A Requirements Specification for a Software Design Support System

Robert E. Noonan

Department of Computer Science
College of William and Mary
Williamsburg, VA 23185
804-253-4748

Final Report

NASA Langley Research Center
Grant NAG 1-647

January 8, 1988

Abstract

Most existing software design systems support the use of only a single design methodology. Our major thesis is that a good SDSS should support a wide variety of design methods and languages including structured design, object-oriented design, finite state machines, etc. It might seem that a multiparadigm SDSS would be expensive in both time and money to construct. However, we propose instead an extensible SDSS that directly implements only minimal database and graphical facilities be constructed. In particular, it should **not** directly implement any specific design methodology or set of methodologies. Instead, the SDSS should implement tools to facilitate language definition and analysis. We believe such a system could be rapidly developed and put into limited production use, with the experience gained used to refine and evolve the system over time.

1 Introduction

In a survey of minimum, recommended, standard software toolsets, Glass [1982] devotes only a single category out of 38 to software design. Of the 16 standard environments surveyed, only four provided any support for software design. All of these relied on tools for recording a design written in a Program Design Language or PDL. Often, PDL's are derived from the programming language of choice. Hence, design using a PDL has often been criticized as not being high level design at all, but really detailed design or high level coding. In short, it is merely a vehicle for recording design decisions already made, and as such, is of little value to the designer. It is also a poor presentation vehicle.

At the current time there is a great deal of research being conducted on the software development environment. Unfortunately, most of the tools, such as syntax directed editors, pretty printers, etc., are aimed at the coding phase, rather than the software design phase.

Over the last ten years, a number of methodologies have emerged to aid the software designer (or architect) in constructing a coherent design. However, despite the adoption of structured design and coding methods, the process of developing software today continues to be unsatisfactory. Furthermore, many software design methodologies have little or no automated support. Even those that do are often characterized as being more concerned with producing design documents than supporting the design process. Such criticisms have been leveled at systems as diverse as PSL/PSA, HIPO, and TAGS.

A number of once-promising software design systems appear to have fallen into disuse; a good example of such a system is HIPO. An analysis of the problems of such systems is instructive. First, they often require the designer to use an unnatural and complex recording mechanism, in order to enhance the presentation. These systems are also often weak on analysis tools. Thus, most designers view such systems more as an obstacle than as an aid to the design process.

More recently, a number of schematic design products have emerged, which run on either personal computers or workstations. Typical of these products are Case 2000 Designaid from Nastec Corp., Teamwork from Cadre Technologies, Inc., and Excelerator from Index Technology Corp. These systems currently provide a very effective schematic diagram capability using the bit-mapped graphics capability of the underlying workstation. Most of these systems also provide for a design dictionary. Currently, their major weakness is in the design analysis functions provided, although this is expected to improve over time.

Users of these products have reported rather dramatic productivity improvements. More importantly, the graphical presentation of the design has brought the users of the system, the designer, and the programmers closer together. All three are better able to visualize both the logic and data flows through the system. Any of the three can more readily identify problems while the system is still in the design phase. Users can identify whether or not the system corresponds to their model of the world. Similarly, if programmers see implementation problems, they may be able to suggest design changes

which avoid these problems.

There is an attempt to merge these systems with backend code generators, usually producing Cobol. This again leads to enoromous productivity gains. However, most of the backend code generators are used for these major purposes: screen generation, input data validation, and database access. These activities consitute a major portion of a standard commercial application, often amounting to thousands of lines of code. The processing elements are by far, considerably smaller in size.

To summarize the experience gained with various systems, a good software design system (SDSS) must provide support for the following major elements:

1. Specification of the design. This is essentially the mechanism with which the designer records his design. Graphical input of the design, where appropriate, is essential.

2. Analysis of the design. The system should provide tools for analyzing the design for such things as completeness, consistency, etc. This aspect is the most useful to the software designer and is the key concern.

3. Presentation of the design. The system should provide tools for presenting the design. This aspect is most useful to the implementor and to the maintenance programmer and is also sometimes used as a vehicle of communication between the designer and the customer. The presentation of a software design should be graphical rather than textual. The old maxim about "a picture being worth a 1000 words" holds for software design.

4. Implementation of the design. The system should provide tools for generating at least a prototype implementation.

2 A Minimal SDSS

Most of the software design systems implemented using personal computers are fairly good, graphics-driven products. They have fairly natural, easy-to-use interfaces and let the designer concentrate on the design rather than using the product. Their major limitation is that they support only a single design methodology, usually some variant of structured design [Yourdon, 1975]. For commercial applications, in which data flow is the major problem to be solved in the design, these systems are more than adequate. Indeed, for commercial users the direct support of a single design methodology is cited as a benefit of these systems.

However, it is a very real question whether data flow design methods are adequate for the types of systems which NASA builds, such as space shuttle, space station, the deep space network, etc. We believe that NASA would be much better off using an object-oriented design method [Parnas, 1972; Parnas, 1979]; for example, one of the reasons cited

for choosing Ada as the language for space station was its support of an object oriented programming style.

More importantly, the best design technique to use varies with both the problem to be solved and with the experience of the designer. A good SDSS should freely allow a software designer to use whatever methodology he/she chooses. This has the distinct advantage that it allows the designer to use different methodologies in different portions of the same system; that is, the designer can choose to use one methodology in one portion of the system, and a different one in a different portion of the system. This allows the problem being solved and the designer's experience to determine the design method used, rather than constrainingg the software designer to using only the facilities provided by the SDSS.

It might seem that a multiparadigm SDSS would be expensive in both time and money to construct. However, we propose instead an extensible SDSS that directly implements only minimal database and graphical facilities. In particular, it should **not** directly implement any specific design methodology or set of methodologies. Instead, the SDSS should implement facilities to simplify language definition and analysis.

We believe that NASA should build a prototype design system along the lines suggested here, and then, based on its experience with the prototype, contract to have a SDSS built to its specifications. We propose to show that such an approach need not be either prohibitively expensive nor time-consuming.

In the sections which follow, we will outline the user interface, implementation strategy, methodolgy implementation tools, and database requirements for our minimal SDSS. For the sake of specificity, we will indicate how various aspects of structured design [Yourdon, 1975] might implemented.

3 User Interface

Our major thesis is that a good SDSS should support a wide variety of design methods and languages including:

- structured design,
- data structure design methods,
- object-oriented design,
- finite state machines,
- Petri nets,
- decision tables,
- program design languages (PDLs), etc.

Thus, a good SDSS must support both textual and graphical languages and combinations thereof. In order to be able to use a variety of these methodologies within the same design, we will need both textual and graphical editors and displays.

Because of the size of the systems NASA needs to design, the SDSS should be hosted on a high performance, bit mapped, scientific workstation with a large virtual memory running the Unix operating system. Appropriate workstations include the Apollo Domain series, Sun 3's, and Microvaxes. However, a prototype system could easily be developed on a personal computer, such as an IBM PC/AT compatible or Macintosh.

The interface to a textual subsystem is straightforward and needs no discussion. The interface to a graphical subsystem would be a conventional graphics interface much like you would find in either Designaid or Excellerator. A design would be constructed using a pallet, pop up (pull down) menus, and a mouse. Like current PC based design tools, the user interface should be fairly natural and easy-to-use, letting the designer concentrate on the design rather than using the product. The system needs to support windows which allow us to browse and backup arbitrarily. Furthermore, the use of windows allows a designer to call up contextual help whenever it is needed. All of this should be provided in an integrated, seamless environment.

We believe that a good SDSS should allows us to explore things abritrarily, without knowing where or how an object is defined. In any given methodolgy a process or procedure can be defined using any other methodology. Thus, a process bubble in a dataflow diagram might be implemented as a child dataflow diagram, or as code in a program description language (PDL), or even as a Petri net. Thus, any object in the system must support abritrary links to any other object, much in the sense of hypertext [Nelson, 1988].

Other basic facilities include version control. A software designer should be able to modify an existing design, and then at some later time be able to recapture the original, unmodified design.

From a software designer's viewpoint, the heart of a good SDSS is the analysis it provides. If the system is weak on analysis tools, an SDSS is not much better than a good word processor.

Unfortunately, the amount and type of analysis that can be done varies widely with the design methodology used. Minimally, the SDSS should provide consistency and completeness checks, that is, ensure that everything in the system is defined exactly once and used at least once. In the case of some methodologies, an object may be used only once; for example, in structured design, processes and dataflows may be used only once.

While a software design is under development, one of the most important functions of the SDSS is to keep track of those objects which have been used but not yet defined. A second, related function is that of browsing through the design so that the designer can see the context or contexts in which an undefined object is used.

Other highly specific analysis tools are possible, depending on the design methodology used. Again, consider the implementation of structured design. One important consistency check is that for each process the set of the outgoing elementary data items does not exceed

the set of incoming elementary data items, i. e., information is not magically created. Another important check is that the dataflows of a child dataflow diagram exactly match those of the parent process (except for trivial error dataflows).

Ideally, a software design itself should serve as a prototype implementation of the desired system. For an appropriate software design methodology, it should be possible to easily describe an interpretor for that methodology within the system. The SDSS should then support execution of the design on live data, including appropriate tracing, breakpoint, and other debugging facilities.

In addition, a number of other implementation facilities are possible and should be able to be easily provided. For example, given a Pascal PDL procedure it is straightforward to generate corresponding Pascal skeletal code. Given a data description you can generate appropriate record descriptions in the target language with appropriate references to the design document. The SDSS must support facilites for describing these implementations.

4 Implementation Strategy

It is clear that the proposed SDSS has a very object oriented flavor to it. Furthermore, the language chosen should support windows, menus, and graphics. Thus, an ideal candidate language would appear to be a language like Smalltalk [Goldberg, 1981].

The major element of a language like Smalltalk is the class or type. All objects are of some class or type, including classes themselves; furthermore, any new class or type automatically inherits all the procedures or methods of its defining class. This inheritance makes Smalltalk a very powerful language for building a system like we propose. The other major aspect of writing Smalltalk programs is polymorphism; it is a simple matter to build procedures or methods which work for a variety of different types.

The concept of classes and inheritance greatly simplifies the building of the basic facilities of the design system. Take the concept of an object such as a diagram or bubble. Many different objects with different properties are simply different kinds of bubbles, e. g., dataflow diagrams, processes in a dataflow diagram, states in a finite state machine, places in a Petri net, etc. Yet, these things also share many common properties: they must be uniquely named, they all have similar graphical representations, etc. Automatic inheritance of procedures for new classes in Smalltalk greatly simplifies the construction of such a system.

Classes specific to software design and thus, built into the SDSS would include bubbles, databases, and connections. Items specific to a particular methodolgy, such as processes in structured design, would be subclasses of one of these, e. g., a process is a kind of bubble.

For a given design methodology the user interface would be constructed via the extension language, including the pallet, the menus, and the objects represented by the icons. Thus, the major purpose of the SDSS system itself are as follows:

- To provide the basic textual and graphical facilites, such as text editors, graphical

editors, the interface to the database system, etc.

- To provide an implementation of the extension language and the tools required to define a methodology.
- To provide the basic mechanisms used by the tools above.

The subject of the tools required to implement a methodology is discussed in the next section.

5 SDSS Tools

Our proposed SDSS must be user extensible via a good extension language. Although in an entirely different domain, a good model for such a system is the Emacs text editor. Emacs is totally user extensible, with the extension language used being a variant of Lisp. Such an approach here would provide a great deal of flexibility. Software designers could customize the system to their own needs. They could also more readily adapt the system to new and evolving design methodologies. As with Emacs, even the basic design methodologies supported would be accomplished via the extension language mechanism.

The ideal extension language for this application is not entirely clear. One obvious candidate is, of course, Lisp. A second possiblity is Smalltalk [Goldberg, 1983]. Others are possible.

For our purposes a design methodology consists of the following:

- A language, its syntax, semantics, and interpretation, particularly, its mapping to classes and objects.
- The visual display of language objects.
- The representation of language objects in the database.
- The procedures used to analyze designs written using the language.

Thus, the extension language is primarily a set of tools for implementing a language. We will discuss the first two items above in this section, and the other two in the next section.

We envision the process of implementing a design methodology as follows:

1. Give an LL(1) grammar for the language, from which an abstract syntax would be automatically generated [Noonan, 1985].
2. Specify the semantics, namely, the mapping from abstract syntax to classes and objects. Type checking would also be specified and might involve the construction of menus and prompts, as needed.

3. Specify the mapping from an object of a particular class to its representation in the database.

4. Specify the analysis algorithms.

While this sounds like the burden of defining a software methodology is being placed on the software designer, this is not the case. Most of the existing design methodologies would be already specified via this mechanism. Second, most of the items above are basically nonprocedural specifications in nature; they are simple to write and to debug. The complex tools needed to support these, e. g., an LL(1) parser generator, would be supplied with the system.

For a given design methodology the user interface would be constructed via the extension language, including the pallet, the menus, and the objects represented by the icons. Thus, the entire design interface could be easily modified by a software designer. Since objects would just correspond to classes in Smalltalk, a designer could easily introduce new objects or redefine existing system objects.

The menus would correspond to Smalltalktalk methods, i. e., operations on objects. By means of these, a designer could, for example, connect two processes via a dataflow (in the usual structured design methodology). In the underlying Smalltalk world this would consist of the following:

process1 dataflow: flow to: process2

The diagram resulting from these messages would then be displayed on the screen.

To see this in more detail, let us consider the implementation of the structured design methodology in more detail.

5.1 Implementation of Structured Design

Consider how structured design [Yourdon, 1975] could be implemented via the extension language in our minimal SDSS. Structured design uses processes, dataflows, etc., to produce a dataflow diagram of the entire system. Because of its size, this diagram is usually hierarchically decomposed into smaller subdiagrams. In any diagram a process can be either a primitive process or can represent a subdiagram. Primitive processes are usually defined in a program design language (PDL).

The major objects (and their superclass) to be manipulated in structured design are as follows:

- Processes or methods (class bubble).
- Subdiagram (class bubble).
- Data flows (class connection).
- Data stores (class database).

- Sources (class database).
- Sinks (class database).
- Data definitions (class database).
- PDL (class bubble).

Creating an object of one of the above classes necessitates filling in a bunch of information about the object, what Smalltalk calls the instances variables associated with the object. The specific holes vary according to the class or type of the object. A process would have: a name, a description, and an implementation (either a PDL or subdiagram) associated with it. A dataflow would have: a name, a description, a *from* process, and a *to* process associated with it.; etc. Having created a dataflow between two processes, the software designer would be prompted for its name and its description. Alternatively, a desginer might wish to be only prompted for the name of a dataflow and not for its description. Similarly, the designer could create dataflow diagrams by concentrating on the dataflows (which must be named) but leaving the intervening processes unnamed until a later time.

5.2 Implementation of a PDL

The implementation of a PDL is instructive because it is very unlike a dataflow diagram. A PDL is a textual object rahter than a graphical object. Also PDLs come in a number of distinct styles depending on one's favorite programming language, i. e., there are Pascal PDLs, Ada PDLs, etc.

For each PDL one must specify both its syntax and semantics. These would be accomplished via an LL(1) grammar with embedded sematic actions [Aho, 1977]. The latter are used mainly for two puproses. One is to control links or relationships with other semantic objects in the design. This would allow a PDL to be pretty much input as raw text and then parsed into a form suitable for storing in the database. Once an object has been classified as some form of object other than text, the method of defining it would automatically be invoked, involing all its menus, prompts, defaults, etc.

The other alternative is to use a syntax-directed editor [Teitelbaum, 1981]. However, experience with a variety of such tools appears to indicate that the input of raw text is preferred by users. This requires the system to parse the text to store a PDL procedure in the software design database and unparse to display it to the user. The method proposed above does not rule out the development of syntax-directed editors.

6 The Data Dictionary

The other key element of a good SDSS is the data dictionary, in which is described the contents of all objects in the system. For the data dictionary, it seems best to use an

ordinary, commercial, relational database system. One requirement for such a system is that it support the emerging SQL standard.

One of the potential problems of relational database systems at the current time is performance, relative to other database systems and to a system specialized to software design. However, a software specification is miniscule in both size and number of transactions compared to an online banking system, an airline reservation system, etc. Therefore, the performance gained by using a network or hierarchical database system is not worth the power lost. Similarly, the flexibility and power gained by using a commercial database system outweighs any performance gained by implementing a system particular to software design.

The data dictionary is used to record all information about the system being designed, including processes, files, data flows, etc. Each of these is a named entity consisting perhaps of other named entities. Particularly, for data flows we must be able to denote: composition, choice, and iteration. Furthermore, we must be able to annotate any object with free-form comments and links to other objects.

It might seem that a relational database is an inappropriate vehicle for storing the software design. Many of the elements of the design database are grammatical in nature, i. e., they are represented by some form of BNF grammar. Even data elements themselves are often represented by grammar fragments. As an example, consider the definition of a telephone number using the William and Mary phone sytem.

```
telephone-no     =  * legal dialing sequence *
                 =  [ local-extension
                    | 9 + outside-no
                    | 8 + scats-no
                    | 0                          * local-operator * ]

local-extension  =  4 + { digit }_3^3

local-extension  =  [ 4748                       * Computer Science *
                    | 4481                       * Mathematics *
                    | 4477                       * Computer Center *
                    | ... ]
```

However, Browne [1978] has shown that a commercial database system is quite appropriate for this kind of work. In fact, it saves a great deal of implementation effort and allows ad hoc queries against the database. Having done all the work for one database system, it is a simple matter to redo the mappings for a different relational system.

Another advantage of using a comeercial database system is that the DBMS automatically provides facilities for consistency checking, concurrent access by multiple users, automatic backup/restore capabilites, etc.

7 Summary

In this report we have proposed that NASA contract to develop a Software Design Support System (SDSS) that will allow a number of distinct software design methodologies to be used. We have proposed that this be a minimal system with the actual methodologies and software designer interface being constructed via an extension language, much like Emacs in the domain of text editors.

For simplicity, we propose that such a system be constructed in an object oriented language such as Smalltalk. Although a specific software design methodolgy would be programmed in the extension language, a software designer need not be aware of this; the system he/she would see would be much like current CASE products with bit-mapped displays, pallets, menus, and mice. Software designs would be stored using a commercial, relational database. The analyses provided for each software design would be programmed in the macro language provided as the ordinary user interface to the database system.

We believe such a system could be rapidly developed and put into limited production use. The experience gained with the system could be used to refine NASA requirements and evolve the system over time. Even if developed on a personal computer with its limited processing power and memory, such a system could easily be ported to a scientific workstation running Unix, such as an Apollo Domain 3000 or Sun 3. These latter should provide ample processing power and virtual memory even for a very large design.

8 References

1. Aho, Alfred V., and Ullman, Jeffrey D. *Principles of Compiler Design.* Addison-Wesley, 1977.

2. Browne, J. C., and Johnson, David B. FAST: a second generation program analysis system. *Proceedings of 3rd Intl. Conf. on Software Engineering*, (May 1978), 142-148.

3. Goldberg, Adele, and Robson, David. *Smalltalk-80: The Language and Its implementation.* Addison-Wesley, 1983.

4. Nelson, Theodor H. Managing immense storage. *Byte*, 13 (January 1988), 225-242.

5. Noonan, Robert E. An algorithm for generating abstract syntax trees. *Computer Languages*, 10 (1985), 225-236.

6. Parnas, David L. On the criteria to be used in decomposing systems into modules. *CACM*, 15 (December 1972), 1053-1058.

7. Parnas, David L. Designing software for ease of extension and contraction. *IEEE Trans. Soft. Engr.*, SE-5 (March 1979), 128-137.

8. Teitelbaum, T., and Reps, Thomas. The Cornell program synthesizer: a syntax-directed programming environment. *CACM*, 24 (September 1981), 563-573.

9. Yourdon, Edward, and Constantine, Larry L. *Structured Design.* Yourdon Press, 1975.