software explanation [19] as demonstrated in the I-DOC tool. We share the goals of supporting interactive querying to explore results of code analyses and of using hypermedia to support browsing annotated software artifacts. Johnson's focus has been on supporting queries that are typically made in the context of performing specific maintenance tasks. While Johnson's work supports task-oriented software understanding, our work focuses on supporting the complementary process of domain-oriented exploration.

5. Future Work

Using Domain Knowledge to Support Automated Program Understanding

One of the original goals of our research was to explore how domain knowledge could 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. In particular, we have expressed our domain model in the Refine language and stored it in the Software Refinery's object-oriented repository. We now need to take the kinds of code analyses that we performed using the SBF tools and do them using the Refinery's code analysis capabilities. In particular, 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. So far, the process has been applied manually. We would like to continue working towards its automation.

Program Visualization

We have performed three kinds of architectural linkage analyses: invocation analysis, type analysis, and coupling analysis. All of these generated large amounts of data that we have viewed with the **dot** graph-drawing tool. While **dot** is robust enough to deal with large graphs, the results are not necessarily useful to the analyst. We would like to look at several software visualization techniques that might further the analyst's understanding.

- Make use of the third-dimension and allow the analyst to navigate through the resulting visualizations to detect patterns;
- Apply some experimental algorithms that propose node-groupings that might suggest modules or architectural components to the analyst [10];
- Make use of information from actual executions to color an otherwise confusing graph, indicating "hot spots," similar to an MRI scan of a human brain taken while the subject is performing some task.

Dynamic View Generation

Our document generation tools make a set of documents easier to access by converting them to HTML, providing internal links, and offering a dynamic, keyword-search capability. One relatively small missing piece is the ability to take an existing diagram and generate an HTML image map from it. The resulting page would allow the software maintainer, using the diagram, to navigate to relevant documentation. Currently, we generate the image maps by hand.

Architectural Analysis and Visualization

The key to domain-driven program understanding is bridging the gap between the conceptual domain model (the application description) and the results of code analysis. We believe an architectural abstraction of the program can bridge this gap. This raises several issues:

- What is the relation between the application-domain model and the architecture?
- How can architectural styles [15] be connected to program descriptions resulting from standard code analyses?
- What annotation mechanisms should be used to make these relationships explicit and allow them to evolve as the domain model is refined and program descriptions emerge?

As the domain model, architectural description, and program description emerge and evolve, how are their relationships captured in annotations? What representations are useful for annotations and what types of inferencing on annotations are needed? During the refinement and elaboration of the domain, architecture, and program models, expectations are generated about how they are connected. It is likely that a truth-maintenance system will be needed to handle confirmation and refutation of expectations.

Knowledge Sources

The most ambitious remaining task is to deal with the issue of external knowledge. In order to make a tangible connection between a requirements document mentioning "performance" and source code implementing a "cache", there needs to be knowledge somewhere that a cache is one way of improving performance. How should this knowledge be structured and accessed? One possibility is to extend our concordance approach to look for common word use in requirements, architecture, and code documents. Another is to actually encode programming knowledge, in the form of a set of "program plans" and use cliche recognition techniques 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 tools framework, the Dowser, and populated it with a variety of commercial and research tools. We then used the Dowser to look at two application. And 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 is 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

The effort reported on in the paper was sponsored by the Army Research Laboratory under contract DAKF11-91-D-0004-0055. 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] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.

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

[4] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[5] Grady Booch. *Object-Oriented Analysis and Design with Applications, second edition*. Benjamin/Cummings, 1994.

[6] Steven E. Brenner and Edwin Aoki. *Introduction to CGI/PERL*. M & T Books, 1996.

[7] Eric J. Byrne and Gokul V. Subramamian. “Deriving an Object Model from Legacy Fortran Code.” *Proceedings of the International Conference on Software Maintenance*, Monterey, California, November 4-8, 1996, 3-12.

[8] Y. F. Chen and C. V. Ramamoorthy. “The C Information Abstractor.” *Proceedings COMPASC 86*, IEEE, 291-298, 1986.

[9] 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.

[10] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. “The Complexity of Multiway Cuts (Extended Abstract).” *24th Annual ACM STOC,* May 1992, Victoria Canada, pages 241-251.

[11] Jean-Marc DeBaud, Bijith M. Moopen, and Spencer 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.

[12] Premkumar Devanbu, Ronald J. Brachman, Peter G. Selfridge, and Bruce W. Ballard. “LaSSIE: A Knowledge-Based Software Information System,” *Communications of the ACM*, 34(5):35-49, May 1991.

[13] *Dialect User’s Guide*. Reasoning Systems, Palo Alto, California, July 1990.

[14] R. K. Fjeldstad and W. T. Hamlen. “Application Program Maintenance Study: Report to Our Respondents.” *Proceedings GUIDE 48,* Philadelphia, Pennsylvania, April 1983, *Tutorial on Software Maintenance,* G. Parikh and N. Zvegintozov, editors, IEEE Computer Society.

[15] David Garlan and Mary Shaw. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1995.

[16] Judith E. Grass and Yih-Farn Chen. “The C++ Information Abstractor.” *1990 USENIX Conference,* 1990, 265-277.

[17] Ralph E. Griswald and Madge T. Griswald. *The Icon Programming Language*. Prentice Hall, 1983.

[18] David R. Harris, Howard B. Reubenstein, and Alexander S. Yeh. "Recognizers for Extracting Architectural Features from Source Code." *Second Working Conference on Reverse Engineering*. Linda Wills, Philip Newcomb, and Elliot Chikofsky, editors, pp. 252-261, IEEE Computer Society Press, July 1995.

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

[20] E. Koutsofios and S. C. North. "Drawing Graphs with dot." AT&T.

[21] MacDougall, M.H., *Simulating Computer Systems: Techniques and Tools*. The MIT Press, Cambridge, MA, 1987.

[22] Mosaic. `http://www.ncsa.uiuc.edu/SDG/Software/Mosaic.`

[23] MySQL. `http://www.tcs.se.`

[24] 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.

[25] *OmniPage Professional Reference Manual*. Caere Corporation, Los Gatos, California, 1994.

[26] Rubén Prieto-Díaz and Guillermo Arango. *Domain Analysis and Software Systems Modeling*. IEEE Computer Soci amitos, California, 1991.

[27] Alex Quilici and David N. Chin. "DECODE: A Cooperative Environment for Reverse-Engineering Legacy Software." *Second Working Conference on Reverse Engineering*, Linda Wills, Philip Newcomb, and Elliot Chikofsky, editors, pp. 156-165, IEEE Computer Society Press, July 1995.

[28] *Refine User's Guide*. Reasoning Systems Incorporated. Palo Alto, California, 1990.

[29] Spencer Rugaber. "Position Paper Domain Analysis and Reverse Engineering." *Software Engineering Techniques Workshop on Software Reengineering*, Software Engineering Institute, Pittsburgh, Pennsylvania, May 3-5, 1994.

[30] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[31] Eliot Soloway, Jeannine Pinto, Stan Letovsky, David Littman, and Robin Lampert. "Designing Documentation to Compensate for Delocalized Plans." *Communications of the ACM,* 31(11): 1259-1267, November 1988.

[32] W. P. Stevens, G. J. Myers, and L. L. Constantine. "Structured Design." *IBM Systems Journal*, 13(2):115-139, 1974.

[33] *Browsing Source Code*. Sun Microsystems. 1994.

[34] A. Yeh, D. Harris, and H. Reubenstein. "Recovering Abstract Data Types and Object Instances from a Conventional Procedural Language." *Proceedings of the Second Working Conference on Reverse Engineering*, Toronto, Ontario, Canada, pp. 227-236, July 1995.

[35] Zeigler, B.P., *Theory of Modeling and Simulation*, John Wiley and Sons, New York, 1976.