DSB: The Next Generation Tool for Software Engineers

Kavoori Kiran Ram

Collaborative Software Development Laboratory
Department of Information and Computer Sciences
University of Hawaii
Honolulu, HI 96822
(808) 956-6920
kavoori@uhics.ics.hawaii.edu

CSDL-TR-93-05

Last Revised: May 4, 1993

Contents

1 Introduction

1.1 Overview

During the development of software projects, there always exists the problem of design specification maintenance. As the project team surges ahead with the development process, there is a strong need to maintain an up-to-date documentation of the current system. This requires an additional effort on each team member to maintain a consistent report of the modifications and additions they make on the system.

This Designbase project attempts to reduce the overhead involved in the maintenance of ever changing design specifications, by generating automatically, a design documentation from the source code and the overview files that are maintained along with the system.

1.2 Motivation

The Designbase (DSB) system satisfies two primary goals: First, it provides automated extraction of design-level documentation about the public interface to any Emacs-Lisp system under development or use within CSDL. Automated extraction of this information significantly decreases the overhead to developers of publicizing how to use or modify a recently developed system. Rather than requiring developers to create and maintain a parallel design or user-level description of their systems, the Designbase simply extracts it from the source code. Moreover, eliminating this redundant description enormously decreases the overhead of maintaining the documentation as the underlying system evolves. The Designbase also has a simple critique section, where a series of checks are run on all the public functions to point out omission of certain standard programming conventions used in software development. In an exploratory yet collaborative development environment, automated design documentation generation and critique will facilitates shared use of evolving systems.

Of course, the Designbase can't reverse engineer this information from just any old source code. Rather, the Designbase can only fulfill its purpose effectively on code that has been designed and implemented according to CSDL object oriented Emacs Lisp standards. This leads to the second major purpose of the Designbase: to enforce and facilitate the use of object oriented design techniques within the group by providing a significant incentive to those who use them.

2 Approach

2.1 Problem Definition

No software system is permanent or immune from change. Often the need to change the software begins as soon as it is accepted as operational. This occurs for a number of reasons including requirements and design omissions, or a better understanding by the user as to what is needed vs what is produced. Regardless of how well the software is designed, the need to make changes (enhancements, extensions and corrections) will arise. Therefore, it is essential to anticipate future use and maintainability of the software during the requirements and design phase.[Wilma, 1984]

So there should a software framework to address the maintainability of software during it's life cycle. Such a framework would take into consideration some important requirements

as:

- the ease of understanding the software. A significant portion of the cost of software maintenance can be attributed to an inadequate understanding of the software, its intended purpose, and how best to make the changes. Normally, software is maintained by someone other than the developer. The work of the maintainer is drastically reduced if the developer provides an accurate documentation for the code he is developing.

- the ease of making software changes. Maintainability is improved if the system is designed in an Object-Oriented style with hierarchical, modular units that perform one principal function with minimal interaction between the modules.

2.2 Proposed Solution

This project focuses attention on "the ease of understanding the software" through proper documentation. Software projects are outcome of combined efforts of programmers, software engineers and managers. Coordinating individual member of the team and the final integration of work plays an important role in producing good software systems. Thus the work done by each member of the team should be easy to understand, reuse and modify if required, not only by other members within the group, but also by people reusing code for further extensions. In order to attain this higher level of understandability of code, development teams adopt certain norms and conventions to be followed by each member of the team. These norms concern to issues in programming techniques, like standards in naming conventions, documentation for each function, public and private functions and so on.

Given such standardized conventions, an automatic system can be developed to parse the source code files, and extract a design level description of the system that encompasses the module level structure only, and abstracts away the underlying implementation. As an additional feature, the designbase can check for the deviations of these standard conventions and can report to the developer, who then can make the required corrections, before a public release. Such a system would not only be able to improve the understandability of the software system simultaneously reducing the burden on the developer, but also help him/her in defining his/her functions precisely thereby reducing the error percentage from the system.

3 Current Status

3.1 Operational Status

The Designbase application has been conceived and implemented about four months back and has been in operation since then. This is a masters level project started in Feb '93. Over this period of four months, dsb has been fine tuned to incorporate a lot of functionality and bug fixtures. From the initial version of parsing source files to generate a LaTeX file of the public interface of an application, dsb has been extended to generate LaTeX report/article style formats. Dsb has been structured using Object-Oriented concepts making it an easily extendible system.

Dsb has been in regular use in the CSDL research environment. The following statistics of each application within CSDL illustrates the extent of usage of dsb.

- **EGRET**
  2.0MB code, 4 subsystem modules, 32 classes and 300 public operations. [Johnson, 1993]
- **CLARE**
  1.7MB code, 4 subsystem modules, 33 classes and 174 public operations. [Dadong, 1993]
- **CSRS**
  1.0MB code, 2 subsystem modules, 18 classes and 100 public operations. [Tjahjono, 1993]
- **URN**
  0.14MB code, 3 subsytems, 5 classes and 31 public operations. [Brewer, 1993]
- **DSB**
  0.5MB code, 2 subsystem modules, 9 classes and 48 public operations. [Kavoori, 1993]

3.2 Object Orientation Conventions

The applications in CSDL environment are designed in Object-Oriented fashion. The major entities in Object Oriented design are classes, objects, attributes, operations, and collections. Attributes, operations and collections can be private or public. Public nature is indicated notationally by the use of * as the visibility token; ! as the visibility token indicates a private. If an attribute is private, then functions outside the implementation of the class should not access or set the attribute's value. This, of course, is not enforced in Emacs Lisp, but is rather a way of helping programmers to understand what parts of a subsystem are internal details and can thus be safely ignored. Designing and programming in an object oriented fashion will greatly ease understanding of the system architecture, as well as supporting migration to other languages providing direct support for these concepts.

3.2.1 Classes

A *class* is a collection of objects with related structure and behavior. Each class has associated with it a set of superclasses, a set of subclasses, a set of attributes, and a set of operations.

Classes frequently have constructor and destructor operations for the objects associated with them. These operations are always called make and delete. Classes are, in some sense, purely a design-level representation. No explicit Emacs Lisp support for defining classes exists.

3.2.2 Objects

Classes are an organizational entity–what actually exists during execution are particular instances of classes, or *objects*. Each object has a unique identifier associated with it, and this unique identifier (or the object itself) is virtually always the first argument to the operations associated with a class.

3.2.3 Attributes

Each class instance has a set of characteristics, or *attributes* associated with it. For example, each screen has a geometry, color, name, and so forth. These are attributes of the class. Some attributes are *set-able*. This means that an operation corresponding to the attribute is defined to change its value. This operation is named by prefixing the attribute name with **set-**.

3.2.4 Operations

While attributes refer to characteristics of classes, *operations* refer to the behaviors of the instances of the class. Operations may take any number of arguments, but the first argument is, by convention, an object representing the class instance. Operations must always document what form their first (as well as the other) argument must take. The operation may take additional required or optional arguments depending upon the nature of the behavior.

3.2.5 Collections

Operations, as defined above, operate on individual instances of the class. There is another kind of operation, however, that operates on the set of all objects that exist in the class. These are called *collection operations*. For example, an operation that returns a list of all nodes of a specified type is a collection operation. These operations, taken together, represent many of the classical database query and retrieval operations supported in Egret .

Collection operations are always defined relative to a class, and are named by wrapping { and } around the class name.

3.2.6 Administrator

These are special operations that are defined for a few classes. They typically provide system initializing operations and recovery operations. These administrator functions are used by the system administrator while bring up the application and in times of system crash.

3.3 Notational Conventions

3.3.1 Identifier Syntax

Function and variable names in CSDL have a standard format.[Johnson, 1993] Each name must adhere to the following template:

```
<sys-name><sys-vis><class-name><class-vis><name>
```

where:

- `<sys-name>` is a set of characters that identifies the subsystem membership of the object.
- `<sys-vis>` and `<class-vis>` are single characters indicating the external visibility of the object. The character * indicates that the object is public, the character ! indicates that the object is private, and the character @ indicates that the object is a system

administration function to be manipulated only by distinguished users at special times (such as database initialization, recovery, and so forth).

- `<class-name>` is typically the full class name, but may sometimes be an abbreviation.
- `<name>` is the actual name of the operation or attribute.

For example, the function **i*cmd*find-node** is the interface subsystem public operation find-node from the class command, which is abbreviated as **cmd**. By prefixing operations with their class name, the operations class membership is automatically documented, as well as allowing the same operation name to be defined for different classes (for example, **node*find-node**.)

3.3.2 Common Name Components

Beyond naming syntax, Egret also has conventions for naming certain kinds of semantically related operations. These conventions are

- Constructor and instantiation operations are prefixed by **make-**.
- Deletion operations are prefixed by **delete-**.
- Operations that set the value of an attribute are prefixed by **set-**.
- Recovery operation names are by convention prefixed with **reset-**, and whenever possible suffixed with the name of the attribute or operation whose functioning they repair.

3.4 Parsing Process

The parsing process of dsb relies heavily on the conventions followed in the CSDL environment. DSB is designed to carry out the following procedure:

- Read in each lisp source file.
- Process each lisp form in each file selecting only public functions and macros. Macros are expanded and then processed. Generate an entity for each operation. Classify them into attributes, operations, collections, variables, and administrator functions.
- Sort the entities into operations under each class, and then the classes under each subsystem. Critique each operation and class for dis-conforming standards.
- Generate table of operations for each class and for each subsystem. Format the output in an appealing way to the user, presenting the public interface of the system.

3.5 Software Quality Assurance Process

The Designbase has a set of critique instances that can be applied on each class. The list of current critique instances are

- **Missing Constructor**
  Every class that is defined should have an operation that creates an object of that class, typically called the constructor function. Absence of this operation shows a serious flaw in the design.

- **Missing attributes**
  Each class has an associated set of attributes, which need to be documented in an appropriate manner, understandable by dsb. The lack of attributes for a class would indicate that some operations might have been wrongly classified or the system design does not provide much functionality to that class. This would draw attention of the designer to think better ways of providing functionality.

- **Missing Operations**
  A class exists when it has a defined functionality which is exhibited by the set of operations available for that class. Absence of at least a single operation for a class indicates that the existence of the class is not well justified.

- **Missing Documentation**
  Undocumented functions are the worst nightmare any software engineer trying to decode and reuse existing systems. Missing documentation needs immediate attention.

- **Missing Overview files**
  Overview files for each class and subsystem provide necessary information to understand the modules from the developer's perspective.

- **Undocumented Parameters**
  When ever a function gets reused, it is obvious that it should take the same type of arguments every time. Unless each argument is documented properly, it would be difficult to use the function.

- **Undocumented Return Value**
  It is important to know what to expect as a return value when a function is called, so that the calling function can handle the return value properly.

The critique operations returns a list of failed instances which form a part of the documentation. These glaring shortcomings in the system implementation would initiate appropriate action from the developer. Elimination of these defects would make the system more easy to understand by the other members of the group and by the people involved in maintaining and extending the system.

4 Future Directions

4.1 Short comings of DSB

- The designbase parses only Lisp code.

- Emacs Lisp does not have Object-Oriented constructs and the concepts of Object Orientation is incorporated only by convention followed in the CSDL environment.

- Information that is extracted from the source code is pretty much existing within the source files and not much information is derivable from the designbase hardcopy.

4.2 Future additions

- The parsing technique can be modified to recognize constructs of different Object-Oriented languages like C++ and Common Lisp. Once the basic dsb entities are created, then it can do similar processing.
- A higher level of information can be derived from the current dsb system by means of different representations. For example, an Egret system can be used to represent dsb entities in the form of nodes and links. This network can be manipulated from a higher level to record changes made to a system over a time period, in a sense the evolution of a software system itself. In Software Engineering circles, this kind of representation has a great potential in the research of software life cycles.

References

[Wilma, 1984] Wilma M. Osborne. A Framework for improving Software maintenance throughtout the Software Lifecycle IEEE 1984 3rd Software Engineering Standards Application Workshop.

[Johnson, 1993] Philip Johnson. EGRET Requirements Specification Also published as Technical Report CSDL-93-02, University of Hawaii Department of Information and Computer Sciences.

[Dadong, 1993] Dadong Wan. CLARE-1.2d Design Document CSDL, University of Hawaii, Department of Information and Computer Sciences, May 1993.

[Tjahjono, 1993] Danu Tjahjono. CSRS-1.2.1 Design Document CSDL, University of Hawaii, Department of Information and Computer Sciences, May 1993.

[Brewer, 1993] Robert Brewer. URN-1.0.2 Design Document CSDL, University of Hawaii, Department of Information and Computer Sciences, May 1993.

[Kavoori, 1993] Kiran Kavoori. Designbase-1.3.5 Design Document CSDL, University of Hawaii, Department of Information and Computer Sciences, May 1993.