Domain Based Design Documentation and Component Reuse and their Application to a System Evolution Record

Final Report

Table of Contents

Introduction

Background

The Domain Based Design Documentation and Component Reuse and Their Application to a System Evolution Record project (DARE project) is investigating the recovery of design rationale from software artifacts. The derived information is intended to comprise part of the System Evolution Record for ARL's *Rapid Prototyping for System Evolution Record* STO.

Approach

The DARE project makes use of the Synchronized Refinement (SR) reverse engineering method, basing design recovery on a model of a software system's application domain. Domain analysis is used to drive code analysis. The two analyses interact to construct an architectural model of the software system. Additionally, We have tried throughout all three aspects (domain, code, and architecture) to explore the automatic generation of interlinked documentation.

Case Studies

As part of the project, we have engaged in two case studies. The first examined the Mosaic Internet web browser. In this case, the source code is written in the C language, comprises about 100KLOC, and is publicly accessible. The second case study examined the domain of software loader/verifiers (SLVs) for embedded systems. In this case, the software we examined was written in the Ada programming language, comprised about 25KLOC, and was obtained from the Army Tank and Automotive Command (TACOM) where it was part of their SPAIDS effort (Software Program for Affordability Improvements in Depot Systems). In addition to the source code, various documents were available including a Software Requirements Specification, a Software Design Document, Operating Instructions, and two Test Reports.

Domain Analysis

Application-domain modeling constructs a model of a problem's application domain. The nature of the model depends on its intended use. Synchronized Refinement uses the application-domain model as a high-level guide for program understanding. In the case studies, the application-domain model starts as an abstract model derived from existing textual descriptions and evolves toward more specialized models by incorporating detailed information extracted from source code analyses.

Method

For the Mosaic case study, We created an application-domain model for the World Wide Web in three steps. First, various descriptions of the World Wide Web 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). Third, the constructed OMT Object Model is instantiated in a machine-manipulatable form using the Software Refinery tool set.

Written Descriptions

The first step in creating the application-domain model is to extract information from descriptions of the World Wide Web. The best descriptions are specifications, which should be authoritative and fairly complete. Unfortunately, the existing specifications for the World Wide Web are fragmented and concentrate on lower level details such as protocols (e.g. HTTP) and fine-grained semantic definitions (e.g. style sheets). Higher-level specifications, when they exist, are usually given in unhelpful forms, such as binary-only reference implementations (e.g. Arena).

The alternative to formal specifications are informal descriptions; there are many such informal descriptions of the World Wide Web available both on and off the Web. All the considerations that make a description informative also make it a good source for domain analysis: conciseness, completeness, clarity, and so on. In addition, to allow for machine-aided object analysis, it is helpful if the material is available in machine-readable form. For the case study, we chose Chapter 1 from *Introduction to CGI/PERL* (S. Brenner and E. Aoki, M & T Books, 1996).

Lexical Analysis

Lexical analysis consisted of word-frequency analysis is a first pass over the World Wide Web descriptions to determine the important words. After eliminating noise words such as "the", the remaining words are counted; more frequently occurring words are assumed to be more important than less frequently occurring words. To get a feel for the larger structure of the descriptions, we also looked at digram (e.g. "server sends") and trigram counts. Table 1 shows some frequency analysis results.

Table 1 A portion of the word-frequency analysis of *Introduction to CGI/PERL*, Chapter 1.

Word	Frequency
server	61
script	36
browser	24
client	23
user	14
information	12
scripts	11
output	11

Objects and Relations

We then performed a noun and verb analysis on the World Wide Web description to extract possible object classes and associations. The analysis resulted in the 25 object classes, and 6 associations depicted in our Mosaic Object-Model Diagram.

Products

OMT Object-Model Diagram

We have Object Modeling Technique Object-Model Diagrams of the Object Architecture for Mosaic and the Software Loader/Verifier.

Requirements Document

We have an HTMLized version of the Requirements Document.

Software Refinery Code

We have some Refine and Dialect code to assist in our domain analysis.

Tools

We have some filters that do simple lexical analysis on text as a prelude to doing a more complete object analysis.

Keyword Search

One simple way to bridge the gap between code and domain is to examine one side for words used in the other side. For example, by examining the domain, it may be possible to determine that functions containing the word "cache" are used to cache retrieved pages or open socket connections. In the other direction, the requirement that caching be used to improve performance raises the expectation that various parts of the code may be involved with aspects of caching.

We have examined the possibilities from the domain side by constructing a Web page that performs keyword searches on a design document, returning those pages containing matches to the search. The Web page captures the search request and sends it back to a CGI program on the server side. The CGI program applies the search request to a concordance generated from the design document, noting the pages containing matches. The CGI program then generates a Web page in reply to the search request; the reply page contains a thumbnail of each matching page with the matches highlighted in color. Each thumbnail is too small to read individually, but the density and color of the matches easily stands out to indicate relevance. Clicking on a thumbnail retrieves a full-sized version of the associated page.

In addition to indicating the general usefulness of keyword-directed searching for domain concepts, this application also nicely ties together several of the other tools created during this project, including the lexical analysis tools used to generate the concordance and the free-text formatter used to generate the

full-sized versions of matched pages.

Code Analysis

High-level models are useful in the description and comprehension of complex constructs such as programs. Their utility is due primarily to their abstraction of the details of the systems they describe. In contrast, the source code of a program is full of such details. A rough analogy between the high-level models of an application program (in terms of a domain model, design decisions, and a proto-$typical architectural model) and the field of building architecture is the difference between the blueprint of the classical Frank Lloyd Wright Prairie House and a particular instance of it. For example, there may be differences in the instantiation that are derived from pragmatic concerns due to the site peculiarities.

Method

Synchronized Refinement matches a body of source code to an application-domain model. Through code analysis, an abstraction of the code is developed that is resolved with the expectations generated by the models. This is a iterative process reflected by the dataflow cycle in Figure 1 from the Abstraction process to Abstract program description to the Detection process to the Mappings/ annotations and finally back to the Abstraction process. Additionally, the high level model is also enhanced to fit the source code. In reference to the building architecture analogy, SR also constructs a blueprint of a Prairie House that incorporates the peculiarities of this instantiation (i.e., the deviations from the classical).

Products

Tools

software reflexion model (Murphy, Notkin, and Sullivan, "Software Reflexion Models: Bridging the Gap between Source and High-Level Models." ACM SIGSOFT 95 Symposium on the Foundations of Software Engineering, October 1995) compares an architectural model of a program to its actual implementation. It does this by using a map between source code and an architectural model suggested by the analyst. A reflexion model can be used to incrementally refine an architectural model. The process begins with an analyst specifying what the suspected architectural model is for the source code. By incrementally adapting the mappings between major structural elements of the source code (e.g. file names, directories, name fragments), an architectural model is matched to the source code. The differences between the specified and derived models reflect mismatches between the two models. The mappings may be modified to construct a closer convergence between the two.

--> One way to construct an initial architectural model is to analyze procedure calls. **Call-graph analysis** determines which functions call others in a program. For some languages this may be a problem. In C, for example, a function may be called indirectly through a function pointer. In polymorphic languages, determination of the most appropriate version of a function may not be known until runtime. Even with these limitations, call-graph analysis can still be useful in generating abstract program representations.

Analysis of source-code data types may be used to develop an object-oriented representation of the code. This may be done even if the source language doesn’t natively support object-oriented constructs. **Part-of** associations (aggregation data structures) may be derived from analysis of the hierarchical

structure of the types. This may be supplemented by call-graph analysis. For example, a datatype's subcomponent may refer to a collection. If a member function associated with the datatype retrieves elements from the collection, that would indicate 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 functions' formal arguments and return types and associating functions to the abstraction interface of one particular type. Additionally, datatypes serving as collections have to be identified. Implementation of this in the Software Refinery requires interaction with both the low level (parse tree/symbol table) and high level (who-calls, etc.) representations of the program. The Refine language makes both representations manipulable by many of the same operations and transformations.

`sbdump` Filter Tools

We have a program that reads `sbdump` files and produces various kinds of static program analysis on the data read.

Analysis by Relational Modeling

Refine is an excellent tool for representing program text as abstract syntax trees, and provides many powerful, but low-level, mechanisms for manipulating abstract syntax trees. However, the low level with which one engages the abstract syntax tree, coupled with the number of details involved with using Refine itself, means that analyses tend to be rather heavy-weight in terms of the effort required to produce them.

We felt it desirable to be able to perform tentative, speculative program analysis with less overhead than would be required by using Refine, and decided to investigate whether relational database systems (RDBS) could be used to perform the kind of lightweight program analysis we had in mind.

We used the Leap relational database system to prototype our ideas. Leap is a teaching tool for learning about relational databases, but it does offer the usual compliment of relational operators, and we were betting its simplicity would extract a low overhead when implementing our ideas. If we judge our prototype worthy of further consideration, we can move on to a more production-oriented relational database system.

The first step was to extract program information and store it as relational tables. Our program information came in the form of `sbdump` information produced by the Sun SunSoft C compiler. Our relational table design involved a straightforward translation of the tabular information provided by `sbdump`. We wrote several AWK scripts to parse the `sbdump` tables and generate Leap scripts that create the tables.

Some of our analyses requires non-relational operations, for example, regular-expression matching, which meant our analyses had to be specified in something a bit more involved than just Leap scripts. Leap provides only a line-oriented client interface; it has no programming-level interface. We used emacs, which can run Leap as a background process and deal with Leap's output in buffers with the usual emacs text-manipulation facilities. Using these techniques, we were able to, for example, create a analysis of which subroutines access which global variables in the Mosaic World Wide Web browser.

Architectural Analysis

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 three primary issues:

- What is the relation between the application-domain model and the architecture?
- How can architectural styles 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?

Method

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 (or reason-maintenance) system will be needed to handle confirmation and refutation of expectations.

Products

Architecture Document

You can compare the results of our architectural analysis for the Software Loader/Verifier with a data-flow diagram taken from the Software Loader/Verifier design document.

Tools

Part of our architectural analysis is bottom-up, driven by information extracted from code. We have two resources for bottom-up architectural analysis: Refine and `sbdump`-oriented tools.

Call-Graph Generation

By extracting various information from `sbdump` files, we can construct assorted architectural analyses. We have a program to produce call graphs. We also have another call-graph generator that extracts its information from a relational database containing the `sbdump` information.

Automatic Documentation

Automatic Linking of Artifacts

We have a filter that reads free-formatted text and produces text decorated with HTML statements

Conclusions and Future Work

The DARE project has made significant progress in satisfying its research goals, but there are several areas where interesting further work is possible.

Synchronized Refinement

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 will 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. This process is called Synchronized Refinement, and we have had suggest applying it manually. We would like to continue working towards its automation.

Program Visualization

We have performed three kinds of code analysis: 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. o Make use of the third-dimension and allow the analyst to navigate through the resulting visualizations to detect patterns. o Apply some experimental algorithms that propose node-groupings that might suggest modules or architectural components to the analyst. o 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.

Knowledge Sources

The final, and 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 try and encode programming knowledge, in the form of a set of "program plans" and use cliche recognition techniques to find them in the code.

Related Projects

- I-Doc
- The Designer’s Associate

Papers

- Program Comprehension Workshop paper
- Journal Version

This page last modified on 29 October 1997.