SSDE:

STRUCTURED SOFTWARE

DEVELOPMENT

ENVIRONMENT

Michael John Norman

September 1989

SSDE:

STRUCTURED SOFTWARE

DEVELOPMENT

ENVIRONMENT

A thesis submitted to the Department of Computer Science, University of Cape Town, in fulfillment of the requirements for the degree Master of Science.

Michael John Norman

September 1989

ACKNOWLEDGEMENTS

I wish to thank Mrs. S. Berman, my supervisor, for her help in the form of constructive criticism and useful suggestions.

My gratitude towards my family for their patience and moral support.

Finally, a word of thanks for my parents for their interest in my academic activities.

Except as noted above, this work is entirely my own and all references are adequately cited.

Michael John Norman September 1989

ABSTRACT

Software engineers have identified many problem areas regarding the development of software. There is a need for improving system and program quality at design level, ensuring that design costs remain within the budget, and increasing the productivity of designers. Structured Software Development Environment (SSDE) provides the system designer with an interactive menu-driven environment, and a framework within which he can conveniently express and manipulate his proposed solution. This representation is in terms of both a conceptual model and a detailed software logic definition. Thus SSDE provides tools for both high-level (or logical) and low-level (or physical) design. It allows a user to follow his own preferred methodology rather than restricting him to one specific strategy. SSDE builds and maintains databases that record all design decisions. It provides the system designer with a mechanism whereby systems can easily be modified and new systems can evolve from similar existing systems. There are several auxiliary facilities as productivity aids. SSDE generates PASCAL code for low-level design constructs, full documentation of both the high- and low-level designs for inclusion in the project file, as well as a skeleton manual. The system was evaluated by a number of independent users. This exercise clearly demonstrated its success as an aid in expressing, understanding, manipulating and solving software development problems.

TABLE OF CONTENTS

CHAPTER 1:

INTRODUCTION

1.1 Preface.

Software authors and specialists are unanimous that the software development process is at present beset with a number of problems. In order to avoid an ongoing or a new software crisis, it is appropriate to investigate new strategies, design aids and methodologies which can individually or collectively address these problems [Gutz et al:1981 p45].

1.2 Motivation.

Many systems designers are still using totally un-automated methods in designing complex software logic. These manual methods are restricting the potential design productivity of the systems engineer.

About two-thirds of the software maintenance cost can be attributed to misconception, i.e not identifying the user's real system requirements, or improper conceptual design [Ramamoorthy et al :1984 p191-209] [Manna :1974]. The relative costs of hardware components are decreasing (from 90% in the 1950's to 10% in the 1990's), whereas the relative cost of software, when compared with hardware, is increasing at a similar rate [Wasserman & Gutz: 1982 p196-206] [Schindler :1981]. These figures confirm that the manner in which software

logic is produced, needs urgent attention. It is against this background that the Structured Software Development Environment (SSDE) project was undertaken in an attempt to facilitate fast, reliable system design. This aimed at providing an integrated set of system design tools for the personal computer which would be easy to use and inexpensive.

Thus the power of the computer itself will be used to assist in the construction of software destined to run on a computer.

1.3 An Automated Design Environment.

1.3.1 Structured Software Development Environment (SSDE).

Structured Software Development Environment (SSDE) provides the system or program designer with a number of automated tools which allow him to express his software solutions in an orderly and organized manner.

The tools provided within the environment are all integrated so that the designer can move from one facility to the next with ease.

1.3.2 High-Level and Low-Level Design.

Within the environment, there are tools which allow for a high-level functional design to be performed as well as a low-level or detail design activity. These are provided by means of a hybrid HIPO-Structure Chart, and Nassi-Shneiderman

diagramming facilities respectively. There are also tools which support the modification of these designs with ease and speed as the designer re-thinks and expands his design.

1.3.3 Software Engineering Databases.

A central repository, consisting of a number of software engineering databases, is maintained which captures all the data associated with a particular design. Previously designed systems can quickly be retrieved, modified and re-used. The databases can be independently queried to determine the systems being developed and/or the details of specific designs.

A hard-copy document can be automatically produced, giving the complete high-level and low-level design information. The documents produced in this manner can be incorporated into the project file affiliated with this particular system.
Using the high-level conceptual design as a basis, a skeleton manual is written to an ASCII file by SSDE.

1.3.4 Validation Mechanisms.

Numerous validation mechanisms exist throughout the environment to ensure, wherever possible, that the designer is proceeding with his design in a meaningful manner. For example within the low-level design constructs, the text entered is scanned to guarantee that appropriate text is being entered for such a construct.

1.3.5 Software Re-Use.

Since a modularized design activity is supported, it is possible to design a common activity only once and subsequently to re-use this particular logic elsewhere in the same or other systems.

1.3.6 Automated Code Generation.

Enough precise information is captured about the low-level design in order that code can be automatically generated by SSDE. Pascal programs are produced.

1.3.7 Methodology.

At present there is no standard method which can be employed universally to perform a successful system development process. Accordingly SSDE does not force the designer to use a particular methodology. Since the environment is integrated, the analyst can choose to follow his particular methodology or a combination of methodologies.

1.4 Thesis Outline.

The thesis consists of the following chapters:

Chapter 2 gives an Overview of Structured Development. Here the classical approach to systems design with its shortcomings are discussed together with a historical perspective. Present objectives and advantages of structured program development are investigated including a discussion of some popular diagramming techniques used in system and program design.

Chapter 3 surveys Software Development Environments and Automated Tools. Definitions and examples of the latest software engineering (SE) techniques, such as CASE technology, are mentioned and possible future developments in SE are also deliberated.

Chapter 4 provides the Motivation for this research and Chapter 5 introduces the main facilities provided by SSDE.

The issues which caused SSDE to evolve and problems encountered during the development of SSDE are discussed in chapter 6.

Structured Software Development Environment (SSDE) is presented from chapter 7 through to chapter 11. The facilities provided by SSDE and how they have been programmed are discussed.

Chapter 12 evaluates SSDE. The contribution which this environment makes towards the software development life-cycle as well as other advantages provided by SSDE are set forth. The results of a user survey are reported.

Design Methodologies are presented in chapter 13. This gives a general introduction and discussion of different design methodologies and how these can be incorporated within SSDE.

In the conclusion the project is evaluated in terms of the initial objectives and possible future work for SSDE outlined.

Appendixes: A complete user manual is given which can explain the use of the automated facility as well as examples of the environment in use. Examples of actual designs completed using SSDE are given as well as an appendix reporting the results obtained from a user survey. A glossary of useful terms is also provided, and a list of existing software engineering tools.

CHAPTER 2:

STRUCTURED DEVELOPMENT :

AN OVERVIEW

2.1 Introduction.

Since the 1968 NATO conferences [Naur et al :1969 and Buxton et al :1969] on software engineering, computer scientists have been discussing and refining an approach to program and system design known as structured programming or (structured design). This concept promised great improvements over the then ad hoc methods in use. This chapter looks at the reasons which prompted structured design techniques to evolve, their aims and advantages, and some typical diagramming techniques associated with them.

2.2 The Classical Design Approach.

"If we consider software to be an information subsystem, clearly the classical approach has failed" [Aktas: 1987 p22]. The Classical Approach is an algorithm or procedure which may be followed in order to produce a computer solution for some stated problem.

Briefly the procedure involves the following steps:

2.2.1 Planning

- request for a system study
- initial investigation
- feasibility study

2.2.2 Analysis

- redefine the problem
- understand the existing system
- determine user requirements and constraints on a new system
- logical model of the recommended solution (conceptual, logical, or architectural design) or functional specifications

2.2.3 Physical Design

- System design (or general design or system specifications)
- Detailed design (or specific design)

2.2.4 Implementation or Construction

- system building
- testing
- installation / conversion
- operations (refinement / tuning)
- post-implementation review

2.2.5 Maintenance

- maintenance and enhancements

The above procedure seems to have a sequential order of doing things, but this need not be the case. In fact the steps in the development process have an iterative nature. This means that work on one step might require the systems developer to go back to the previous step(s) or phase(s). The result of going back might mean that what has been done up till now has to be completely revised.

The classical approach states that if the systems developer follows the above steps of the information system life cycle then this should yield a successful information system.

2.3 Shortcomings of the Classical Approach.

A study done by Connor [Connor: 1980] concerning maintenance costs of information systems developed via the classical approach revealed that more money was spent on maintenance than on the analysis and design of the information system. The following diagram illustrates this.

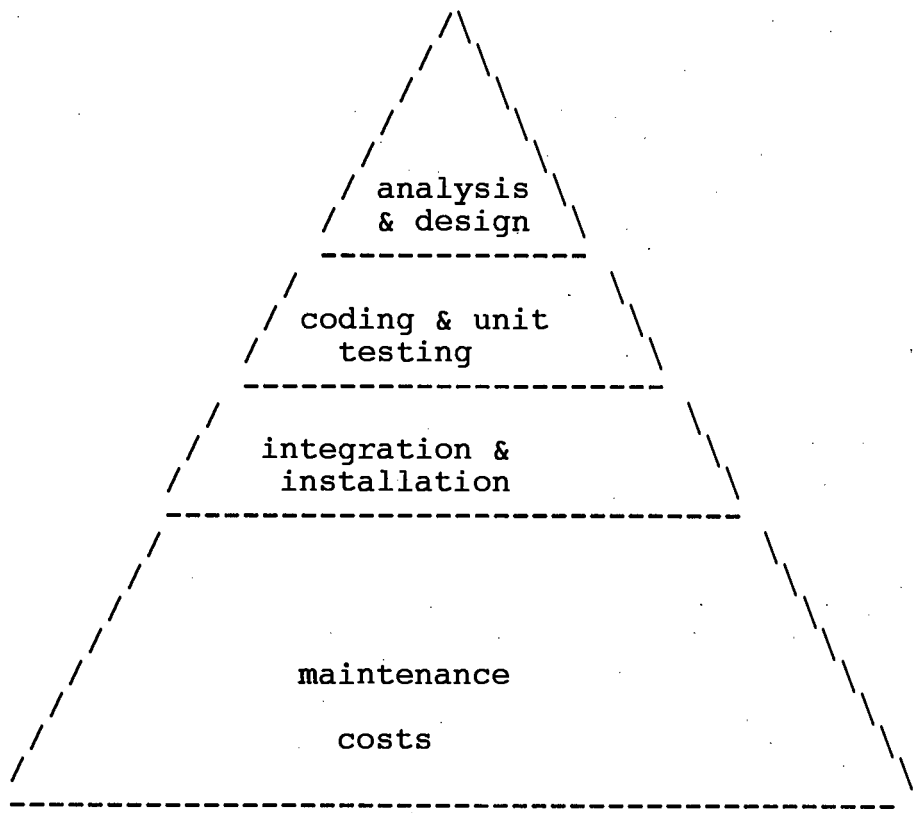


Figure 2.1 Systems' Development Costs.

The diagram below could possibly explain the reasons for the above situation [Aktas :1987 p24]:

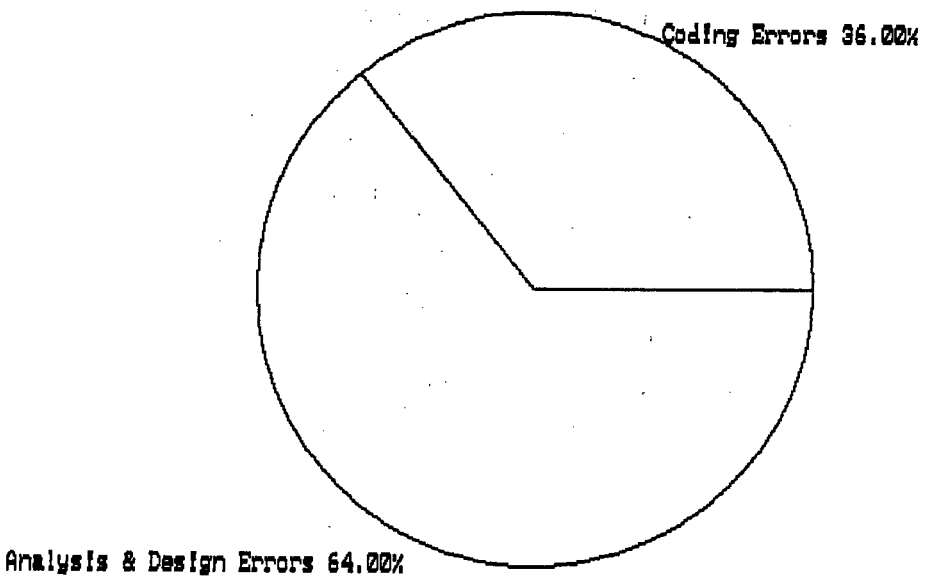


Figure 2.2 System Development Errors.

As a result of the above findings the following questions arise: Why has the classical approach failed and how can it be improved, changed or perhaps even replaced?

Much of the emphasis of the classical approach has been on technical personnel rather than on the user.
The classical approach normally incorporated user input during the planning and analysis phases. However, experience has shown that the user requirements are not always clear and/or correct and could also change as the system development cycle progresses. Errors made by the user during the initial

requirements specification have a serious impact on the system produced, which inevitably leads to user dissatisfaction.

Dissatisfaction with the classical approach has led to disturbing comments, for example:

"We build systems like the Wright brothers built airplanes - build the whole thing, push it off a cliff, let it crash, and start over again" [Graham: 1969].

"There is a widening gap between ambitions and achievements in software engineering. The gap appears in several dimensions: between promises to users and performance by software: between what seems to be ultimately possible and what is a achievable now: between estimated costs and expenditures. This gap is arising at a time when the consequences of software failures in all its aspects are becoming increasingly serious" [David & Fraser: 1969].

A number of basic approaches have been suggested to improve the ways in which successful systems can be developed in an economical manner.
Among the remedies proposed are: powerful high-level language structures, paradigms which can help analysts in the design and structuring of software systems and the provision of development processes and environments [Henderson et al :1987 p12].

2.4 Summation: Classical Design Strategies.

Development methods have to change from the classical, manual , non-structured and sometimes ad hoc methods to disciplined and well thought out strategies. Advances in software development have been much too slow and incommensurate with hardware technology developments [Reps et al :1987 p29].

2.5 The Structured Approach.

Professor Edsger W. Dijkstra is often considered the founder of structured programming [Dijkstra: 1969 p84-88]. He also presented top-down design, a technique which seemed to go hand-in-hand with structured programming.

Structured design and development of programs according to James Donaldson in his original article [Donaldson: 1973 p53] can be defined as a manner of organizing and coding programs that makes them easily understood. The essence of the whole problem is to simplify the control paths in a program or design.

A more modern definition of structured programming [Martin & McClure: 1985 p41] is that "structured programming is a methodology that lends structure and discipline to the program form, program design process, program coding and program testing". Structured programming is a programming methodology for constructing hierarchically ordered modular programs using standardized control structures incorporating stepwise

refinement, top-down and bottom-up programming.

Structured techniques include the concept of breaking a problem down into smaller manageable parts.

2.6 The Objectives of Structured Techniques.

According to Martin [Martin & McClure: 1985 p5] the primary objectives are:

1. to achieve high quality programs of predictable behavior.
2. to create programs and systems which are easily modifiable and maintainable.
3. to simplify programs and also the development (construction) of programs.
4. to maintain control and predictability in the program / systems development process.
5. speed up the system development process i.e improving productivity.
6. lower the cost of systems development

Another objective which should be incorporated is:

7. improve communications with end users in order to involve more end user input in the design and development phase.

A number of secondary objectives are desirable in order to meet the primary objectives [Martin & McClure: 1985 p6]. These include:

1. Tools and techniques which can report errors as soon as they are recognized and provide the programmer with immediate feedback. This requires more advanced structured techniques as well as computerized validation, preferably on-line at a screen.
2. Use automatic code generators where possible - a much higher quality of code can be generated which can also be checked automatically.

2.7 The Advantages of Structured Techniques.

Goldberg [Goldberg :1986 p340], Bromberg [Bromberg :1984 p75], Kowalski [Kowalski :1984 p92], Wasserman and Gutz [Wasserman & Gutz :1982 p202], DeMarco [DeMarco :1979], Yourdon [Yourdon :1975 p140-144] and McCracken [McCracken :1973 p52], amongst others regard increased programmer productivity as an important advantage of structured techniques.

Other advantages identified are fewer testing problems (the larger a program the more expensive the cost of testing it) and clarity and readability of programs.

2.8 Structured Diagramming Techniques.

Diagrams clearly show the structure and inter-relationships within a system and thus provide support for system changes. Many diagramming techniques have evolved as part of the structured technique philosophy.

The following are amongst the more important functions that a diagramming technique should provide [Martin & McClure :1985 p9]:

* framework for clear thinking
* systems documentation
* enforce structured techniques
* aid the maintenance task
* fast design with computer supported diagramming
* linkage to automatic code generation

2.9 Examples of Diagramming Techniques.

A vast number of diagramming techniques exist for software engineering. Only these relevant to SSDE will be described below.

2.9.1 The HIPO Diagramming Technique.

HIPO is an acronym for Hierarchical Input Process Output. The technique can be used to illustrate the input, output and functions of a system. The HIPO diagrams show what the system does (tasks), rather than how it actually accomplishes these

tasks, i.e it emphasizes the functional components of the system. Each box in the diagram can represent a system, subsystem, a program or a module. HIPO charts can be used in either the analysis or the design phase. At the highest level of the HIPO chart, the highest level goals of the system are identified. In subsequent levels (i.e lower levels) the high-level goals identified to date can be further functionalized with more and more detail. HIPO charts therefore makes it possible to follow a top-down design process. An example is given in figures 2.3 and 2.4 below.

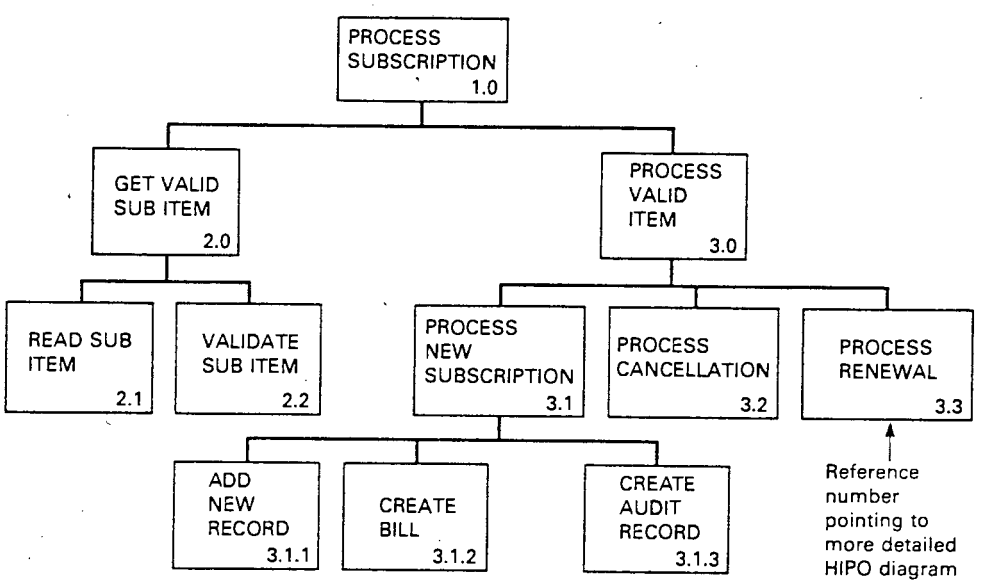


Figure 2.3 A High-Level HIPO Chart.

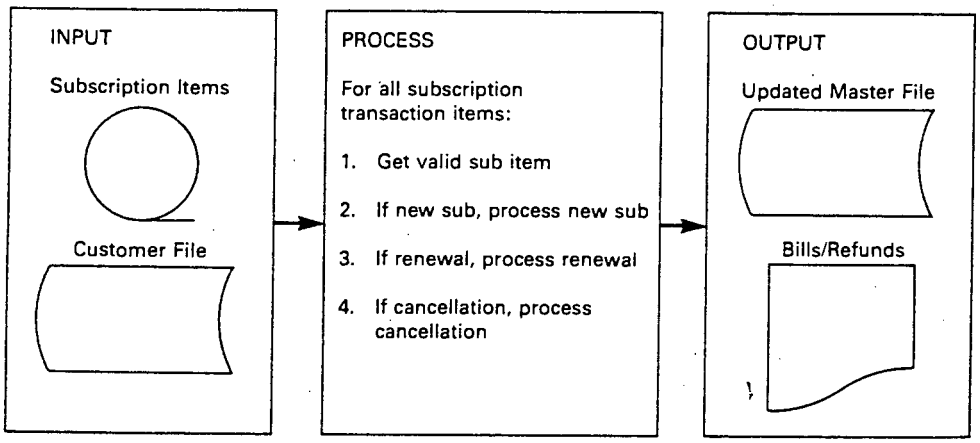


Figure 2.4 A Detailed HIPO Chart.

2.9.2 The Structure Chart Diagramming Technique.

The Structure Chart diagramming technique is very similar to the HIPO diagramming technique in that it also emphasizes components (modules) and their relationship with each other. A hierarchical top-down design structure is also possible here. Additional information that can be shown here (but which is not part of the HIPO chart) is data and control information passed between modules. An example is given in figure 2.5, which also serves to illustrate the distinction between HIPO and Structured Charts.

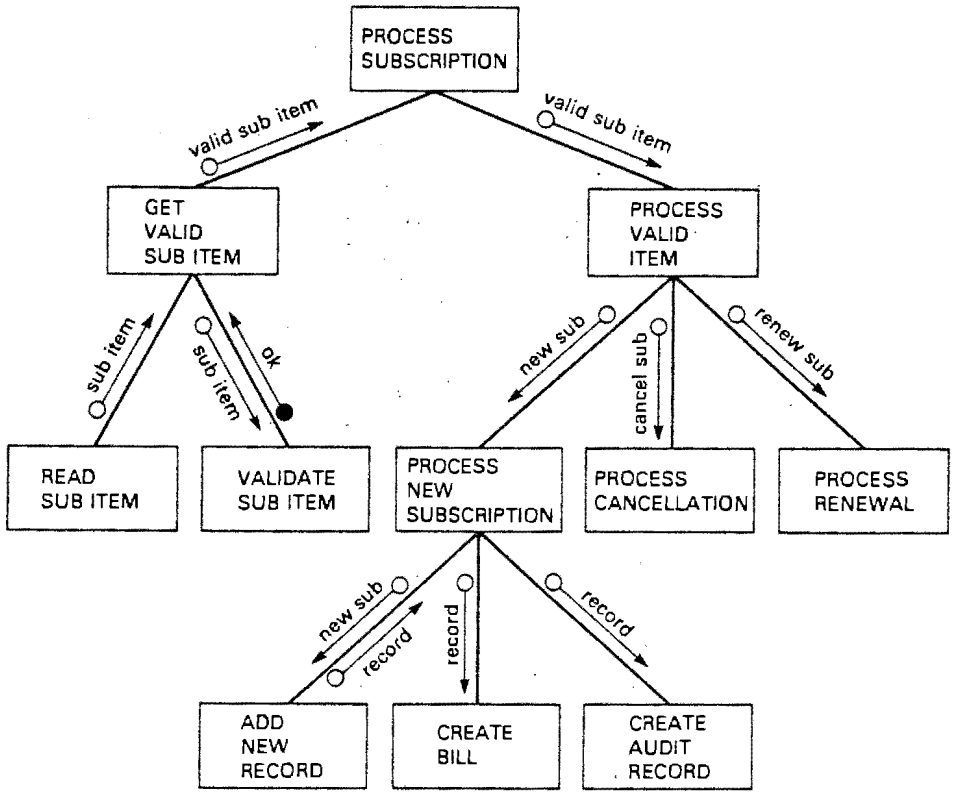


Figure 2.5 A Structure Chart.

These two diagramming techniques are well suited to illustrating the high-level functional components of a system. They do however lack constructs to show low-level design logic and hence SSDE incorporates a low-level diagramming technique to augment this [Martin et al :1985a], [IBM HIPO :1974].

2.9.3 The Nassi-Shneiderman Diagramming Technique.

Nassi-Shneiderman (N-S) charts can represent low-level program constructs sequence, selection and iteration.
This technique is well suited for designing low-level (detail) program logic. Using this as a design technique, results in structured program with its associated advantages [Martin et al :1985a], [Nassi et al :1973]. An example appears in figure 2.6.

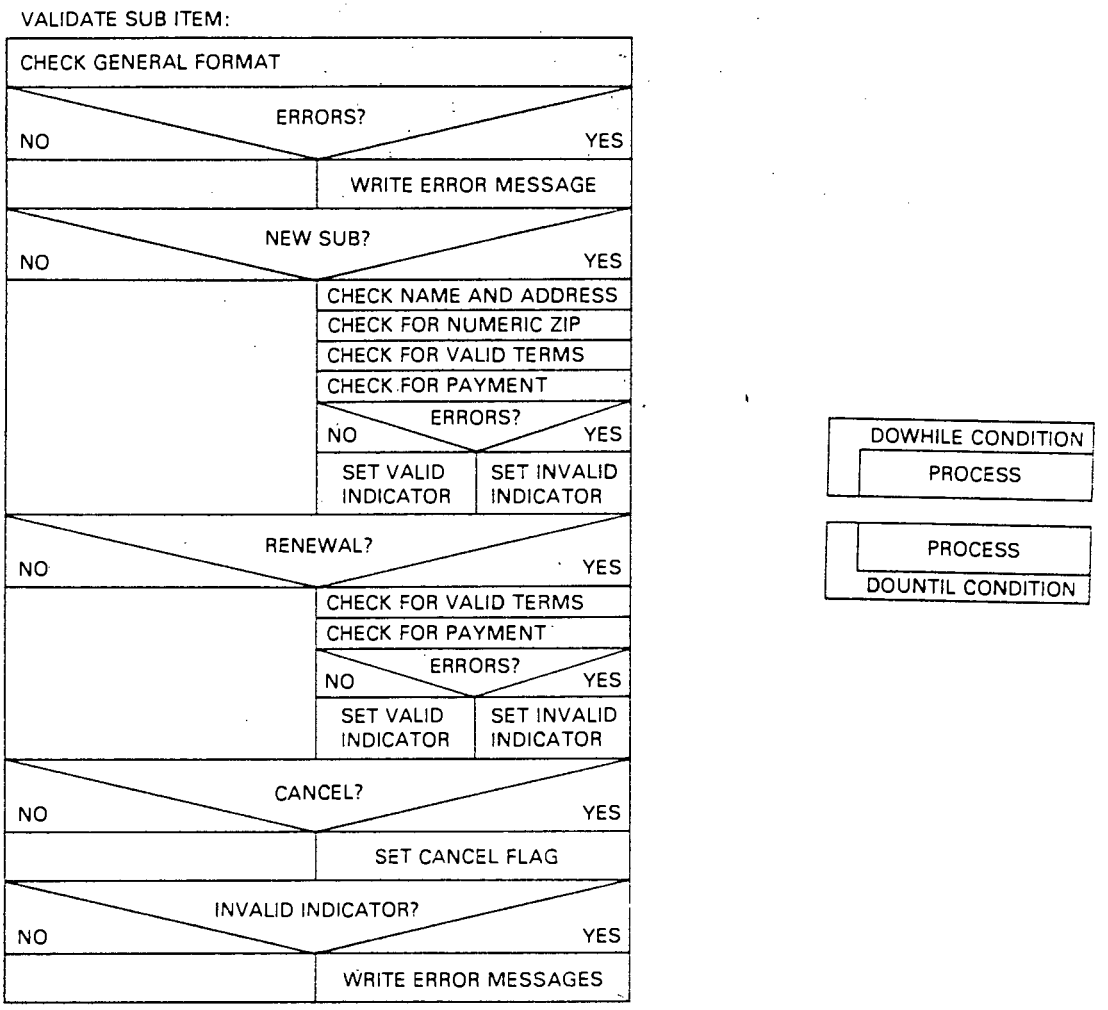


Figure 2.6 A Nassi-Shneiderman Chart.

2.10 Commentary.

It is clear from the literature that in order to produce programs and systems with quality features, it is essential that design techniques and strategies be used. To achieve quality software, the old ad hoc methods of development had to be discarded and replaced with new disciplined and scientific procedures.

Many diagramming techniques are in existence today since no single diagramming technique has yet emerged which satisfies all design preferences. Among the preferences which designers have are: good control structures, showing data flow, showing complex logic, functional decomposition, linkage to fourth generation languages, cross-reference checking, etc. In order to accommodate these preferences a combination of techniques will have to be employed by the designer.

2.10.1 HIPO.

At the high-level design of a system I prefer the HIPO technique since it shows what the system does, rather than how. A hierarchical representation of the functional components which constitute the system can be drawn. For the logic of components which are not complex, a detailed HIPO can be drawn showing the input, process steps and output. Using these aspects of HIPO charts, I can quickly draw a hierarchical view of the system, and the logic of any trivial components. It is useful here as an analysis and design tool for the high-level functional representation and also to represent a data structure.

The HIPO chart cannot show how these components are related via data and furthermore a module can only be invoked by one caller. The Structure chart can be used to address these drawbacks. Another limitation of HIPO is that the detailed process diagrams are not suitable for showing program structures such as condition, case and loops. A technique like

the use of Nassi-Shneiderman diagrams is necessary for complex detailed logic design.

2.10.2 Structure Chart.

Structure charts have certain similarities with HIPO charts (e.g they both show what the system does) and this is the reason why I use it to augment the shortcomings of HIPO as regards the high-level design of the system. It can show how the components of the hierarchy are related via data. It can also let one component be invoked by multiple modules and this reduces having to duplicate components as is the case in HIPO. Structure charts also do not show control structures such as sequence, selection and iteration, as do Nassi-Shneiderman charts.

2.10.3 Nassi-Shneiderman Diagram.

The main reason why I prefer this technique is that it allows me to represent detailed program logic in a structured manner. This makes it easy to convert the diagram into structured code such as required in TURBO PASCAL. This technique has all the necessary control constructs to show sequence, selection and repetition. Also nesting and recursion can be easily illustrated. An important aspect I find useful is the facility to use more than one diagram (ie. secondary diagrams) to show complex logic. This permits me to design and test low-level code in a modular fashion and allows a system to "grow" as each module is implemented.

The Nassi-Shneiderman chart cannot be used for designing the high-level hierarchical control structure. Although it is easy to read a Nassi-Shneiderman diagram, is not easy to draw by hand particularly if the diagram needs to be edited. Automation of the technique can alleviate this problem and improve design productivity, particularly as regards editing.

2.10.4 Other Diagrams.

Flowcharting is a diagramming technique I would not use because it is not a structured technique. Although it has good graphic symbols, the arrow lines drawn between these symbols encourage the use of the unstructured "GO TO" statement.

Data flow diagrams are useful for charting the flows of documents and computer data in complex systems. My main reason for not using this technique is that as systems become complex with many data items, the flow chart becomes very cluttered. This complicates the task of cross-checking the consistency of all inputs and outputs. This type of discrepancy can result in a rectification task when coding which could be time-consuming.

Warnier-Orr diagrams can graphically represent the hierarchical structure of a program, a system or a data structure. Warnier-Orr diagrams are an alternative for HIPO and Structure charts with a number of similarities. It can be used for high-level and low-level logic. My preference for Structure Charts is because it is drawn from the top to the bottom of a page, unlike Warnier-Orr diagrams which are drawn across the

page. For complex low-level logic, Warnier-Orr diagrams become large and difficult to read. Another shortcoming at the low-level is that Warnier-Orr do not show conditional logic.

2.10.5 Conclusion.

There is now an increasing emphasis on the design and construction of a system. It is no longer sufficient to produce a system which only supplied the correct output. Systems have to make optimal use of the available resources, be reliable, maintainable and convenient for the user. HIPO, Structured charts and Nassi-Shneiderman charts are also a form of documentation and can thus assist in the maintenance task by showing the influence and impact any change would make. Nassi-Shneiderman diagrams promote structured design which results in structured code being produced. Structured code provides clear and standardized specifications which, if properly tested, leads to reliable code. In this way errors are reduced and productivity improved resulting in resources (e.g manpower) not being wasted.

2.11 Summary.

Diagramming techniques are part of the structured development philosophy. Combinations of diagramming techniques must be employed since no single diagramming technique is adequate for the high-level definition of a system as well as for the low-level definition.

CHAPTER 3:

SOFTWARE DEVELOPMENT :

AUTOMATED TOOLS AND ENVIRONMENTS

"Demand for reliable software systems is stressing software production capability and automation is seen as a practical approach to increasing productivity and quality" [Hoffnagle & Beregi :1985 p102].

3.1 Introduction and Definitions.

In brief, the ultimate goal of software development environments is to support and simplify the software development (life cycle) process.
The two major areas of research are concerned with programming environments, and with system development environments, respectively [Haberman et al :1986 p1117].

Programming environments can be defined as a collection of software tools, computer support, management controls, databases, libraries and other tools, facilities and procedures which all work together to help in supporting the programming process [Mathis :1986 p708].
The environment should also provide the necessary mechanisms to expedite the analysis, documenting and verifying phases [Rzepka et al :1985 p9].

In the broader context, a system development environment can be defined as a combination of software tools and methodologies which support the complete software development life-cycle.

This new software development technology has been named variously CASE (Computer Aided Software Engineering), Programming Process Support System, Life Cycle Software Engineering Support Environment [Mathis :1986 p708], Integrated Development Environments [Ross :1985 p33], Requirements Engineering Environments [Rzepka et al :1985 p9] and Front-end Programming Environments [Zvegintzov :1984 p80].

CASE Technology can be defined as an environment, consisting of a collection of software tools, which can support the activities of a system development methodology [Bornman :1989 p1]. The essential elements in a CASE system are procedures (e.g life-cycle methodology), methods (e.g design techniques) and the integration of automated tools (e.g a database manager, diagrammers, program editors, debuggers, documentation tools).

A list of existing Software Development Aids appears in Appendix E.

As a result of the "Software Crisis", the main challenge facing software development today is the achievement of high quality software and increased productivity in software development. Figure 3.1 shows important software quality features. The "Software Crisis" taken together with the predicted shortage of skilled software personnel has resulted in the increasing

tendency among ordinary computer users to develop their own software for their particular applications. This has produced a number of automated software development tools and software environments geared towards end-users.

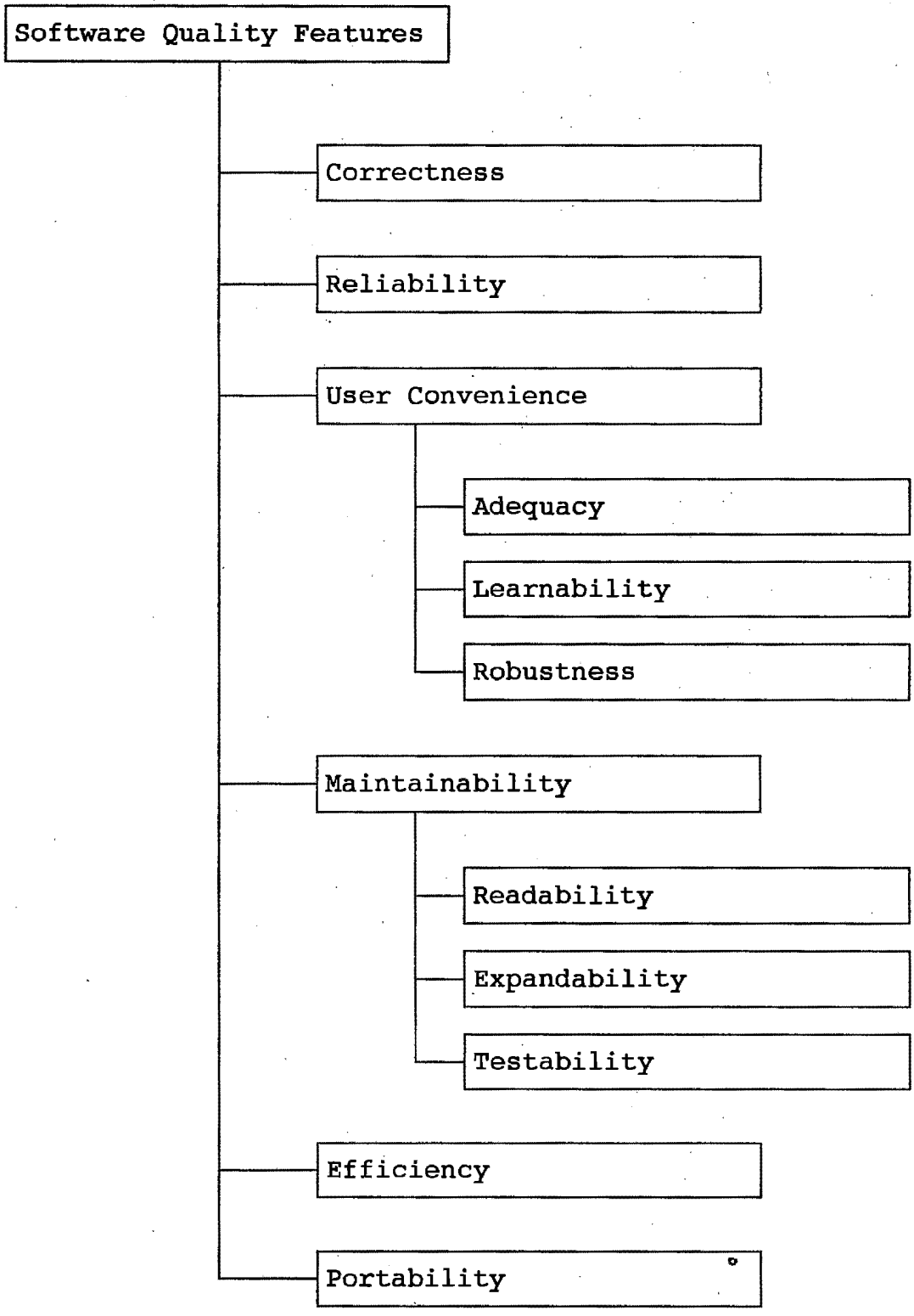


Figure 3.1 Software Quality Features

3.2 Benefits of CASE Usage.

A study of CASE Technology done by the Sloan School of Management at MIT, has identified the following eight benefits provided by the use of CASE Technology [Computing SA :1989a].

Consistency.

Design changes are fully communicated to all other participants involved with the project.

Diagram Merging.

Two separate design concepts can be merged into a single diagram so that differences can easily be seen.

Simulation.

Design concepts can be simulated thus avoiding the construction of prototypes.

Generic Transformation.

A specific model can be transformed into a generic one which can then be adapted by other interested parties.

Decentralized Design.

Designers located in different locations can communicate and participate in a design.

Project Management.

Timesheets can be incorporated into the system to ensure that budgeting constraints are being adhered to.

Personnel Independent.

If key personnel leave the project, then information is not lost.

Decreased Programming Effort.

Fewer programming staff are needed.

3.3 CASE for the PC.

This section briefly reviews a few CASE products which have similar objectives to SSDE and are all PC-based.

(a) FOUNDATION.

FOUNDATION covers the entire systems development process and was developed by Arthur Andersen. It runs on a personal computer and the systems designed using FOUNDATION run in the IBM/MVS/DB2 mainframe environment. It consists of three main components, viz. METHOD/1 (project management), DESIGN/1 (planning and design) and INSTALL/1 (implementation and support). DESIGN/1 allows design via Structure Charts and Data Flow Diagrams. INSTALL/1 can amongst other things generate about 90% of the COBOL code for the system. The COBOL code is generated from screen definitions, data dictionary definitions, files, records, conversation flows, etc.

(b) HP Teamwork/SD.

HP Teamwork/SD forms part of an integrated environment for structured systems development and reduced life cycle cost. It features a structure chart editor, data dictionary entries, module specifications which describe how a module in the chart will perform its function and consistency and completeness checking. It can interact with other HP Teamwork products, e.g HP Teamwork/SA (Structured Analysis) [Hewlett-Packard :1987].

(c) Yourdon/Analyst Designer Toolkit.

Fully menu-driven, it permits an automated structured development process using Yourdon diagrams. The following diagrams can be created: context, entity-relationship, data flow, state transition and structure charts. A project dictionary can contain full textual specifications. Consistency and accuracy checks can be done. The toolkit also generates documentation. The Yourdon methodology can be followed, but an option permits the designer to modify the methodology to suit his requirements [Yourdon International :1988].

(d) Software through Pictures.

Software through Pictures (StP) is an integrated set of tools that aid in the entire software development process. It is supplied by Interactive Development Environments. The application and tools range from requirements traceability,

graphic editors, through to code generation (ADA, C, Pascal) and has multi-user support. It also provides support for applications which involve conversational access to a database [Hewlett-Packgard :1988 p7].

(e) MICROSTEP.

MICROSTEP by Syscorp International, is a computer-aided software engineering product that can produce up to 100% C source code from graphic specifications. It has an interactive graphic design environment, automatic specification analysis, generation of executable code and can produce documentation [Datamation :1989 p71].

(f) IEW/CWS.

Information Engineering Workbench / Construction Workstation (IEW/CWS) was created to generate COBOL applications from diagram specifications created by IEW/Design Workstation. It generates the necessary COBOL source code, database schemas, access routines and documentation for an application [Computer : 1989 p98].

(g) Other Toolkits.

A number of toolkits which support development in specific areas are available on the PC. For example: Analysis and Design Toolkits; Database and File Design Toolkits; Programming Toolkits; Maintenance and Re-engineering Toolkits; Framework

Toolkits and Project Management Toolkits [McClure :1989 p246]. See also Appendix E, which includes PC products.

3.3.1 CASE - A Critique.

At this stage in the development of CASE products a number of criticisms have been mentioned:

* at present there are no standards in the CASE field. The Software Institute at CMU and the Center for Advanced Information Management at Auburn University are producing standards for CASE [Gibson :1989 p209].

* many CASE tools require a level of software development knowledge that many users do not possess.

* much user time is spent entering a large amount of data into the CASE tool.

* the learning curve for CASE tools is fairly steep and extensive training and experience is required to adequately use a CASE product [McClure :1989 p242-244].

* Methodology training may also be required if the developers or even users are not familiar.

* from a management point of view it is often difficult to cost justify using a CASE tool in an organization [McClure :1989 p242-244]. The benefits of CASE can be said to be more long

term.

* initially when a project is started, it is difficult to convince a user that he is getting his money's worth. This is so because much time is spent on analysis and design. (Users think they pay for code only.)

* an experienced CASE user has noted that CASE products are unlikely to replace the skills and expertise of human designers in producing quality systems [Wilk :1989 p13].

3.4 The Future of Computer Aided Software Engineering.

It is evident from the proliferation of CASE products (see appendix E), that vast amounts of money are being invested in research into the development of software environments [Mathis :1986 p711-712]. At the moment the products are designed mainly for people who are computer literate in order to speed up their productivity and the quality of the products that they are producing.
Because of the successes of some environments to date, much research is currently being done in software engineering environments [Brooks :1987 p13] [Stamps :1987 p56] [Reps et al :1987 p30] [Henderson :1987/84].

3.5 Research Areas.

The following is a list of areas where research in Software Engineering automation is most likely to be concentrated:

3.5.1 Increased Component Re-using.

In-order to reduce design activities and code writing, increased use should be made of existing designs and code, in the construction of new software systems. Libraries of available software should be established and made accessible to software designers [Boehm :1987 p54].

3.5.2 Improved Automated Tools.

Many of the automated tools force the systems designer to follow one or other inherent design methodology. To many designers this is a distinct disadvantage since they may not be familiar with that particular design method. Automated tools which therefore allow the designer to follow his own particular methodology, will be increasingly sought after. Furthermore, these environments should efficiently automate as many of the development processes as possible, particularly trivial tasks, in order to free the designer so that he can concentrate more on the actual software design logic [Boehm :1987 p57].

3.5.3 Distributed Software Engineering.

Possible candidates for distributed software engineering are Strategic Defense Initiative Systems, process-control systems such as nuclear-reactor control systems and embedded monitoring

systems.

In these systems there is a great need for correct, reliable and maintainable distributed software [Shatz et al :1987 p23].

3.5.4 Reverse Engineering Tools.

The automated reverse engineering tool provides a technique which makes it possible to extract physical or logical models from existing databases. Reverse engineering seeks to incorporate CASE technology to accomplish database design and maintenance [McWilliams :1988 p30].

3.5.5 Using "Fragtypes" as a Basis for Programming Environments.

This type of environment advances the use of fragments of various forms, called fragtypes, in the development of software. These fragtypes can range from simple expression types to complete sub-system types. They can realize unusually powerful effects on the development of software [Madhavji :1988 p85].

3.5.6 Life-Cycle Support in Software Engineering.

Life-cycle support will become an important factor in the production of quality software. A number of environments which provide complete life-cycle support have already evolved, eg: Gandalf [Haberman et al :1985], Genesis [Ramamoorthy et al :1985] and Saga [Kirlis et al :1985]). Much more research is

expected in this area [Ramamoorthy et al :1987 p35].

3.5.7 Intelligent Tools.

AI techniques are now being used in software development. Two such systems are : the Programmer's Assistant (MIT) and Psi (Stanford University). These systems attempt to model the knowledge programmers use in understanding, designing, implementing and maintaining programs. Thus expert systems can use this knowledge to automate some of the software development process [Tichy :1987 p43]. Certain optimization tasks such as data structure selection may be implemented using AI technology [Ramamoorthy et al :1987 p35]. Knowledge-based tools have already been reported in the Rome Air Development Center Report [Frenkel :1985] and the Japanese Fifth Generation Project [Moto-oka :1981].

3.5.8 Visual Programming.

Program visualization (PV) (i.e graphically displaying program and / or system documentation) strives to help designers to form precise mental images of system structure and function. PV tools can manipulate static and dynamic computer system diagrams, program and documentation texts. The challenge which PV researchers need to address is to develop unified tools which support the complete life-cycle. As new and improved graphics hardware devices become available the art of PV can be extended and enhanced. A number of pioneering systems incorporating PV features have already been developed [Brown et

al: 1985 p27].

3.5.9 Special-Purpose Languages.

This type of language will permit the designer of a system to enter his design specification in a high-level language, and then the special-purpose language will translate these specifications into efficient executable code [Bornman :1989 p11].

3.6 Commentary.

3.6.1 CASE Features.

(a) Life Cycle Support.

It is important to note that CASE products are classified into two general categories, namely, a CASE Tool(kit) or a CASE Workbench. A Toolkit specializes in the automation of a particular software task and a Workbench supports the full software development life-cycle which could include either a rigid or flexible development methodology. "Foundation", with its three major components supports the full life-cycle and in addition to this, the "Method" component provides management with an integrated facility for project control. The latter feature is lacking in many CASE products and project managers are forced to use another stand-alone management facility.

"Teamwork" comes as a family of products which together support the software development life-cycle.
"IEW" is a single workbench consisting of a number of components which support the entire life-cycle.
My preference would be for a workbench CASE product rather than for a toolkit because the former provides support for the complete life-cycle.

(b) Features.

The Structured Analysis/Design methodology is supported by all these products. All these provide for the drawing of structure charts and dataflow diagrams and conclude consistency and completeness checking which is useful because of the many inputs/outputs in a complex system. Customized methodologies are also permitted.

Each product has a central database where diagrams, specifications, data dictionary entries, design and documentation information is kept. This reduces redundancy, assists in consistency checks and helps with team communication.

In these areas, current CASE technology thus provides adequate function and flexibility, giving us useful and sophisticated tools.

Code generation is now becoming more common because of the accurate specifications (e.g screen layouts) kept in the central design database. Except for the "Yourdon" Toolkit, the

other products all generate code from specifications. This is a useful productivity aid.

(c) Ease of Learning and Ease of Use.

Experience with CASE products have shown that it takes time to learn how to use a CASE product. All these CASE products do have on-line help tutorials and other help packages. The productivity gains often mentioned can only be realized once the designer is familiar with the CASE tool and its related (or designer chosen) methodology. Here "IEW" seem less complex and thus easier to learn and less time-consuming to use.

3.6.2 Research Areas.

The most important research area seem to be the incorporation of artificial intelligence and expert system technology in CASE products. This will promote the more general use of CASE technology by new designers, shorten the lengthy learning time and reduce the dependency on the skills of a specific person. This will make CASE products more attractive for potential users.

3.6.3 Conclusion.

CASE products are expensive and difficult to cost justify since their advantages are more long term. The advantage is more the quality of the resulting system (e.g reliable code, less maintenance) using a methodology and CASE, rather than the

increased productivity. As new concepts are incorporated into CASE, it becomes more generally used and as pioneering CASE users report their long term gains, the future of CASE products will be secured.

3.7 Summary.

Stand-alone automated tools or environments providing a number of integrated tools (e.g CASE technology), have the potential to provide the analyst with complete support throughout the software development life-cycle. Furthermore, as it becomes possible to have built-in intelligence and expertise in such a tool or environment, the scope of usage can be extended to include ordinary computer users with little or no system building exposure.

CHAPTER 4:

MOTIVATION

4.1 Introduction.

Many software specialists have noted that there are many problems associated with the design of software.

These problems can be addressed by considering new design techniques and procedures, by automating as much as possible of the software design task and by providing tools and/or environments which support and assist the system designer [Schindler :1981 p190-191].

4.2 Software Crisis.

Wasserman [Wasserman :1980 p5] and others [Goldberg :1986 p334] have shown that a significant number of software systems are unsuccessful in that they do not satisfy the requirements of the application or desired system. As far as those systems which were successful, Wasserman states that these were normally late, over budget, expensive in terms of resource utilization and/or poorly suited for the intended users of the system.

Parnas [Parnas :1985 p1327] also states that much of the software we have today is unreliable and it is quite usual to have a system with a number of "bugs" which does not work reliably for some users. He points out that because of this,

many software products carry a specific disclaimer of warranty. In certain extreme situations, the complete resulting system could be useless. Large amounts of time and money would have to be spent to reduce the gap between the system delivered and the goals set [Roman :1986 p14].

4.3 Cost Implications.

The economic ramifications of these software problems are vast and concern provoking.

There is an information systems development backlog of more than four years. A 1987 survey by Datamation / Cowen & Co showed a 30% backlog in software applications [Stamps :1987 p55]. A hidden backlog of applications, the so called invisible backlog also exist, and has been estimated at eight years [Martin :1982].

The relative costs of hardware components are decreasing drastically (from 90% in the 1950's to 10% in the 1990's), whereas the relative cost of software is increasing at a similar rate [Wasserman & Gutz: 1982 p196-206], [Schindler :1981].

A principal expense in computer applications is that of program and systems maintenance [Rogers :1983 p199].

Furthermore, about two-thirds of the maintenance cost can be attributed to design misconceptions [Ramamoorthy et al :1984 p191-209], [Manna :1974].

As man becomes more and more computerized the need for reliable software will grow substantially and an army of software people will be required to meet the software requirements of the 1990's. Goldberg [Goldberg :1986 p334] suggests that new approaches and tools (e.g automated environments) will have to be produced to avoid an on going "Software Crisis". Many of the standard, well defined, menial tasks associated with software design should be automated as far as possible [Henderson et al :1987 p13].

4.4 Software Solutions.

Research will eventually focus on developing software environments that will present an analyst/user with all the facilities to build an application from scratch to a final implemented form which will meet with all his requirements.

Many users are not literate in program and application writing, but are able to solve problems in their particular domain [Brooks :1987 p17]. This should encourage the development of tools which will enable users to construct their own solutions.

4.5 Diagramming Techniques.

Complex system and program logic can be made much easier to understand with a good, clear diagramming technique. Diagrams advance team communication and enable management to implement and monitor control. Diagrams are also language independent [Dart et al :1987 p21] and provide a design consistency [Rogers

:1983 p199]. Furthermore, the trend to-day is to involve the end-user much more in the design phases of a system, and the use of language-independent diagrams will greatly encourage the users' contribution [Rzepta et al :1985 p10] [Raeder :1985 p11] [Grafton et al :1985 p7] [Brown M et al :1985 p28-39].

4.6 SSDE Objectives.

The aim of SSDE is to provide an automated environment for software engineering students. As such it must be inexpensive, easy to learn and use, and flexible. It must cover high-level and low-level design and be targeted at the personal computer running DOS. Naturally it can also be a valuable aid in solving real-life problems.

SSDE is directed at personal computer users, rather than the user of a large or medium-sized machine. The main reason for concentrating on the small user is because of the declining popularity of mainframes [Rushinek & Rushinek :Jul 1986 p598] which is being spurred on by the tendency to rewrite or revise mainframe software for micros (eg. SPSS/PC and SAS/PC). Also, users feel more in control of microcomputers, since they are often the only user, and hence have a more positive attitude towards them. Powerful workstations for smaller machines with advanced input/output facilities and increased main memory are fast becoming popular amongst analysts [Henderson et al :1987 p13].

Even applications which will eventually run on a mainframe can be developed on a personal computer using SSDE.

Until recently, CASE tools cost in the five- and six-figure price range and only ran on mainframes. This situation has changed greatly with CASE tools now being able to run on personal computers [Tazelaar :1989 p206]. Nevertheless prices remain significant. A major aim of SSDE is to be sufficiently cheap for all users including students, etc.

4.7 SSDE and Diagramming Techniques.

Diagramming can be considered a vehicle for clear, concise and structured design and it is for these reasons that SSDE uses diagramming techniques for both the high-level and the low-level design phases.
SSDE allows the analyst to construct a system from a high-level overview diagram to a final low-level diagram for which the system will automatically generate program code.

A form of combined HIPO/Structure Chart was chosen for the high-level phase of the design methodology. Nassi-Shneiderman Charts are used for the low-level design of the software logic. All these methods are integrated into an automated environment where each is used at appropriate stages of the design process.

4.8 Using Structured Design.

A major design goal of SSDE was to employ structured and modular techniques. It has already been shown in chapter 2 that such techniques offer many advantages over the adhoc methods [Karimi et al :1988 p196] [Henderson et al :1987 p12] [Davis :1986 p145] [Pomberger :1984 p10].

4.8.1 High-Level Structured Design.

The diagramming technique for the High-Level Design will be a combination of the HIPO and Structure Chart Techniques.

(a) Motivation.

The reasons for choosing these diagramming techniques for the high-level design are:

* they are well suited to illustrating the high-level functional components of a system being designed [Brooks :1987 p14] and the relationships between modules (components) [Hartzband et al :1985 p40];

* an indication of which data is moved between modules is provided.

* The method of capturing high-level design information is simplified with the help of these techniques [Roman :1985 p16] [Rogers :1983 p201].

* HIPO representation is well suited for the global view of the system which is shown with no clutter of data items

* the Structure Chart representation adequately displays the functional components and their inter-relationships which is required by the local (zoom in) design view. It enables data transfers between several interrelated components to be displayed in a compact manner.

(b) Limitation.

These high-level diagramming techniques are lacking in constructs to show low-level design logic. It was for this reason that these high-level diagrams were combined with a diagramming technique which is well suited for low-level design. A low-level diagramming technique has all the necessary constructs to effectively design detail logic.

4.8.2 Low-Level Structured Design.

The structured technique incorporated into SSDE for the Low Level design is the Nassi-Shneiderman Diagramming technique [Nassi et al :1973 p12-26].

(a) Motivation.

The main characteristics and advantages of the Nassi-Shneiderman Diagramming technique are as follows: [Mathis :1986 p717]

* the functional domain is well defined. Three constructs are emphasized, viz. sequence, selection (or decision) and iteration (repetition). This has been extended by incorporating secondary diagrams.

* it does not allow arbitrary control transfer, i.e it enforces structured design

* it is easy to determine the scope of local and global data

* it is easy to represent recursive features

* step-wise refinement from a high level abstract design to a final more detail design is possible (i.e different levels of abstraction) [Henderson et al :1987 p14] [Raeder :1985 p13] [Wirth :1971]

* structures are clear and concise allowing a thorough understanding of the flow and structure of the program being designed [Stamps :1987 p56] [Cerveny et al :1987 p98] [Raeder :1985 p12] [Rzepka et al :1985 p10]

* clear and concise diagram structures are also an important aid in clear thinking [Martin et al :1985c p1]

Other advantages which I have found are:

* because secondary diagrams can be supported, it is possible to produce a design that is modular (divide and conquer principle is applicable here)

* debugging / verifying the design is simplified since modules can be considered one at a time (module testing can be accomplished by viewing it (the module) as a black box)

* duplicating similar activities is avoided, i.e previous designs can be re-used in structuring new and similar logic

* the PASCAL language has the necessary control structures required for the Nassi-Shneiderman design constructs

* diagrams are often a communication tool. Nassi-Shneiderman diagrams allow designers to interchange ideas and design thoughts and also facilitates the integration of their various components into a single system.

* simplicity and readability of the Nassi-Schneiderman constructs

* since the diagrams are clear and readable, it aids the maintenance task

* as the design "grows" the individual diagrams (modules) can be tested component by component. This aspect incorporates the advantages of separate compilation and type checking which is possible in PASCAL.

Other people have already implemented Nassi-Shneiderman Diagrams in software development systems, e.g Graphics-based Programming Support System [Frei et al :1978] and Graphical Interactive Monitor [Clark et al :1983].

4.9 Databases Used.

All information on any system which the analyst has designed, is kept in on-line databases. Any previously designed system with all its associated detail may thus be recalled, providing accelerated access to any high-level or low-level design. Data retention is a very important back-up measure if any key personnel members should leave the project.

4.10 Code Generation.

Writing code for design structures is a time consuming task when done manually. A prescriptive low-level design methodology is used allowing for SSDE to generate skeleton programs from

the low-level program design information. Incomplete text or abstract entries are generated as comments.

4.11 Documentation.

System documentation is useful for system maintenance or enhancements [Dart et al :1987 p20] [Pomberger :1984 p10]. SSDE provides documentation for the high-level hierarchical design and for the low-level detail design. This documentation can eventually contribute towards the contents of the project information file.

4.12 Summary.

A growing number of system designers, analysts and even end-users will in future seek automated design facilities (stand-alone tools or integrated environments) which will provide them with a vehicle for complete system analysis, system definition and implementation. These design facilities should not enforce too rigid a methodology, but permit any design strategy, or combination, with which the analyst is familiar. This project concentrates on building such a prototype automated environment.

CHAPTER 5:

SSDE: AN OVERVIEW

5.1 Introduction.

The automated environment constructed is intended to provide an analyst type of user with an environment consisting of several integrated design tools. These tools will assist him in the high-level design of a system and also with the design of the logic required for the low-level detail.

As the user constructs a system, a high-level to low-level design methodology is followed. Structured design, stepwise refinement and top-down design are encouraged. Rather than enforcing any one methodology on the user, SSDE provides the designer with a framework wherein he can follow a flexible design methodology.

5.2 The Environment.

5.2.1 High-Level Design.

High-level design is done in the context of the HIPO chart diagramming technique [IBM HIPO : 1974], extended with ideas taken from the Structure chart diagramming technique [Yourdon et al: 1979 appendix A]. These two techniques are combined into a hybrid diagramming technique and all the high level design is done in terms of this.

5.2.2 Low-Level Design.

The Low level design is done using an extended form of the Chapin [Chapin :1974 p341] or Nassi-Shneiderman [Nassi et al :1973 p12] diagramming technique. The extension provides for abstraction and supports the use of secondary diagrams in order to build larger programs.

The analyst can design his detail logic using this technique. He can edit or re-structure his design as it grows and changes. Secondary diagrams allow the process of stepwise refinement and modular design to be used by the system architect. Secondary diagrams are used when the detail logic of a certain specific step in the main (or primary) diagram occurs as another (secondary) diagram. This feature allows the designer to avoid large diagrams. It can also be used to postpone the consideration of some specific logic while designing the overall logic of the primary diagram. An abstract entry is also permitted if the designer is unable to show exactly how a step is achieved, but only states what it does, using natural language.

5.2.3 Code Generation.

Automatic generation of skeleton PASCAL programs from the low-level program design information is supported. Low-level constructs are translated into equivalent executable statements wherever this is possible. Incomplete text or abstract entries are generated as comments.

5.2.4 Menu-Driven.

The analyst interacts with the environment via a menu system as this is both fast and easy to use [Reps et al :1987 p29] [Wasserman :1981 p8/10]. The user can easily and quickly move between various parts of the environment. Status information is displayed throughout, informing the user of his present position within the components of SSDE and giving information about the high- and low-level systems currently under consideration. Extensive on-line help is provided in two alternative forms: brief and detailed.

5.2.5 Databases and Query Facilities.

A record of all the high-level and low-level design is stored in on-line databases. Any previously constructed system can easily be retrieved from the database to be viewed, edited or printed. Systems deleted are not physically removed and can be recovered. Other general database query facilities are also supported.

5.2.6 Auxiliary Facilities.

System documentation for both the high-level and low-level detail design is provided. As such it is valuable during the

design for teams working together, as a history of the design progress, an up to date record, etc. This documentation can eventually contribute towards the project information file. SSDE creates a skeleton user manual which the designer can edit using a word processor.
Mechanisms have been included at various points in the design process to ensure that design entries are meaningful in the context in which they appear.
Existing components from pre-designed systems can be incorporated into the design of a new system. This can be done with both the high-level and the low-level diagrams. Thus the reuse of existing software in the construction of new applications is made possible [Henderson et al :1987 p12]. Components can also be re-used within the same system.

5.3 Programming Information.

SSDE was developed and written in TURBO PASCAL version 3.0 [Borland :1986]. This language was chosen because it does not only provide a programming language, but a complete development environment. The program source is entered using an interactive WordStar-like editor. One of the advanced development features is a one-step fast compilation, which finds the error statement, takes you there and then instantly re-compiles. Error identification during run-time is supported. For example, if the program aborts, then the address supplied can be entered via the source code editor, and it will locate the statement where the error occurred. Automatic overlays are supported which allows large programs to run in small amounts of

memory.The language supports dynamic data structures which are extensively used in SSDE. The environment was constructed in a modular way and consists of a number of programs which are called ("CHAINed") as required. Data and information are retained in primary memory and any program which is loaded has access to the data. By defining the size of the data and code segments when compiling, it is possible to share data among different programs, since TURBO PASCAL does not automatically initialize variables.

The Database components were implemented using the TURBO PASCAL Database Toolbox [Borland :1985] which is a complement to the TURBO PASCAL programming language. The databases are implemented as B+ Trees. Three major tools are supplied as part of the Toolbox, viz., the Turbo Access system, the Turbo Sort system and the GINST general installation system. The Turbo Access system and the Sort program are provided in a modular fashion so that the programmer can include these as he requires them in his Pascal programs. The Database Toolbox retrieves information either randomly by key, or in sorted sequence. The following files belong to the Turbo Access system:

ACCESS.BOX (basic file creation and maintenance routines; eg. DataFile - used to declare the data file structure, MaxDataRecSize - specify the maximum record length, MaxKeyLen - determines the maximum key length, ADDREC - adds a new record to a data file and returns the record number of the newly allocated data record.)

GETKEY.BOX (search routines: SEARCHKEY returns the data record number associated with the first entry in an index file that is equal to or greater than a specific key value.
FINDKEY returns the data record number associated with a key.
NEXTKEY returns the data reference associated with the next key in an index file.
PREVKEY returns the data reference associated with the preceding entry in an index file.)

ADDKEY.BOX (used for inserting keys into the index files. The add will be unsuccessful for duplicate keys, if they are not allowed.)

DELKEY.BOX (used for deleting keys from index files)

All of these can be incorporated in programs which will simplify the manipulation of the databases created.
I have selected this database software since it is compatible with the TURBO PASCAL language and since the Toolbox permits the programmer very low-level control over data storage. It frees the programmer from the inconvenience of writing his own B+ tree routines. Furthermore, the Turbo Pascal language and the Database Toolbox package are relatively inexpensive and efficient.

5.4 Summary.

SSDE is an interactive, menu-driven development environment which permits high- and low-level system and program definition. It includes a central database repository, design documentation and code generation in addition to facilities for manipulating, verifying and re-using design diagrams. SSDE was developed using TURBO-PASCAL with the TURBO-PASCAL DATABASE TOOLBOX.

CHAPTER 6:

DEVELOPMENT ISSUES.

6.1 Introduction.

The initial reasons which prompted the development of SSDE are discussed. The evolution of SSDE together with some of the problems encountered are also mentioned.

6.2 History.

The author often has the task of teaching program design and development to undergraduate computer science students. This experience has shown that the issue is not so much to teach students the syntax of a particular programming language, but to proceed from an initial problem statement to a completed program. It was against this background that the idea of developing an environment such as SSDE crystallized.

6.3 Non-Automated Design.

The students used Structure Charts and Nassi-Shneiderman diagrams to design their systems and programs. This was done manually, resulting in a number of difficulties:

* time-consuming

* editing is difficult and amounted to a re-draw to accommodate any changes. Applying levels of abstraction meant having to re-draw each time.

* diagrams were often untidy and difficult to read (difficulty in communicating your ideas to others)

* converting the completed design into program code was time consuming.

* it was difficult to re-use portions of designs

* design errors and omissions often occurred

6.4 Providing Design Support.

An automated environment such as SSDE could address these problems by providing a framework wherein the programmer or analyst could design the necessary logic, with these difficulties eliminated.

6.5 SSDE : Development Reasons.

The reasons why SSDE was developed instead of using other products on the market, was because specific design techniques were being used in the teaching program. These techniques were Structure Charts for the high-level conceptual design and Nassi-Shneiderman diagrams for the low-level code design. No such customized product was available on the market at a reasonable price. CASE tools available are very expensive especially as a number of copies are required for large teaching groups. Also SSDE is flexible as regards design methodologies and such flexibility is important in education.

6.6 Development Problems.

In developing SSDE, a number of problems arose, which are discussed below. A prototype system was built, to which considerable changes were made in producing the final version.

6.6.1 High-Level Design.

The limited screen size had to be noted in the drawing and displaying of the HIPO-Structure Charts. Two modes of display were used: a global view (with limited individual component detail) and a local view where all the information about a functional component are shown in full detail. Six character names were used in the global view to permit as many modules to be shown as possible without the diagram becoming cluttered or the component identification becoming unintelligible.

6.6.2 Low-Level Design.

The challenge here was to capture the low-level design information in a language independent manner. This had to be sufficiently simple and flexible not to impede the design process, yet be such that code could be generated. Thus two general low-level entries for constructs were allowed. The first is a fairly rigid entry to allow for code generation. The other a high-level abstraction entry for which the strict syntax requirements were relaxed.

6.6.3 Databases.

Much design information (high- and low-level) had to be stored for a system. Furthermore some information entered (e.g the general comment associated with each high-level component) was optional. Considerable changes were required in most programs when the prototype's database designs had to be altered.

6.6.4 Validation Mechanisms.

To ensure that text entries are meaningful, was a problem. A mere diagram facility would not be sufficient. Thus various validation mechanisms are employed to ensure as far as possible that textual input from the designer is appropriate and can be interpreted.

6.6.5 Integrated Environment.

Stand-alone tools have the inconvenience of having to move back and forth between tools. Since SSDE is an integrated environment, the menu system had to be such that the designer can move freely from one activity to the next. Design activities from different levels can thus be randomly selected, so the SSDE system does not distract the user's thought processes.

6.6.6 Methodology.

A methodology or strategy is an important plan which can be followed to arrive at a "good" solution. Different designers use differing methodologies. The question arose: should a specific methodology be imposed on the user? It was decided not to impose a rigid methodology, since designing is a creative task and a flexible approach has a much broader appeal. Thus SSDE allows the designer to follow a flexible design strategy.

6.6.7 Implementation.

As designs could be large, it was essential to utilize as little space as possible in representing high-level components, Nassi-Shneiderman diagrams, symbol tables, etc. Databases were converted to 3rd normal form to save disk storage. Care was also taken to ensure that SSDE ran sufficiently fast to make diagram changes appear instantaneous and system transfers between disk and memory cause negligible delays.

6.7 Summary.

It was indeed satisfying to see an initial idea evolve into a development environment such as SSDE. In converting the prototype to the final product, several changes were made and additional features added. In particular, data representation in memory was improved, the database designs were altered and diagram presentation was refined. Considering the favorable evaluation given, it was worthwhile constructing SSDE.

CHAPTER 7:

THE MENU SYSTEM.

7.1 The Menu System.

The following diagram provides an overview of the various menus available and their interaction:

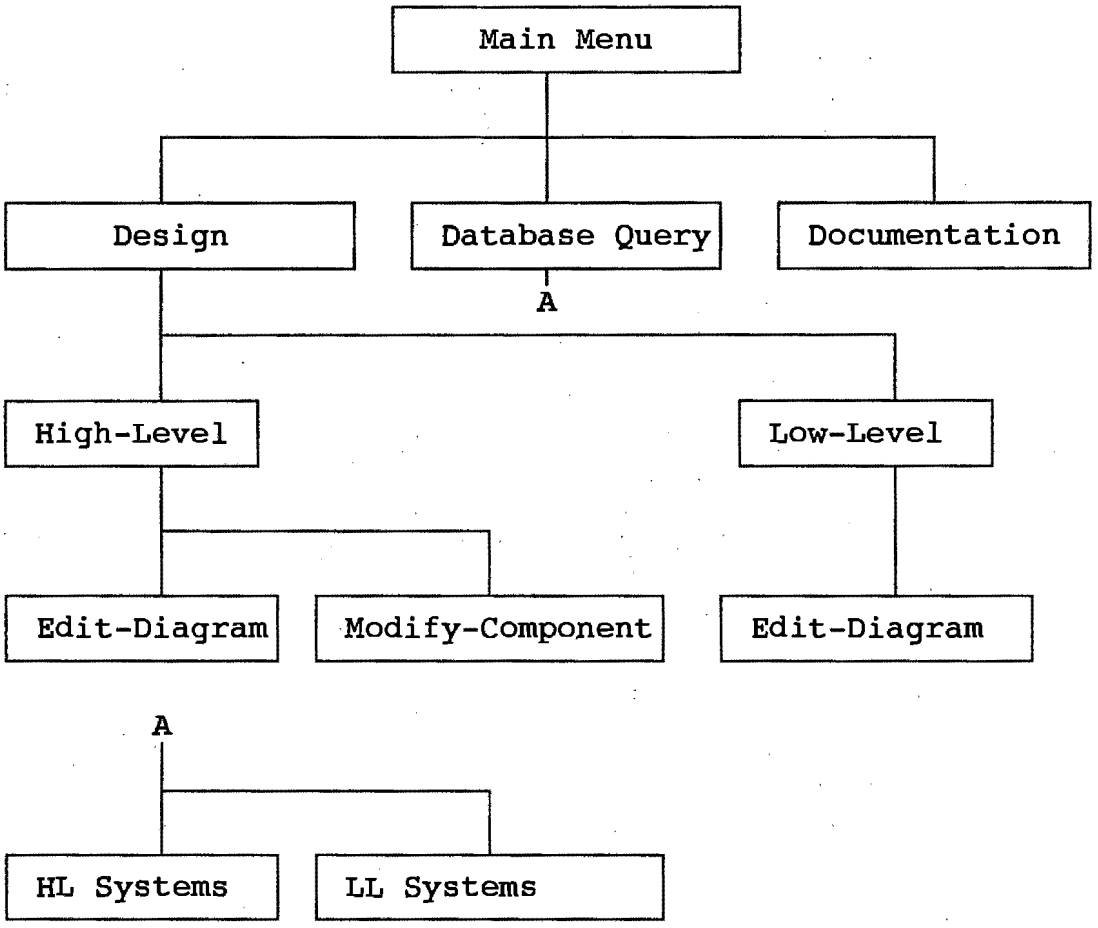


Figure 7.1 The Menu System

This chapter describes all but the bottom level menus, in order to outline the ways in which the designer can move between the different SSDE subsystems. As such it also provides an overview of the environment and its major components.

7.1.1 The Main Menu.

The user is presented with the three major options (facilities) provided by SSDE from which he can make his choice depending on what activity he would like to do next. The main menu presented looks as follows:

```
        1.  Design
        2.  Database
        3.  Documentation
        4.  Exit

H=help
```

Figure 7.2 The Main Menu Options.

This menu permits the choice between the three major system activities and help screens.

7.1.2 The Design Facility Sub-Menu.

The following sub-menu is presented when the design option has been chosen from the main menu:

```
  1. High-Level Design
  2. Low-Level Design
  3. Verify HL-LL Link
  4. Generate LL code
  5. Return - Main Menu

H=help
```

Figure 7.3 Design Sub-Menu.

Option: High-Level Design and Low-Level Design.

Once the designer has identified the system of interest, these lead to the high-level and low-level design facilities respectively, where a new design can be started or an existing one modified. With high-level design a subtree can also be created, with some existing high-level component as its root. Figures 7.4 and 7.5 show the high-level and low-level design screens which are described in later chapters.

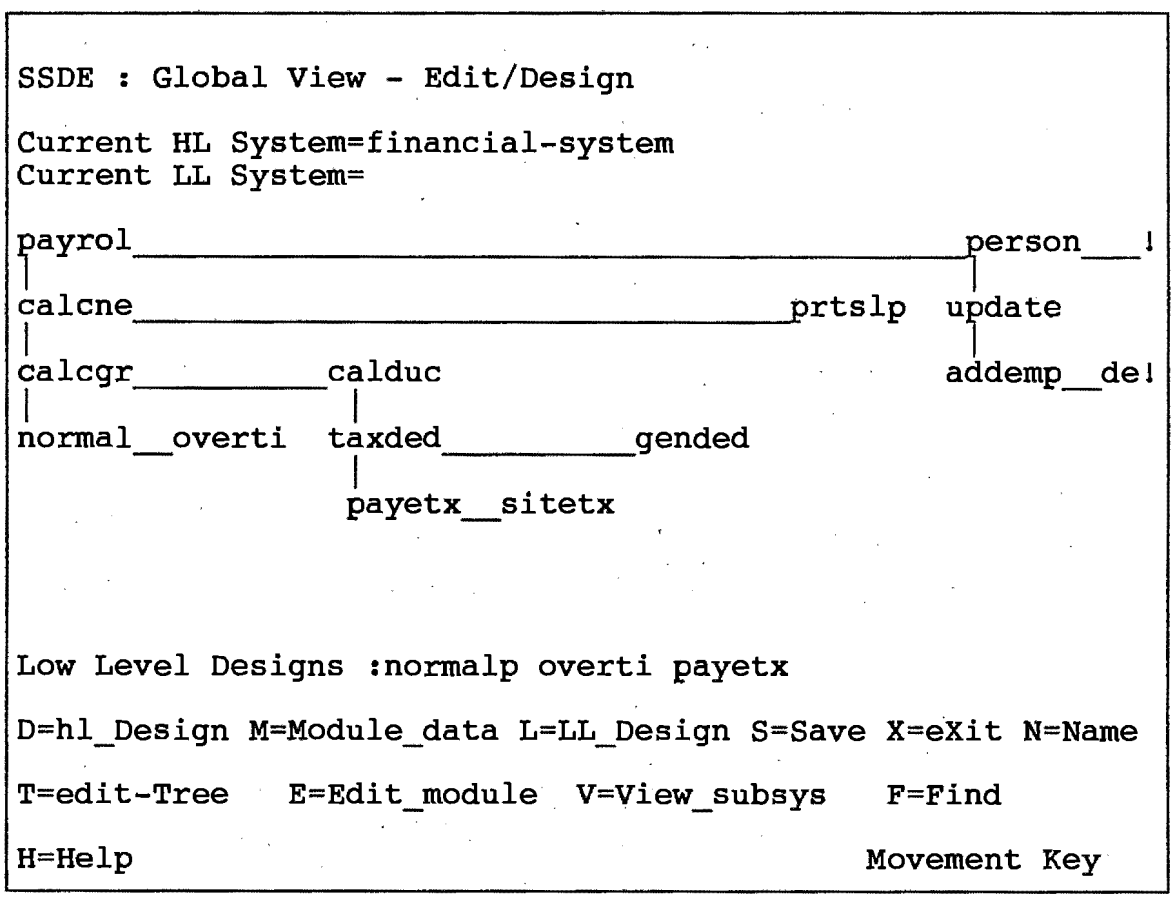


Figure 7.4 Global View and Menu.

```
HIPO = add-invoice Module# 6-1 LL Name = stock-add-invoice Edit#

+-------------------------------------------------------+---+
| open database file "invoice.dat"                      | 1 |
+-------------------------------------------------------+---+
| open index file "invoice.ndx"                         | 2 |
+-------------------------------------------------------+---+
|    get invoice-type%                                  | 3 |
+-------------------------------------------------------+---+
| until invoice-type% =1 or 2 or 3                      | 4 |
+-------------------------------------------------------+---+
| **update-database                                     | 5 |
+-------------------------------------------------------+---+

Q=seQ I=If R=Rept L=Loop O=clOseUN Z=End_construct C=Case
F=testFlg T=Txt-Ed E=EditT D=reDrw V=Vars N=Name S=Save/X X=Exit
H=help
```

Figure 7.5 The Low-Level Design and Menu Screen.

Option: Verify HL - LL Link.

This option will perform a check between the variables defined in a particular high-level component and the variables defined in the corresponding low-level variable list. This can be repeated for any number of components. Verification cannot be incorporated automatically because incremental and partial design methodologies are permitted. Thus, repetitive and annoying errors of omission, which could distract the analyst's design thoughts, are not generated. Hence stand-alone verification is provided here.

Option: Generate Low-Level Code.

This will generate skeleton Pascal code for the low-level design currently in memory.

Option: Help.

Provides brief and/or detailed information about the options on the present screen. This option exists in every menu.

7.2 The Database Query / Update Sub-Menu.

This menu is displayed when the database option has been chosen from the main menu

```
  1. High-Level Systems
  2. Low-Level Systems
  3. Return - Main Menu

H=help
```

Figure 7.6 Database Sub-Menu.

There are many database activities which can be performed for both low-level and high-level designs. It was for this reason that the database manipulation and query facilities are grouped accordingly.

7.3 The High-Level Database Sub-Menu Options.

When "High-Level Systems" from the above menu is chosen the following menu appears:

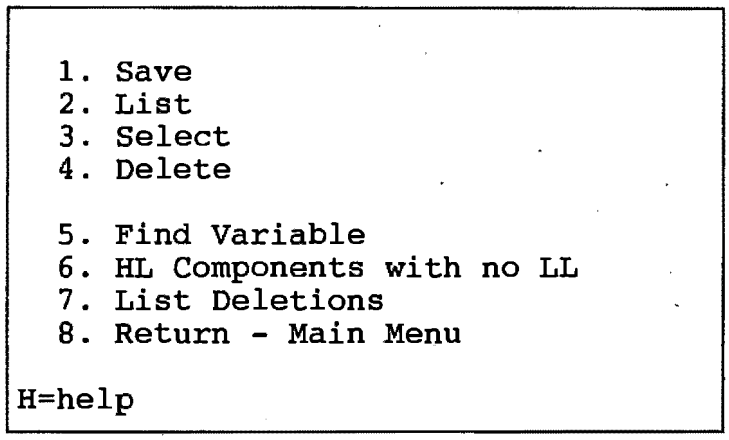


Figure 7.7 The HL Database Sub-Menu.

Option: Save.

This will result in up to date database records being made for each high-level functional component. The low-level save has been kept separate from this option to provide increased flexibility.

Option: List.

This lists all the high-level systems from the database.

Option: Select.

Selecting an existing system from the database to continue its design. This avoids the need to navigate up and down the menu system.

Option: Delete.

The High-Level definition of a complete system, together with its associated low-level definitions, is marked as deleted.

Option: Find Variable.

All the components can be identified which have referenced a specified variable, whether as input or output.

Option: HL Components with no LL.

List all high-level designs which have no dependents and for which no corresponding low-level design has been completed.

Option: List Deletions.

Lists previously deleted systems eg. so that they may be recovered.

To recover any deleted system, it can be retrieved and then renamed.

7.4 The Low-Level Database Sub-Menu.

```
  1. Save
  2. List
  3. Select
  4. Delete

  5. List Variables
  6. List Abstractions
  7. List Deletions
  8. Return - Main Menu

H=help
```

Figure 7.8 The LL Database Sub-Menu.

Option: Save, List, Select, Delete.

These are analogous to their high-level counterparts.

Option: List Variables.

List all variables, their types and structure, for a particular low-level design.

Option: List Abstractions.

Locate all abstract entries in a particular low-level design.

Option: List Deletions.

Lists previously deleted low-level systems eg. so that they may be recovered.

To recover any deleted system, it can be retrieved and then renamed.

7.5 The Documentation Sub-Menu.

This menu is displayed when the documentation option is chosen from the main menu:

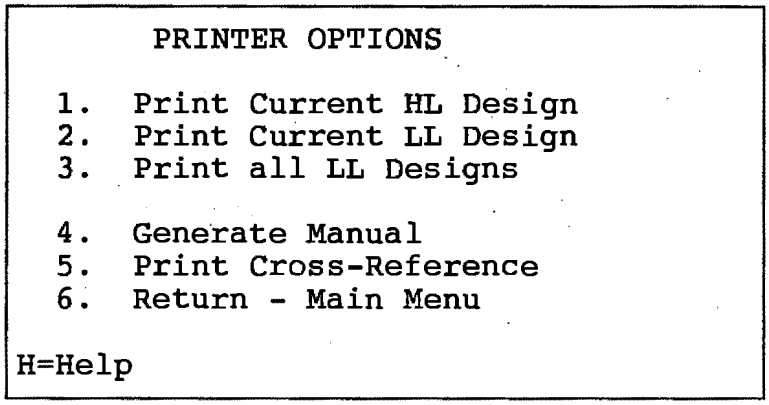


Figure 7.9 Printer Sub-Menu.

Option: Print HL Design and Print LL Design.

This prints out a hardcopy of the high-level (or low-level) design together with all associated information. It is possible to generate partial documentation by requesting only specific low-level diagrams that are of interest at the time.

Option: Print all LL Designs.

This prints out all the low-level designs of the current system.

Option: Generate Manual.

This will generate a skeleton system manual write-up from the high-level definition of the system.

Option: Print Cross-Reference.

A particular variable may be entered and all high-level functional components which have a reference to that variable are listed.

```
SSDE : Global View

System=financial-system

payrol_____________________________________________person___!
|                                                    |
calnet_____________________________________prtslp    update
|                                                    |
calgro_____________calduc                            addemp__d!
|                   |
normlp__overti   taxded_____________gended
                    |
                  payetx__sitetx


 Low Level Designs :normalp overti payetx

 Components which have been designed:

  payrol> Module name=payroll calculation
                Input=emp_record : pay_rec
               Output=gross_sal : deduct : net_sal
              Comment=payroll calculation for salried staff
              process=get employee record
                      calc gross and net salary

  person> Module name=personnel system
                Input=emp_number : update_rec
               Output=emp_record
              Comment=Update personnel records
              process=get employee number & update detail
                      update the employye file
                      print out new employee record


.  Similar prints for the other modules on the Global View  .
```

Figure 7.10 High-Level Printout Documentation.

```
System=inventory-reorder
Component=inventory-reorder-add-invoice

open database file "invoice.dat"
open index file "invoice.ndx"
    get invoice-type%
 until invoice-type% =1 or 2 or 3
**s-update-database
```

Figure 7.11 Low-Level Printout Documentation.

7.6 A Typical Development Process.

Although different designers will use varying design approaches, the following steps give a general example of a design strategy:

* define a data structure using the high-level hierarchy.
* create the functional components using the high-level design. A fast design can be done specifying only the names of the components. The detail (e.g variables, input/output, process steps and general comment) of each component can be added later.
* print the high-level.
* when difficulty is encountered during the high-level design, then the process steps can be stated or a low-level design done with abstract entries. The clarity obtained in this way will assist in the continuation of the high-level design.

* low-level designs for functional components can be designed.
* print the low-level and symbol data.
* verify high- to low-level linkages.
* generate code.
* test code.
* print final documentation, manual and cross-references.
* the database query facility can be used during maintenance to find the relevant part. Low-level designs can be copied and changes made.

7.7 Summary.

SSDE provides the designer with a functionalized menu-system which distinctly groups all similar activities together, making it easy to know where you are at any point. The menu system also allows fast navigation between the integrated parts. A flexible design methodology is permitted allowing the designer to move freely from one menu to another without loss of design information. The chapter has outlined only the major SSDE facilities. The other capabilities such as software re-use and design modification will be discussed in succeeding chapters.

CHAPTER 8:

DATABASE UTILIZATION.

8.1 Introduction.

Since a large amount of information and data needs to be recorded in order to select, recall and display previously designed systems (low-level and high-level) for maintenance, documentation and other purposes it was necessary to create on-line databases. This decomposition into specific databases was essential in order to increase the system's efficiency in terms of access times and secondary storage utilization. In addition to the four databases associated with capturing the design, another two databases exist which provides the brief and detailed help information respectively.

8.2 Information about the Databases used.

The five databases which capture the design are divided into two principal parts: those which capture the high-level data (HLDB, PROCESSDB and COMMDB) and those which capture the low-level data (LLDB and VARDB).

8.2.1 Databases: High-Level Design.

The high-level data is split into three parts (HLDB, PROCESSDB and COMMDB) in order to save secondary storage space and to improve access times. For each module or component in a high-level hierarchy, a record will be created in HLDB. Process

steps which can form part of a high-level component, are stored in PROCESSDB. A separate database has been used since these entries can be optional and because the low-level design fully addresses detail processing definition.
An optional general comment can be associated with each module. Since it can be expected that the analyst will use meaningful module names of a reasonable length, the use of this general comment facility in practice will probably be fairly limited. To accommodate this facility COMMDB is used. This saves space in HLDB, as there will be some functional components which have no accompanying comments.

8.2.2 Databases: Low-Level Design.

There are also two distinct parts associated with the low-level data, viz, the actual low-level structures and the symbol table data. Two databases were therefore used for the low-level design. LLDB stores the design data (structure type and accompanying text) and VARDB stores the symbol table data (variable name and type).

8.3 The Databases which contain the High-Level Information.

8.3.1 HLDB.

The number of records created for a particular system will be equal to the number of components in its hierarchy. SSDE will create one additional status record for each system that will indicate the maximum level this system has progressed to and

the maximum number of modules on any one level. This status record greatly improves the speed with which all the records for a system can be retrieved. The two items kept in the status record define the outer parameters for the search.

The definition of this database record is as follows:

```
hldbrec = record
            syskey : string[50];
            modname : string[50];
            shortname :string[6];
            modinput : string[50];
            modoutput : string[50];
            modnumber : integer;
            modparent : integer;
            modchild : integer;
          end;
```

This contains all the relevant data associated with one module of a high-level hierarchy.

The information in this record is as follows:

* syskey - which holds the system name with the module number e.g "inventory-reorder34". Since the module number will be unique, this is also the key.

* modname - the module name given to this component, e.g invoice-file-process.

* shortname - a six character name for this component

* modinput - input which will be made available to this module by its parent.

* modoutput - output which will be passed by this module back to its parent.

* modnumber - the module number (level and position in hierarchy) of this module, e.g 34 is the 4th module on the 3rd level

* modparent - the module number of its parent, e.g 23

* modchild - the child number of this module with respect to its parent, e.g 2 if this is a second child

A status record has the same record format as above. In the status record "modnum" has been used for the maximum level number of this system. The field "modparent" holds the maximum number of modules in any level.

8.3.2 COMMDB.

A number of records can be created here for each system designed. The number of records for each system is determined by the number of module comments entered and their size. In this way minimal disk space is used.

The definition of this database record is as follows:

```
commdbrec = record
            syskey : string[50];
            linenum :integer;
            modcomment : string[50];
          end;
```

This database contains the general comment with may be associated with a module.

The information in this database is as follows:

* syskey - which holds the system name with the module number to which this general comment belongs, e.g "inventory-reorder34". This matches the key in HLDB.

* linenum - an integer which indicates the line number of this comment. This allows for any number of lines per module.

* modcomment - the text of the general comment.

8.3.3 PROCESSDB.

A number of records can be created here for each component in a system.

The definition of this database record is as follows:

```
processrec = record
           modname : string[50];
           linenum :integer;
           modstep : string[30];
         end;
```

This database contains the process steps which may be associated with a module.

The information in this database is as follows:
(modname and linenum constitute the key)

* modname - which holds the system name with the module number to which this step belongs, e.g "inventory-reorder34".

* linenum - an integer which indicates the line number of this step. This allows for any number of lines per module.

* modstep - a brief (abstract) process statement

8.4 The Databases which contain the Low-Level Information.

8.4.1 LLDB.

One database record is kept for each low-level construct. Generally a number of records will be created here for each low-level diagram.

The definition of this database record is as follows:

```
lldbrec = record
          nskey : string[70];
          nsinstruction :string[2];
          nstext : string[50]
         end;
```

This database captures the information received during the design of the low-level system.

The system name plus the module name together with an integer number forms the key for a typical database record. Primary low-level designs have an "*" appended to their name and secondary diagrams have no such attachment.

The information in this database is as follows:

* nskey - which holds the actual name of this low-level design (the key)

* nsinstruction - this field holds the code giving the type of construct this record represents (eg. sequence, IF, etc)

* nstext - holds the actual text associated with this structure just as the analyst entered it when he created the construct

SSDE will create one additional status record. It uses "nsinstruction" to indicate the number of records which has been stored for this low-level component. In the status record "nstext" stores the test flag value of the low-level component (indicating if this low-level component's code has been tested). The status record has the same format as the other

records of this database. The status record will greatly improve the speed with which all the records for a system can be retrieved since it indicates exactly how many records to search for.

8.4.2 VARDB.

This data base contains all the variable names and their types, for each low-level design.
The key here is the same as the key for LLDB.

The definition of this database record is as follows:

```
vardbrec = record
           nskey : string[70];
           symbolname :string[50];
           symbolfld  :string[50];
           symboltype : string[1];
           arrayvar   :string[1];
           arrayfld   :string[1];
          end;
```

The information in this database is as follows:

* nskey - which holds the actual name of this low-level design plus a unique integer which is incremented for each record (the key)
* symbolname - the name of a variable in the low-level design identified by "nskey"
* symbolfld - the field name (if variable is a record)
* symboltype - the type of this variable (or field)
* arrayvar - indicates whether the variable in "symbolname" is an array variable

* arrayfld - indicates whether the field in "symbolfld" is an array

One database record is kept for (each field of) each variable in every low-level component. Generally a number of records will be created here for each low-level design. SSDE will create one additional status record that will indicate the number of records which have been stored for this particular low-level component.

8.5 The Databases which contain the Help Information.

8.5.1 BHDB.

This data base contains a single line of help about a particular option on a menu. The key here is constructed from a name given to the menu plus the option in question.

The definition of this database record is as follows:

```
bhdbrec = record
            briefkey : string[20];
            briefhelp :string[50];
          end;
```

The information in this database is as follows:

* briefkey - holds the actual name of this help message.
* briefhelp - the text of this help message.

8.5.2 DHDB.

This data base contains help text about a particular option on a menu. The key here is constructed from a name given to the menu plus the option in question.

The definition of this database record is as follows:

```
detailhelpc= record
             detailkey : string[20];
             linenum : integer;
             detailmsg :string[50];
           end;
```

The information in this database is as follows:

* detailkey - holds the actual name of this help message. Same as in BHDB.
* linenum - an integer which indicates the line number of this comment. This allows the detailed help text length to vary.
* detailmsg - one line of this help message.

8.6 Summary.

Five databases have been used to capture the complete system design. The high-level design is contained mainly in HLDB and PROCESSDB with COMMDB being used for storing an optional general comment for a particular high-level component. The complete low-level design is stored in the remaining two databases, viz. LLDB and VARDB. Additional status records have been written to databases in order to further speed up access

times by restricting the scope of the database search effort. Two further databases, BHDB and DHDB contain the help information. The databases have been designed to make the processing task easier, improve access times, reduce the searching activity and to use minimum secondary storage. This was accomplished by creating separate database records instead of repeating groups, by using status records, by placing optional information in a separate database and by ensuring that the system is in 3rd normal form.

CHAPTER 9:

HIGH-LEVEL DESIGN.

9.1 Introduction.

The purpose of the high-level design facility is to allow the analyst to define the high-level modules or components into which a system is decomposed. This hierarchical definition also reveals the relationship between these components. This chapter describes how the high-level definition is done by the analyst and then explains how SSDE accomplishes these tasks.

9.2 High-Level System Design.

The high-level design is an overview of the complete system being constructed. Using the HIPO-Structure chart combination the system can be represented as a hierarchy of modules. A module can be considered the basic building block of a system or program. The modules which address the high-level system tasks are normally placed at the upper levels of the hierarchical structure and the modules representing more complex logic appear at the lower end of the hierarchy. In SSDE, the detail concerning the low-level logic is designed in terms of the low-level diagramming technique described in the next chapter.

The high-level design paradigm in this system is therefore used to reveal the overall structure of the system by showing the system modules and their inter-relationships; not the actual low-level logic. Each functional component ("box") in the

visual representation can represent a system, subsystem, program or program module. Inside the box the designer can enter text to give information about that particular component. Not much detailed design is required here, since the object of the design phase at this point is to show the major components and their relationships with each other.

9.3 Designing the HIPO / Structure Chart Components.

9.3.1 Decomposition.

Designing a system involves the process of decomposition. During the decomposition phase, the analyst specifies the functional components (modules) and further also specifies the input and output variables of each. This process is very much an informal procedure with the analyst assigning names to the various modules in a natural language as suggested by Archibald et al [:1983 p180].

9.3.2 Global and Local High-Level View.

Two views are provided as part of the high-level design activity. The global view shows the system hierarchy with as many components as possible on the screen. During global fast high-level design (discussed later) only the module name needs to be supplied when a component is created.
The local view shows only the parent component and surrounding dependents. However, the full details associated with these are also shown, i.e the input/outputs and full names. This permits

a more detailed design of a high-level component in the context of modules in close proximity.

9.3.3 Local Design of Functional Components.

In order to create a high-level functional component, the analyst need only specify the parent of the new module. The environment will determine the local neighbours of the module to be designed. The module name of the parent and its inputs and outputs are displayed as well as that of the two brothers immediately preceding it. This facility is limited to showing one parent and (at most) 3 children for the following reasons. Firstly new children are added on the extreme right. Secondly this enables SSDE to display clearly as much information as possible about these modules within the confines of a limited screen size. This avoids making the data on the screen too voluminous. To view the complete system in totality, the global view can be used.

9.3.4 Defining the Components.

When control is transferred between two modules, data is usually transferred as well. Input associated with a particular module means data which will be transferred to it by the invoking module. Output associated with a particular module means data which it will produce and then return to its invoking module.

After the name of a new component has been accepted, its inputs and outputs need to be entered. Multiple inputs and outputs may be specified. If a variable name is invalid, due to the use of illegal characters, it is rejected and the analyst has to correct his entry. Instead of using arrows to indicate the data transfers, SSDE puts the data inside the "box". This avoids the diagram from becoming cluttered with information. This same approach is also used by the STRADIS/DRAW Software which draws and edits structure charts [STRADIS/DRAW].
In keeping with the HIPO methodology, a number of process steps may also be entered.
A general comment associated with this functional component may also be entered.

9.3.5 Defining a Data Structure.

The high-level design facility can also be used to define a data structure. Since structure charts are a form of decomposition, the data structure can be represented in terms of a hierarchy. The inter-relationships between the elements of the data structure can also be illustrated. Thus the high-level design facility can be used to document the data structures used in a system.

9.4 The Local Neighbour Design View.

This screen looks as follows:

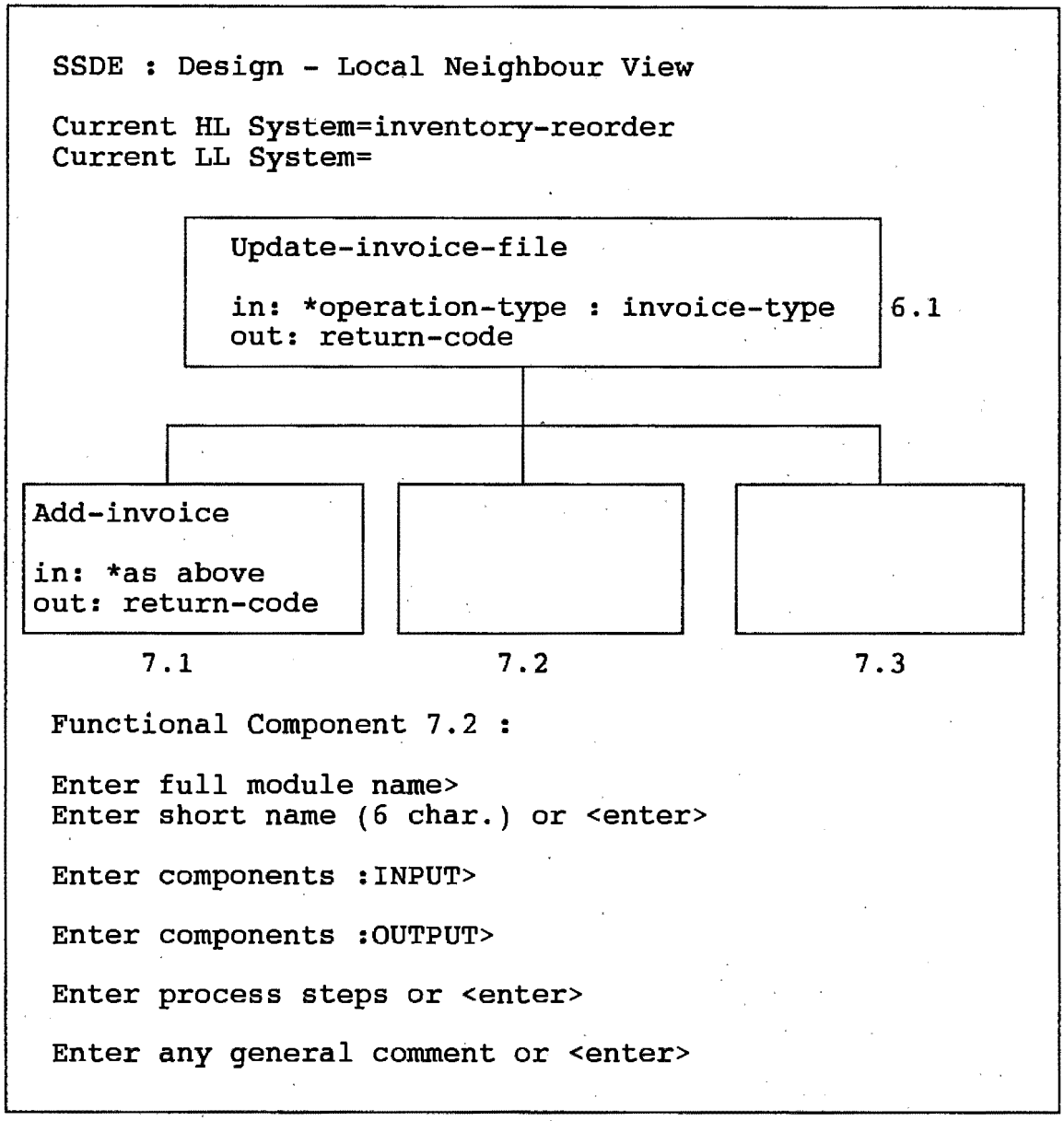


Figure 9.1 Detail High-Level Component Definition.

9.5 The Global View of the High-Level Design.

When entering the high-level design sub-system (or leaving the local neighbour design view), the global view of the high-level is presented:

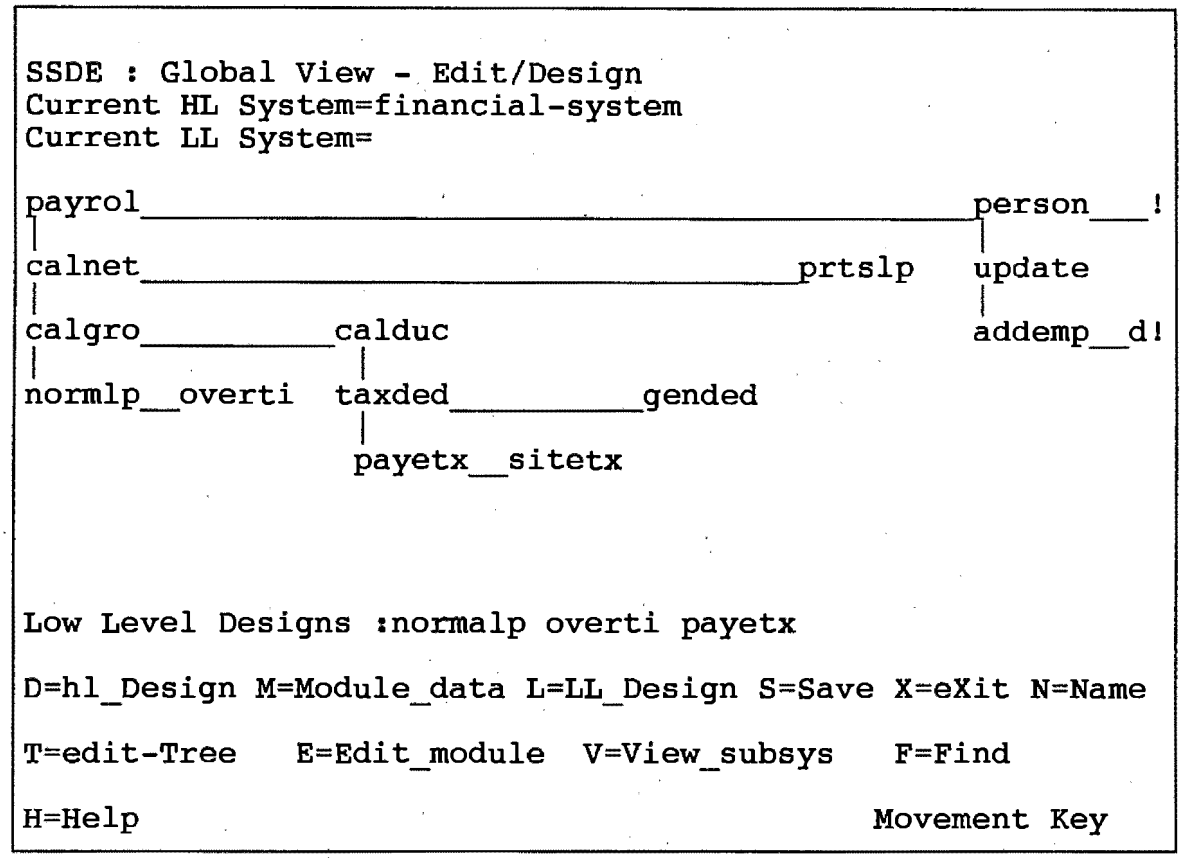


Figure 9.2 High-Level Design Global View.

This hierarchical view is provided to enable the designer to view his design done so far and to assist him in designing the next part. This global view is also presented to the designer at this point, so that he can choose from a number of other options which he might want to perform. The design methodology is therefore not absolutely rigid, but allows the designer to perform incremental design if he so wishes. With a creative task such as this, it is important to have as much freedom and flexibility as possible and not be hampered by unnecessary and distracting restrictions.

The global view also presents the analyst with a menu, and the current system name, current low-level module name (if any) and a display of the low-level modules which have been designed up to this point (if any).

9.6 The Sub-Menu on the Global View of the High-Level Design.

Option: D. hl_Design.

This option allows component definition in terms of the detailed local neighbour view.

Option: M. Module_data.

All the relevant data about any module can be extracted: full name, number, inputs, outputs, process steps and comment. This option assists the designer in determining where he should continue his high-level design. This is useful if he has forgotten high-level design information or if he needs confirmation of a relationship of a high-level design he wishes to enter next via option A. Also, if the designer wishes to do a low-level design at this point and he wants to determine the particular high-level module, he can do so using this option.

Option: L. LL_Design.

This transfers control to the low-level definition facility. The environment requests the module "handle" (6 character name) for which he wishes to complete a low-level design.

Option: S. Save.

The high-level definition is saved in the relevant databases.

Option: X. eXit.

The main menu is displayed.

Option: N. Name.

The system name can be changed. If the user wants to construct a system with a very similar structure to the current one then he can use this option to change the system name. He can then simply save this high-level design and he would then have created a new system in the database.

Option: T. edit-Tree.

The global view can be edited or extended for fast design. The analyst is presented with a further small menu which allows him to delete, insert, copy or move functional components.

Option: H. Help.

Provides brief and detailed help screens.

Option: E. Edit_module.

The data items which describe a functional component (module) may be modified.

Option: V. View_subsys.

A subtree of the complete hierarchy can be displayed.

Option: F. Find.

A search process that will locate a module in the hierarchy, given its full name or a portion thereof. For example "calc*" will list all modules starting with "calc". Since the

high-level global view only displays handles, not full names, this is a useful option.

9.7 Fast High-Level Conceptual Definition.

It is important to note that an entire high-level system can be quickly constructed using option G. This option can be used to by-pass option A on this same menu. When using this option to construct a system, the difference is that the detail of the local surrounding modules are not given and the input / output and comment parts are not requested. The design is then done by adding / moving handles in the global context of the system. The outstanding information can of course be entered later using option I.
High-level functional components can be quickly designed in this way, without specifying their related input/output and comment parts. The same global view of the system with its menu is retained throughout this process and the global system view is constantly updated as this facility is used.

9.8 How the system stores information.

9.8.1 Dynamic Data Structures.

Dynamic data structures are used to capture the high-level design. Dynamic data structures can expand and contract as the program executes, unlike arrays which have a fixed number of storage locations. As the analyst is busy with the design activity, the system definition "grows". Using dynamic data

structures, it is possible to capture this growth without wastage of internal memory. The following definition realizes this:

```
type
  ptr = ^rec;
  listpointer1 = ^listnode1
  listpointer2 = ^listnode2

  listnode1 = record
               processline :string[30];
               link :listpointer1;
              end;
  listnode2 = record
               commentline :string[80];
               link :listpointer2;
              end;

  rec = record
        modname : string[50];    {name of module}
        shortname : string[6];   {6 char name}
        modnumber : integer;     {hierarchy number}
        modinput : string[50];   {input variables}
        modoutput : string[50]; {output variables}
        modprocess : listpointer1; {process steps}
        modcomment : listpointer2; {general com}
        modparent : integer;     {parent number}
        modchild : integer;      {child number}
  end;

  var
        tracepic : array[1..40,1..40] of ptr;
```

Figure 9.3 High-Level Record Component Definition.

The reason for using an array of pointers was because this facilities the global and local display of the high-level design. The high-level hierarchy can be formed with ease using this data structure. It can be seen from the above data structure that the maximum hierarchical structure is a 40x40 tree. It is highly unlikely that such a large single tree will ever be developed since it will be contrary to the philosophy of structured techniques. If however, such a large system needs

to be designed, the decomposition methodology can be used. Using decomposition, a number of sub-trees can be designed and together they constitute the complete system.

9.8.2 Capturing the Design.

Each module in the high-level hierarchy has a number associated with it. The number has two parts: the first part indicates the level and the second part the position within the level. On level one (at the top) there is only a single root module. Design decomposition starts at level 2. When a new hierarchical component is being created by the designer, memory is allocated to contain its definition. A data structure, as defined above, will be created in main memory to hold the component's data. Example: if module number "21" is being created, then the following is executed:

```
new(p);
tracepic[2,1]:=p;
```

The module name is captured as follows:

```
p^.modname:=module_name_entered;
```

In a similar way the other information is obtained. The analyst need only specify the parent number. SSDE will determine the module number using a 1-dimensional array "maxnum" that indicates the number of modules that have so far been designed on each level. This is used often in SSDE, so keeping them in such an array avoids unnecessary re-calculation of these values. As the designer edits the high-level design, the above array of pointers is updated, as is "maxnum", to reflect the new state of the system.

9.9 Drawing the Tree.

It is possible that the complete design cannot fit onto one screen. In this case the global view can be considered a window on (a part of) the complete chart. The ability to move the display window is essential. With this in mind, the following technique was employed to accomplish the display process. Every line seen in the global view is the contents of an alphanumeric variable. The display is first formed in these variables, then the contents of these variables are displayed. This is very useful when the display has to be scrolled in a particular direction.

9.9.1 Constructing the Display.

The diagram is drawn as follows: the "short name" ("handle") of all high-level functional components which have been designed to date will be displayed, subject to screen size constraints. The field "maxlevel" will indicate how many levels there are; the fields "maxnum[i]" (i:=1 to maxlevel) are summed to calculate how many modules to display. The actual display is formed by searching through the elements indicated by the pointers in "tracepic". Module names are truncated to six characters if no handle exists. First the top left-most module name is placed in the variable which will display the first row. Then all its dependents, i.e its children, grand-children, great-grand children, etc, are all placed in the display variables which will display the 2nd, 3rd, etc. rows. Then the

next top-most module is obtained, placed in the variable for the first display, and the procedure is repeated until all the modules are placed. Since every record has a "modparent" field, it can be determined to what parent any module belongs. The module drawing is therefore formed from top left to bottom right. The position that a module occupies in the display variable is determined based on the position of its parent.

9.9.2 Inserting Lines into the Display.

Lines are also drawn which assist in the viewing of the hierarchy. This makes it easier to see which modules belong to a particular parent. The line drawing is accomplished as follows: before the module names are placed, the display variables are filled with lines. Next the handles are filled in as described above. By keeping a record of the various display points of these handles, lines which do not link components can be erased from the display variables. A parent-child link is drawn whenever a "modchild=1" situation is encountered. Only once all the modules are placed and the unnecessary lines removed, can the contents of the display variables be placed on the screen. Below are two examples of how the display and hierarchy is formed within the display variables: (here the top-left module and all its dependents have been placed and some lines which at this stage are already unnecessary have been removed and replaced with "+" (used instead of space here for clarity.)

display variable 1:

payrol_____

display variable 2:

calnet_____

diplay variable 3:

calgro_____calduc_____

display variable 4:

normlp__overti++taxded_____gended_____

display variable 5:

++++++++++++++++++payetx__sitetx_____

(+ indicates a blank character will be printed)

Figure 9.4 Global View - Display Process.

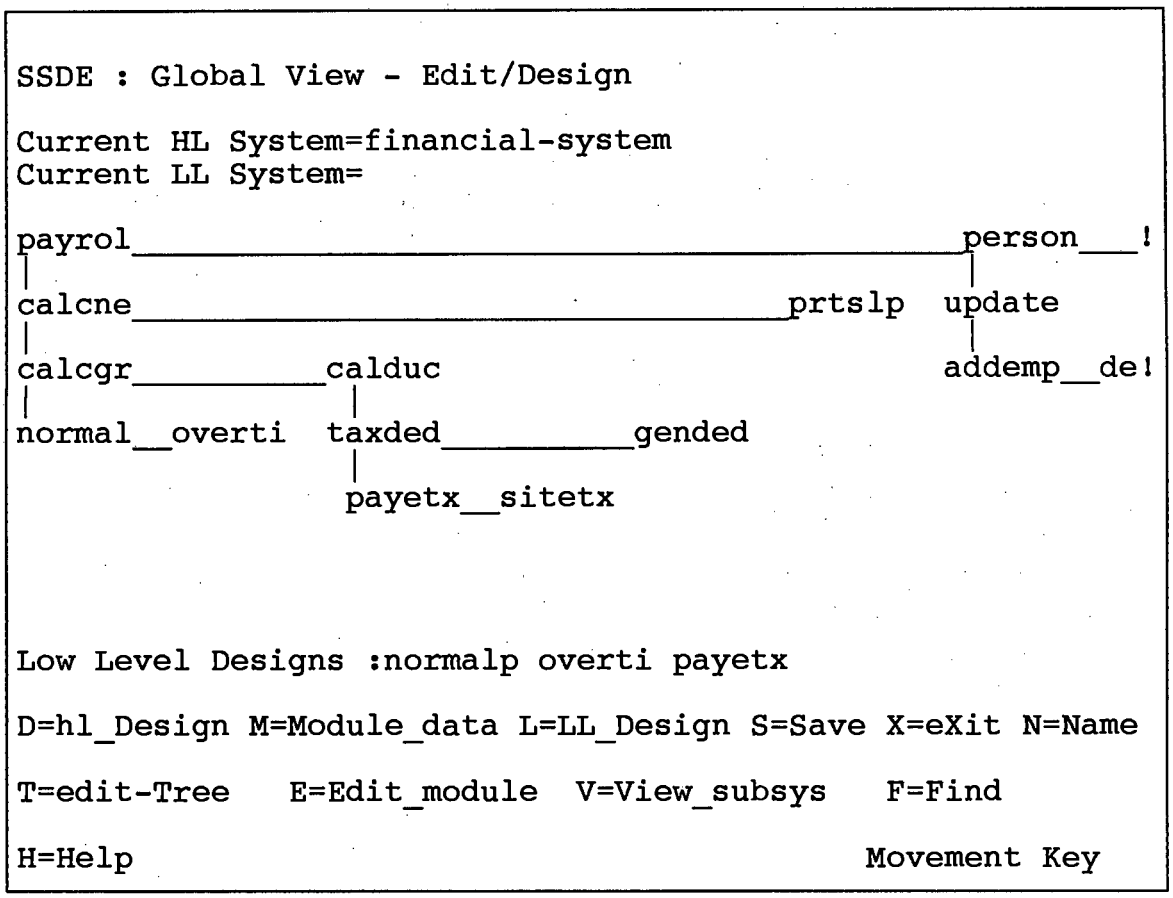


Figure 9.5 Global View - Display Process.

After all handles have been placed and unnecessary display lines eliminated, the contents of the display variables are shown on the global view of the system. Finally figure 9.5 is the screen seen by the analyst.

9.10 Shifting the Global Display.

Since the global display has been placed in display variables, the moving of the display is greatly simplified. If the "right-movement" key is pressed once, then the fixed portion of the display variables which is shown, is shifted one character right. The display is thus moved since a portion, one character to the right, is now shown. This is similar to the "window" concept in for example a spreadsheet. The display is moved in a

similar way when the other movement keys are pressed.

9.11 Implementation of the Global View Menu Options.

9.11.1 Edit-Tree.

(a) Delete.

To delete a leaf component the pointers in "tracepic" are updated (storage space is freed) and also the "modchild" fields of brothers are altered. Appropriate module numbers are automatically updated, so that module numbers strictly increase within a level. The high-level global view is re-displayed to reflect the deletion. Similarly if the analyst wishes to delete a complete subsystem, then the component number which heads the sub-system, together with its dependents, are all deleted. Module numbers are automatically updated and the view is re-displayed. Deletion of a high-level component which has a corresponding low-level design, will result in the low-level design also being deleted.

(b) Insert.

The module name after which the insert must take place is required. The pointers in "tracepic" are updated and new storage is allocated. The appropriate module numbers ("modnumber") are automatically increased by one, so that module numbers strictly increase within a level. The high-level global view is re-displayed to reflect the insertion.

(c) Copy.

The "handle" of the component to be copied is requested. Otherwise, the copy is processed in a similar way to insert.

(d) Move.

The move operation combines aspects from the copy and delete functions.

9.11.2 Displaying a Sub-System.

The sub-system is displayed in a similar manner to the entire system, except only the dependents of the particular component are presented. The level below that of the sub-system root is searched to find its dependents. All components have a parent field and dependents can therefore be identified. These dependents now become the new parents and one level lower is searched to see if they have any dependents. And so the process continues until the entire sub-system is obtained. The actual display is done in the usual way.

9.12 Implementation of the High-Level Database Sub-Menu Options.

This menu looks as follows:

```
1. Save
2. List
3. Select
4. Delete

5. Find Variable
6. HL Components with no LL
7. List Deletions
8. Return - Main Menu

H=help
```

Figure 9.6 The HL Database Sub-Menu.

9.12.1 Deleting a High-Level System.

All high- and low-level systems which are "deleted", are not actually physically discarded. The names of components which have been entered for deletion, have been marked by surrounding the names with a "{" and a "}" character. This ensures that a history of the design is automatically maintained which is very useful to avoid repeating design errors.
SSDE will also search the low-level databases to mark the system's low-level modules as deleted. All the low-level module names are preceded by the system name for ease of identification.

9.12.2 Finding a Variable's References.

This facility involves identifying all high-level functional components from a particular system which reference a specific variable in their input or output specifications. The status record for the system is located. After this the individual records which represent its functional components are brought into main storage in turn and their input and output is

checked. If the variable is present then the component's name and position is displayed. Only a single record's storage space is required since storage is dynamically allocated once and re-used for each functional component. The dynamic allocation here is the same format as is used when a high-level functional component is created.

9.12.3 High-Levels with no Low-Level Design.

This allows all high-level components which have no further decomposition ("tip-nodes"), and for which no corresponding low-level exists, to be listed. The status record is first searched for. The modules in the hierarchy are read from the database into main memory. The lowest level components are first read in one by one. If their names do not contain an "*" character then they are listed (this indicates no corresponding low-level design.) The parent numbers of these lowest level modules are noted as they (the parents) cannot be "tip-nodes". The next lowest level is then searched. Modules which are in the parent list are excluded in the search since they are not tip-nodes. The other module names are checked and those that do not have an "*" are listed. The parents of these that are searched are added to the list of parent numbers. The process is then repeated until the highest level is reached. Only a single record's storage space is required since storage is dynamically allocated only once and re-used for each functional component. The dynamic allocation here is the same format as is used when a high-level functional component is created.

9.13 Summary.

The high-level design facility provides the analyst with the necessary tools to complete the hierarchical definition of the system. This definition shows the inter-relationships of the functional components. The design can be accomplished in detail using the local neighbour view. A fast, but less detailed design, is also possible in the global context (view) of the system.

CHAPTER 10:

LOW-LEVEL DESIGN.

10.1 Introduction.

The low-level program design provides the detailed logic necessary to explain how the tasks of the system will actually be accomplished. This low-level design is done independently of any particular programming language. Each structure in the low-level design paradigm can be transformed into one or more statements in a programming language.

10.2 Low-Level Design Screen.

The following example shows the low-level design screen and menu together with a few low-level structures which have been entered:

```
HIPO = add-invoice Module# 6-1 LL Name = stock-add-invoice Edit#

+-------------------------------------------------------------+---+
| open database file "invoice.dat"                            | 1 |
+-------------------------------------------------------------+---+
| open index file "invoice.ndx"                               | 2 |
+-------------------------------------------------------------+---+
|    get invoice-type%                                        | 3 |
+-------------------------------------------------------------+---+
| until invoice-type% =1 or 2 or 3                            | 4 |
+-------------------------------------------------------------+---+
| **update-database                                           | 5 |
+-------------------------------------------------------------+---+

Q=seQ I=If R=Rept L=Loop O=clOseUN Z=End_construct C=Case
F=testFlg T=Txt-Ed E=Edit D=reDrw V=Vars N=Name S=Save/X X=Exit
H=help
```

Figure 10.1 The Low-Level Design and Menu Screen.

10.2.1 Screen Description:

HIPO = add-invoice :this is the link to the high-level design. The component "add-invoice" is a functional component. The above low-level structure constitutes the logic to accomplish it.

Module# 6-1 : this is the position of the component "add-invoice" in the high-level hierarchical structure. It is at level 6 and it is the first component on that level.

LL Name = stock-add-invoice : this is the name given to this low-level diagram. "Stock" is the system name and "add-invoice" is the functional component name.

Edit# - the heading of the edit numbers which are displayed next to the diagram.

10.3 The Low-Level Design.

The low-level design facility can be entered from the global view of the high-level design or via the design facility main menu. The analyst is presented with the low-level design environment screen and menu.

10.3.1 Low-Level Design Constructs.

All the structures which are available to the analyst in terms of the Nassi-Shneiderman diagramming technique, can now be selected from the menu. The appropriate text associated with each structure may also be entered.
The structures available are the standard structures used in structured program design, viz. a sequence structure (for normal sequential operations), decision structures (CASE or IF) and an iterative structure (unconditional repetition, e.g DO, FOR, and conditional repetition e.g REPEAT, UNTIL, WHILE). As the design grows, it can easily and quickly be modified using the edit functions of the menu.

10.3.2 The Low-Level Design Description Language.

The low-level design consists of selecting appropriate constructs and entering text to describe the logic. It has been suggested that it is better to use a design description language rather than an existing programming language [Sommerville :1985 p76]. SSDE allows the designer reasonable flexibility by permitting him to use familiar structures and keywords used in high-level programming languages. There is however limits to this flexibility so that code may be generated by SSDE for the construct entries. SSDE does however allow an abstract entry where the design description requirements are not applicable.

10.3.3 Secondary Diagrams.

If the logic at a particular point is very complex, and the designer wishes to postpone its consideration to a later stage, then he can accomplish this in terms of a secondary diagram. The "calling" diagram will then be the primary diagram for the "called" diagram, and the latter is considered a secondary diagram. This facility also means that if the analyst wishes to incorporate a previously designed structure (subprogram or procedure) he may do so in terms of a secondary diagram. The concept of software re-use in the construction of similar systems is thus implemented here too in terms of secondary diagrams.

10.4 The Menu.

Option: Q: seQ (Sequence).

This option will draw a sequence construct and obtain its text.

Option: I: If (decision)

A decision construct is drawn.

Option: R: Rept (While/Repeat). (Iterative Construct)

This option will draw a WHILE/REPEAT construct and obtain its related text.

Option: L: Loop

This option will draw the start of an UNTIL construct diagram and obtain its text.

Option: O: clOseUN (Terminate UNTIL loop).

This option will draw a "close" UNTIL construct diagram and obtain the UNTIL condition.

Option: Z: End_construct.

A decision, repetition or CASE construct has ended. No structure is drawn.

Option: C: Case.

A CASE construct is drawn.

Option: F: TestFlg.

A flag is set to indicate that this particular low-level design's code has been generated and tested.

Option: T: Txt_ed.

This option will allow the user to alter any existing text which is already displayed on the screen.

Option: D: ReDrw. (Re-draw LL design)

This option will clear the low-level display on the screen and re-draw the low-level design from the beginning of the diagram.

Option: V: Vars. (Display Symbol table)

This option will display a list of all the variables and their types.

Option: N: Name a current Low-Level system.

Give the low-level design in memory a new name. This can be necessary if one wishes to include this in another system or repeat it in this system. This is useful when so-called common modules (e.g a module that builds and presents a menu) are incorporated in similar systems. The concept of software re-use in the construction of similar systems is applied in this manner here.

Option: H: Help.

Obtain brief and detailed help about the options available and their functions.

The option for terminating a UNTIL construct (option O) has been kept separate from the general end construct (option Z) for the following reason. For Option O a construct has to be drawn and text entered. For Option Z no structure is drawn, but and "end-of-construct" record is created internally by the environment.

Option: S: Save/X.

Save the low-level design information in the appropriate databases and return to the main menu.

Option: X: Exit.

Return to the main menu without saving the low-level design. No low-level design information is lost, since it is retained in memory.

Option: E: Edit.

This option will allow the user to alter the diagram to re-structure the logic of the design. Upon selection of this option, the designer is presented with another menu for the actual edit: he can delete, insert, copy or move structures. On the right-hand side of the screen, edit numbers are displayed. These are used to identify constructs. This is quickest and easiest for both user and the SSDE system.

10.5 How the system stores information.

10.5.1 Dynamic Data Structures.

Dynamic data structures are also used to capture the low-level design. The following definition realizes this:

```
type
  cptr = ^crec;

  crec = record
         chapstr : string[1];  {construct id}
         chaptext : string[50];{construct text}
  end;

var
         tracechap : array[1..80] of cptr;
```

Figure 10.2 Low-Level Data Structure.

The above data structure was chosen in order to permit the edit functions to be simpler and to execute faster. An edit number is supplied when any edits are done and the above structure allows fast identification of the record required. If a large program needs to be designed, then the decomposition methodology can be used.

10.5.2 Capturing the Design.

When a new low-level construct is being created by the designer, memory is allocated to contain its record definition. A data structure, as defined above, will be created in main memory to hold the component's data. Example: if a sequential

construct is being created, then the following is executed:

```
count_ptr:=count_ptr+1;
new(cptr);
tracechap[count_ptr]:=cptr;
cptr^.chapstr:='1';
```

The text for this construct can be then be obtained and validated.

If the designer edits the low-level design, then the above array of pointers is updated to reflect the new state of the system.

10.6 Implementing the Low-Level Design Sub-Menu Options.

For all the constructs mentioned below, at least one new record is created. The next available position in "tracepic" is given the address of the new record. This record will store the construct type number together with its associated text.

10.6.1 Drawing the Structures.

(a) General.

The basic building block of the low-level Nassi-Shniederman diagramming technique is the rectangle. All the other structures are merely derivatives of the rectangle. In the program which draws low-level structures, a procedure is used to implement this important aspect. The general rectangle occupies three lines on the screen. The first and third are solid lines. The middle line is used to display the text entered for that construct. A single variable, "left-margin" is passed to this procedure to indicate how much the left margin has to be indented. As an example, a decision construct within a repetition construct has to be indented. When, for example, the decision construct is terminated, then the indentation "left-margin" is reduced. The amount of change to "left-margin", i.e increase or decrease in indentation, is the same for all types of structures. A "draw-line" variable indicates on which line the next structure should be drawn.

(b) Specific Constructs.

For certain constructs, like the decision construct, the above general procedure is called twice: for the true and false parts. Any special symbols which must be drawn are done by the construct specific procedures. For example, the "T" (True) and "F" (False) of a decision construct are drawn by the Decision-specific procedure. The construct-specific procedures

also have knowledge of the indentation factor in order to position unique characters correctly. The general rectangle and any indentation are handled by the general procedure.

(c) Obtaining the Text.

Once the specific structure has been drawn by the above procedures, the text associated with each is obtained. The text entered is checked (see below) and then displayed within the construct. The above mentioned variables, and others, are used to display the text on the correct line and correct position within the diagram.

(d) Scrolling.

Each time a construct has been drawn, the display position is checked to see whether the screen is full. If this is so and there is more to display, the screen is frozen and the analyst is asked if more should be displayed. If he replies in the negative then the low-level menu is presented. If he replies affirmative, then the display area is cleared and the subsequent structures, starting with the last structure of the previous screen, are drawn on the screen.

10.6.2 Setting the Test Flag.

A variable in memory stores this value for a particular low-level diagram. This test flag data is stored in the status record associated with this low-level design in database 3.

10.6.3 Editing Text Entries.

The edit number must be supplied. This edit number is also the position of the particular record in the array of pointers called "tracechap". In this way the record is located and the text obtained. The text to be edited is then moved to another temporary variable which is used by the routines to check syntax and identify variables. The contents are then displayed and the cursor positioned under the first character. The edit method is by default in insert mode. Any character typed is inserted in the appropriate place and the temporary variable is re-displayed. The left- and right-arrow keys may also be pressed to move the cursor position. If the DEL key is pressed then the character at the cursor position is removed from the variable which is then re-displayed. The position of the cursor is continuously noted in order that the edit can take place at the correct place in the temporary variable. When the edit is complete then the record is updated from the temporary variable. If keywords cannot be recognized, or syntax is incomplete, an appropriate error message is displayed on line 24. This message will persist until validity is achieved. The variables information is updated in the symbol table data for this component.

10.6.4 The Syntax for Entering Text.

After the analyst has requested a construct to be drawn, the text entered therein will be validated in three ways, to ensure it is reasonable, meaningful in that particular context and suitable for code generation. Instead of requiring users to learn a special language, SSDE aims at maximum flexibility by catering for a variety of keywords which are likely to come naturally to designers when specifying algorithms.

(a) Keywords.

All text entered must in general have a recognized keyword. These keywords are those normally used in English or in programming languages (e.g open, display, get, etc.) For the assignment type of text no specific assignment-like keyword is required if an "=" appears. The "IF" keyword within a decision construct is also optional, etc.

The following keywords may be used:

get, read, write, process, add, incr(ement), sub(tract), mult(iply), div(ide), rem(ainder), mod(ulo), print, put, end, begin, display, repeat, for, file, input, let, open, close, case, switch, while, loop, greater (than), less (than), more, exceed(s), equal to, equal(s), unequal, not, and, or, do, zero(ise), for each, foreach, for all, forall, gt, lt, ge, le, eq, ne

The following symbols: :=, =, +, /, -, *, <>, !=, >=, <=, (,), ^, !, [,], <, >, and "." are recognized.

(b) Variables.

All variable names must be preceded by a space and must end with "$" for alphanumeric items. If no "$" is specified then the default type is numeric.
The main reason for this is that variables need not all be declared at the start of each design. As the low-level design progresses, so the analyst can create variables without having to go back to the beginning to define them.
This naming convention ensures that there is no uncertainty of the type of a variable and avoids a continuous "look-up" by the designer to see the variable type. By using the above symbol the "declaration" is established. Ordinary variables, array variables, records and records with array fields can be entered. Array variables are recognized by the use of "(" and ")" or "[" and "]" brackets after the variable. Records and fields are recognized by the presence of a "." before the field name.

(c) General Syntax.

A general syntax scan is done and errors found are reported for correction.

"Type mismatch" situations are identified and not accepted as correct. For example "salary =+ hours-worked * pay-rate + x$", the "=+" and the "x$" will not be acceptable.

The text entry is rejected if no recognizable keyword is found. The rejected text or symbol is displayed and can be edited by the designer. The exception being an abstract entry which starts with a "?" character. Such an entry can be considered as a comment. A blank line will also be accepted. The latter case is applicable within an IF construct where there is no "else" part, or if the designer wishes to enter text at a later stage.

(d) Checking Conditions and Mathematical Expressions.

All conditions entered in decision and repetition constructs are checked for correctness. This check at the low-level design phase is essential to ensure that the code generated for these constructs is accurate.
The method applied is recursive descent parsing. If a condition is not correct then the designer can re-try.
Arithmetic expressions are also checked using recursive descent parsing.

(e) Errors Detected.

The following errors are detected when the low-level text is entered:

(i) Inconsistency in variable types.

Variables entered can end with a "$" or else numeric type is assumed. If within one expression both of these are present, a type mismatch results and it will not be accepted.

(ii) Incorrect arithmetic expressions.

The parser will check for valid arithmetic expressions. If there are errors no proper source code can be generated. So the entry is rejected.

(iii) Absence of keywords.

It is clear from the type of certain constructs what operation should take place. For example, in a decision construct, it is clear that an "IF" statement be generated. Keywords are required in non-assignment and non-condition types of logic. For these a keyword is essential for correct code generation.

10.6.5 Displaying and Re-drawing the Low-Level Design.

This option will clear the low-level display on the screen and re-draw the low-level design from the beginning of the diagram. If the complete diagram cannot fit on to one screen, then as much as possible is displayed on a screen. If the bottom of the screen is reached, the analyst will be asked if he wishes to continue with the diagram display or terminate the re-draw

option. The designer thus sees and works with a "page" of design at a time.
See "Scrolling" for discussion.

10.7 Editing the Low-Level Construction.

The menu presents a delete, insert and copy facility.

(a) Delete.

The edit number supplied will give the record to delete. For example, if "1" was entered, then the construct at "tracepic[1]" will be removed and the space it occupied will be freed. The pointers are updated, i.e "tracepic[1]" will now contain the pointer of "tracepic[2] and so on.

(b) Insert.

When the analyst selects the insert option then he will be requested to enter an edit number at the point before which he wishes to insert. For example, if "4" was entered, then the pointers from there onwards are moved to make "tracepic[4]" available. Different structures will require different number of records. A sequence structure requires one record, where as a decision construct requires at least three. New space is allocated for the record and the details of the new construct are recorded. The structure and text data will be captured in the record. All text which is entered is checked and any new variables defined, are added to the symbol table data.

(c) Copy.

The edit number of the structure to be copied is required. The copy then proceeds in a similar manner as the insert above.

(d) Move.

The move is a combination of the copy and delete operations.

(e) Updating the Low-Level Display.

When the edit is complete, the low-level design is re-drawn to reflect its latest status.

10.8 The Variable Screen Layout.

When the "display symbols" option is chosen the following screen is displayed:

```
HIPO = add-invoice Module# 6-1 LL Name = stock-add-invoice Edit#

  Variable          ArrayVar Field      ArrayFld      Type

   i                                                  integer
   j                                                  integer
   invoice-type                                       string
   reply                                              string
   return-code                                        integer
   emprec                    surname                  string
   a                 Y       m         Y              integer


Q=seQ I=If R=Rept L=Loop O=clOseUN Z=End_construct C=Case

F=testFlg T=Txt-Ed E=Edit D=reDrw V=Vars N=Name S=Save/X X=Exit

H=help
```

Figure 10.3 The Symbol Information Screen.

10.9 Storing Variable Information.

10.9.1 Dynamic Data Structures.

Dynamic data structures are also utilized to capture the variables which are created as the low-level design proceeds. The following definition realizes this:

```
type
  varpt = ^varec;

  varec = record
          varname : string[50];
          varfld  : string[50];
          vartype : char;
          arryvar : char;
          arryfld : char;
  end;

var
          tracevar : array[1..80] of varpt;
```

Figure 10.4 Symbol Table Data Structure.

This particular structure was chosen because when the code is generated, the above records need to be sorted so that all the fields of a particular record are together for ease of definition.

10.9.2 Capturing the Variable Information.

Text is not only scanned for structure, but also to pick up any variables which the analyst might have entered. Variable names are analyst-defined (and followed by a "$" for non-numeric items). Variables which are identified in this way, are stored (together with their type) in the allocated record.
A variable will only be defined once, but may of course be referred to many times. This information about variables and their types will later be stored in a database.
Abstract text is accepted as is and is not scanned for variables.

10.10 Implementation of the Low-Level Database Sub-Menu Options.

The implementation of the following low-level design related options are now discussed.

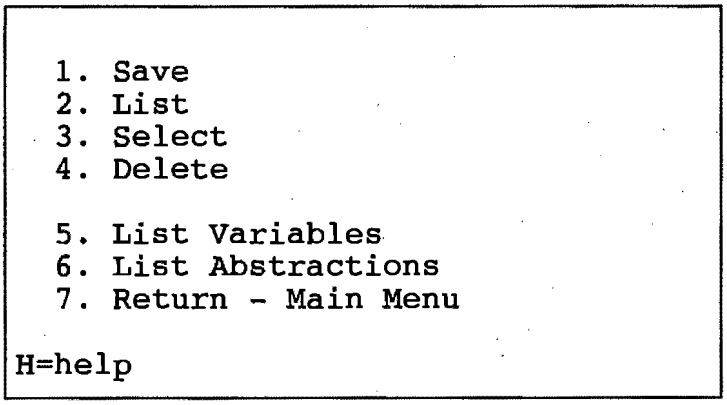


Figure 10.5 The HL Database Sub-Menu

10.10.1 Listing Low-Levels from the Database.

This is merely an informative option which will list all the low-level designs done to date.
The low-level name is constructed in the following way :the system name, a hyphen, the function name and optionally an "*". The first part (viz. the system name) indicates to which system this low-level design belongs. The second part indicates to which component in the high-level design this low-level design corresponds.
An "*" means that this is a primary low-level design, as opposed to a secondary low-level design. In order to determine to which primary design a particular secondary design belongs, one needs only look at the first two components of the name.

10.10.2 Deleting a Low-Level System.

The system searches to see if a "*" is embedded in the name of the low-level design. If no such "*" is present, then a warning is given to indicate that deleting this secondary low-level could have design implications for the corresponding primary low-level design. If an "*" is found embedded in the low-level design name which is to be deleted, then the information in the high-level databases which indicates that a particular functional component has a corresponding low-level design, is removed. The components which have been deleted, are marked by enclosing their names with a braces.

If a secondary low-level design is being deleted, then no information in the high-level databases need be updated. Appropriate entries in the symbol table database, VARDB, are also deleted.

10.10.3 Listing Variables and their Types.

This will list all the variables associated with a particular low-level design, with their types. The status record is located which will indicate how many VARDB records there are for the module selected. These records are retrieved and their contents listed. Only a single record's storage space is required since storage is dynamically allocated once and re-used for each record.

10.10.4 Finding Abstract Entries.

This activity will search through the records of a particular low-level system and identify if there are any abstract entries. The status record is located which indicates how many LLDB records have to be retrieved and searched. Abstract entries start with a "?" character. A message will be displayed indicating if abstract entries are present or not. Only a single record's storage space is required since storage is dynamically allocated once and re-used.

10.11 Summary.

The low-level design facility provides the designer with a framework wherein he can formulate and modify his detail design of the low-level logic. Structured design constructs are provided for the design. A fairly rigid syntax is required for the text entered. However, abstraction is also allowed, where the syntax requirements are relaxed. Thus the type of entry best suited to each step in a module can be chosen by the analyst.

CHAPTER 11:

OTHER SSDE FACILITIES

11.1 Introduction.

Validation of high-level and low-level variable usage, code generation, the manual written by SSDE and cross referencing are discussed here.

11.2.Verifying the High-Level to Low-Level Link.

This facility will verify whether the input/output variables in a high-level component have in fact been referenced in the corresponding low-level component.

11.2.1 The Verification.

SSDE will search through the variable data of the appropriate high-level component in the current design. All the variables defined there will then be compared with the symbol table of the corresponding low-level design.

11.2.2 Verification Report.

A report of all variables in the high-level design component will be given on the screen and next to the variable listed will be a message indicating whether the variable has been found in the input/output of the corresponding low-level design. All variables listed in the high-level functional

component must be referenced in the corresponding low-level design.

If either the high-level or low-level component cannot be found then the verification cannot proceed and an appropriate message is given.

```
    High-Level to Low-Level Linkage Check

Structure Chart Name=gross being compared with
                                    financialservices-gross

 Variable          Response

 ct                -
 hrs               input
 prate             input
 gross             output
 gross2            -
```

Figure 11.1 The High- to Low-Level Linkage Report

11.3 Generating Low-Level Code.

(a) Introduction.

Skeleton Pascal programming code for the low-level design in memory is sent to an ASCII file which can later be edited using any word-processor. This option scans the low-level design information to identify the structure type so the correct program statement can be constructed.

(b) Initial Tasks.

SSDE will assign a file name based on the low-level design name. This is constructed to adhere to the DOS requirements. The extension ".SSD" indicates that this file was created by SSDE. The "ASSIGN" and "REWRITE" statements are used to accomplish this. The first statement with the program name is then written.

(c) Variable Definitions.

An array of pointers called "tracevar" is used to obtain the variable definitions. (See Chapter 10) For each variable created during the low-level design activity, a record was produced containing its name and type. These definitions are placed after the program name statement (or after the procedure heading if it is a secondary diagram). Default sizes of 12 and 20 are used in all string and array declarations respectively. A comment "{*check sizes*} is inserted. These sizes can be changed by the designer in the ASCII file to suit his particular application requirements.

(d) Code Generation.

All text entries have already been syntactically checked at low-level construction time. This important aspect makes the code generation task much easier.

Each low-level design record contains the construct code and the associated text. The text associated with a particular construct is modified to conform to the requirements of the programming language, for example, a ";" is added at the end of a PASCAL statement. Non-Pascal keywords are translated to their Pascal equivalents, for example "get" or "input" are changed into "readln"; "display" or "print" become "writeln". As statements are formed they are immediately written to the ASCII file. When an "IF" statement is formed it is not written until the end-of-if marker, in this case "structure type=I", is encountered. The "IF" is formed in memory with the use of dynamic data structures and when complete, then the whole multi-line statement is written from memory to the ASCII file with appropriate indentation. The reason for forming the IF in memory is because "True" and "false" parts are mixed and cannot be collected as a unit. An indentation count is maintained so that when a statement is written, the necessary indentation is present to improve readability and understanding of the logic.

Due to a lack of information it is not possible to generate code for an abstract type entry. A comment will be generated in this case with all unnecessary blanks removed.
The designer can then easily use an editor to change such a comment to the correct statement(s).

(e) Procedures.

All secondary low-level diagrams are written as PASCAL procedures.

(f) An Example.

Consider the following small low-level diagram:

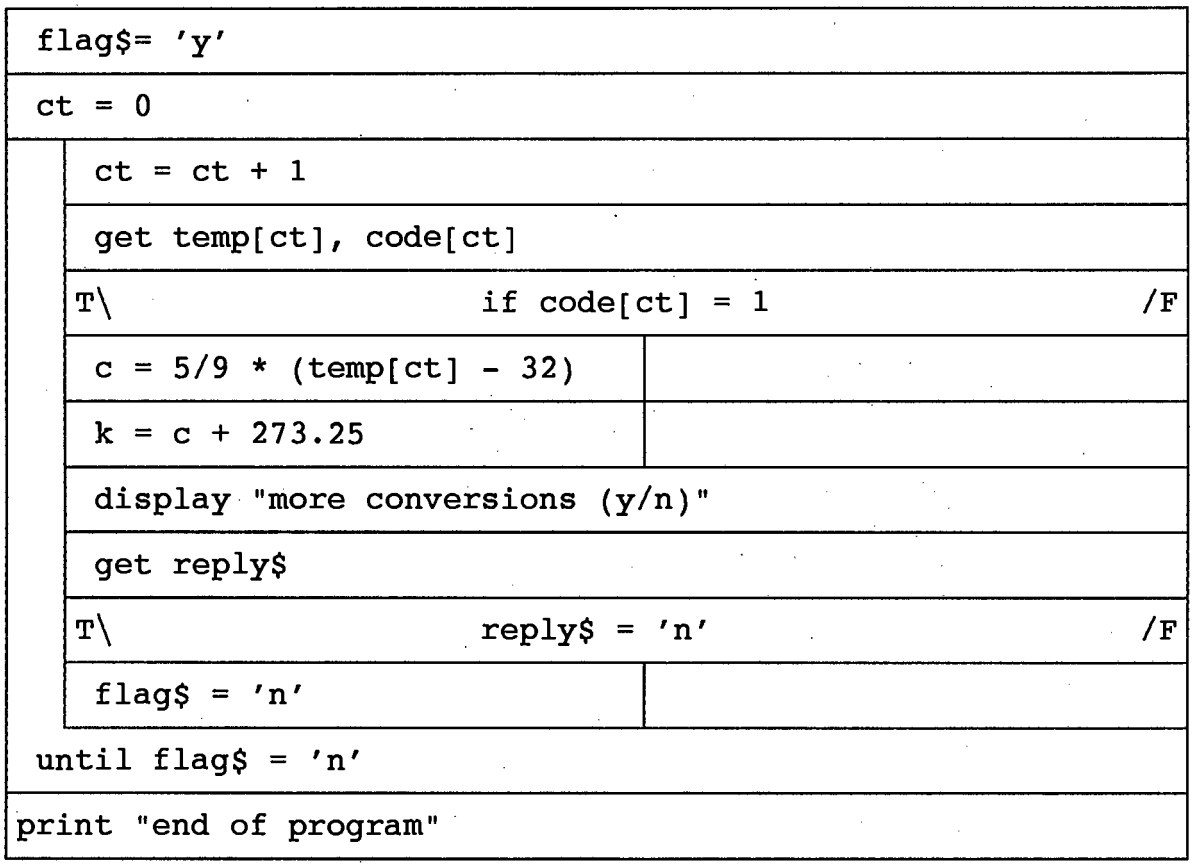


Figure 11.2 A Low-level Diagram

While the code was entered during the design process, its syntax and semantics were checked.

This low-level design is captured in memory as follows:

A structure type and the structure text constitute a record.

```
first construct: structure_type = Q
                 structure_text = "flag$ = 'y'"

second construct: structure_type = Q
                  structure_text = "ct = 0"

next construct : structure_type = L
                 structure_text = "get temp[ct], code[ct]"

next construct : structure_type = I
                 structure_text = "if code[ct] = 1"
               : structure_type = I
                 structure_text = "c = 5/9 * (temp[ct] - 32)"
               : structure_type = I
                 structure_text =

next construct : structure_type = Q
                 structure_text = "k = c + 273.25"
               : structure_type = Q
                 structure_text =

next construct : structure_type = Z (end if)

...similarly for remaining constructs
```

The following table of variables would have been built for the low-level design shown above:

Variable	ArrayVar	Field	ArrayFld	Type
flag				string
ct				integer
temp	Y			integer
code	Y			integer
c				integer
k				integer
reply				string

Figure 11.3 Table of Variables.

(i) Writing the First Statement.

The code for the above is written to an ASCII file as follows: The first statement, namely the program name is generated. The name is taken from the low-level diagram name and the "(input,output);" is added, unless it is a procedure. If the corresponding high-level design has a general comment, then it is written next as a comment here.

(ii) Defining Types and Variables.

Next the types and variables are defined. All the variables created together with their type, were captured in data structures as the design progressed. These data structures are used to generate the variable declarations after the program name. Sort the table based on the "variable" column so that all the fields of a record are grouped together.

The following procedure is used to accomplish this task:

* scan the "arrayvar" of the symbol table for entries which have a "y" and their "arrayfld" entry is empty. This is an ordinary array variable. Call its type name "array1" and check its "type" entry to determine the type of the array. The variable name, e.g "a" is kept so that when the variable part is written, the following is formed "a : array1".
  Subsequent array types will be called "array2", "array3", etc.

* scan the "fld" column for any entries. A row which has such an entry will be defined as a record. Call its type name "rec1". Check the "type" entry for this "field" entry. The record can now be defined. Scan the next record to see if it has the same variable name. If it has, then its "field" entry is also part of this record definition. Continue until the variable name differs. Conclude this multi-line statement with a "end;". The variable name has been kept in order to write "variable_name :rec1" in the variable definition part. Subsequent record types will have the name "rec2", "rec3", etc. If the "arrayvar" and "arrayfld" columns have entries then these record variables are also arrays. Their array portion will be generated as above.

* any remaining variables are just ordinary variables. Their types are given in the "type" entry column.

(iii) Writing the Program Body.

The "Begin" is now written and the definition of the first construct consulted. The structure type (=1) indicates a single sequential operation. The text part has a "=" character which indicates assignment. The "$" is dropped, the "=" changed to ":=" and a ";" added to the end of the text. This is then written to the file.

The next construct has a type of "4" which indicates the start of a loop. The string "repeat" is written. The indentation factor is increased to that subsequent statements written will be indented. The text "get" is changed to "readln(" and the variable part added. Next the ");" part is added and "readln(temp,code);" is written to the file with indentation.

The next record has a structure type of "2" which means an IF statement will be generated. The IF will be generated in main memory until a structure type of "13" (end if) is encountered. The "If code = 1" plus "then" is kept in a dynamic data structure. A "begin" is also stored. The indentation factor is increased. The "then" and "else" parts are similarly stored. When the structure type "I" is encountered, the IF is concluded and it is written from the structures in main memory to the file. The indentation factor is decreased.

The rest of the constructs are translated in a similar manner. Finally the "end. {*name*}" is written and the file is closed. Any word processor can then be used to edit the file.

```
program convert(input,output);
{this comment from HL comment}
type array1 :array[1..20] of integer; {*check size*}

var temp :array1;
    code :array1;
    flag :string[12]; {*check size*}
    ct :integer;
    c :integer;
    k :integer;
    reply :string[12]; {*check size*}

begin
   flag := 'y';
   ct := 0;
   repeat
     ct := ct + 1;
     readln(temp[ct],code[ct]);
     if code[ct] = 1 then
     begin
       c := 5/9 * (temp[ct] - 32);
       k := c + 273.25;
     end;
     writeln("more conversions (y/n)";
     readln(reply);
     if reply = 'n' then
     begin
       flag := 'n';
     end;
   until flag = 'y';
   writeln('end of program');
end; {*convert*}
```

Figure 11.4 Code Generated

11.4 Drafting the Manual.

(a) Introduction.

Applications being designed should be accompanied by a system and user manual which explains the functions of the system and how it can be used. SSDE provides support for this requirement

by writing a skeleton manual for each from the high-level design information available.

(b) Writing the System Manual.

The array of pointers containing the high-level design records are used to locate the record for each high-level component. Its fields, viz., the component name, inputs, outputs, process steps and the general comment for each functional component are used as a basis to generate the skeleton system manual as an ASCII file.
All the modules which make up the high-level definition, with their components, are used to construct the manual. The write-up of the hierarchical tree is done "branch-by-branch". The manual is not written a level at a time, but proceeds through the levels to complete one branch of the tree. This means that the write-up for a particular module and its dependents are first completed before the next high module and its dependents ("branch") are written. In this way the closely related activities are written up together.
The designer can extend the skeleton manual to a more comprehensive document using a word processor.

11.5 Summary.

Automatic code generation is a useful productivity aid and relieves the designer from the time consuming task of manually having to write code. SSDE provides code generation in the form of Pascal statements. SSDE assists the designer in compiling

the system manual by writing a skeleton one from the high-level definition.

CHAPTER 12:

EVALUATION.

12.1 Introduction.

The contribution that SSDE can make towards supporting the software development life cycle is investigated and other advantages offered are mentioned. SSDE was evaluated by a number of users and the results of this exercise is reported.

12.2 SSDE - Its Role in the Software Development Life Cycle.

12.2.1 Planning.

Planning, in the sense of an initial investigation, is possible via the high-level design facility. The designer can quickly use the environment to perform an initial outline of the system for presentation to other interested parties and for use in a feasibility study.

12.2.2 Analysis and Logical Design.

The environment provides adequate support for this stage through the high-level design facility. Using this the designer can view or edit or print the complete conceptual model at any point during the design [Sceffer et al :1985 p52]. The whole design of the logical model is greatly accelerated through automation. A fast high-level definition can be completed in the global context. Using the local-neighbour view, the analyst has to specify the inputs, outputs, process steps (and any

comment) for each component. This strategy will assist in further understanding the problem and clarifying the design. Design alternatives may be constructed and compared.

12.2.3 Physical Design.

Physical design is concerned with the more detailed design of the system. Through the use of the low-level design facility the analyst can design his low-level logic in as much detail as he requires. This logic can at any point be revised and/or re-structured. Design refinement, in terms of using secondary low-level designs, is also supported. Abstraction and the use of secondary diagrams makes it possible for the analyst to postpone the design of certain specific logic to a later stage. This avoids distracting his thoughts when designing the logic.

12.2.4 Implementation or Construction.

A complete document of the system design can be produced which can greatly assist the programming staff in the actual implementation of the system.
The environment provides for the automatic translation of the low-level constructs into skeleton PASCAL code. This facility should make a considerable contribution towards improving implementation schedules.
During the system testing phase, possible refinement or tuning might be necessary to ensure that the software fulfils the desired system objectives. The implications of such steps can quickly be determined by consulting the document produced by

SSDE or via on-line database query. Test status is also maintained.

12.2.5 Maintenance.

By viewing the system via the environment or its documented output, maintenance personnel can quickly familiarize themselves with the overall system. They can also view any low-level logic. Any changes can be done via the environment to ensure that the documentation concerning the system is up to date. The edit facilities of SSDE makes these changes easy to implement.
Since the information about any system is kept in on-line databases, the time required to modify the system should be greatly reduced [Aktas :1987 p16]. Historical data can be checked to preclude repeating earlier design errors.

12.3 User Survey Report.

12.3.1 Introduction.

In order to measure the usefulness of an environment such as SSDE, an evaluation exercise was conducted. A group of senior undergraduate computer science students were presented with a typical application problem. Since the software engineering students were the target group, they were asked to evaluate SSDE. They had to use SSDE to construct a software solution. Finally they were asked to complete a questionnaire. The details of this user survey can be found in appendix C. The

author also used SSDE in completing a larger problem and found SSDE to be a helpful tool in the software development task.

12.3.2 Problem Addressed.

The evaluators were required to design a financial information system which would provide different financial services for its users. These services included payroll, credit card application, economic indicators reporting and banking systems.

12.3.3 Time Constraints.

To ensure the evaluators gained as much experience as possible with every facet of SSDE, they were assigned a large number of design tasks, not all of which had to be completed in full. The problem presented to them was not a large one but was kept to a reasonable size because of time constraints and to make the problem more manageable. In this way a broader spectrum of problems were encountered and a minimum of time spent on repetitive but uninteresting tasks.

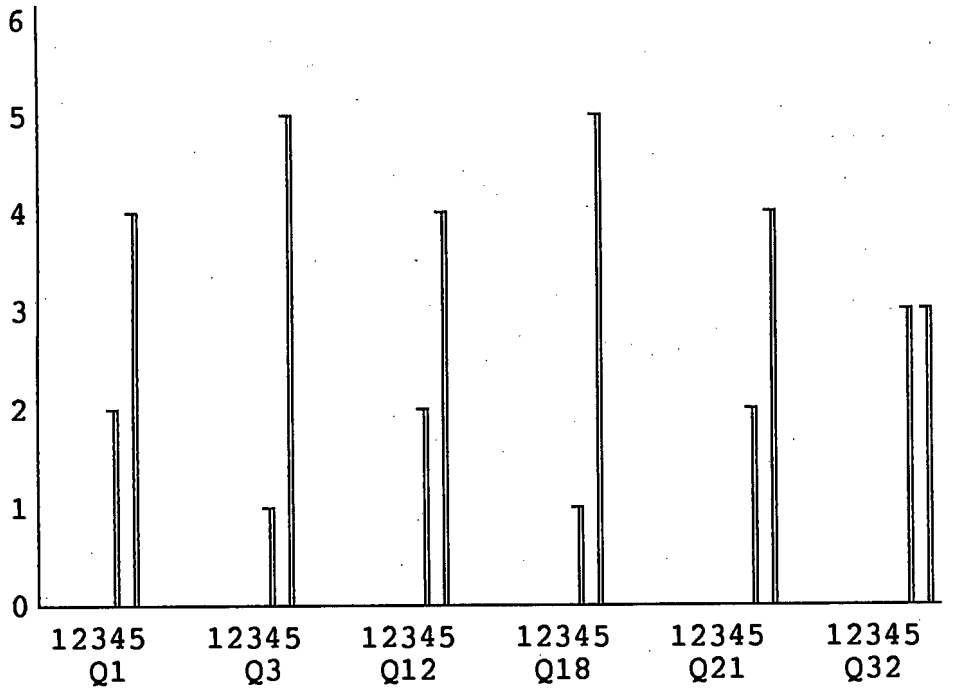
They were required to complete all high-level designs. For the low-level detailed definition however, they were only required to complete some of the low-level components. This latter constraint reduced the design undertaking and made it possible for the evaluators to try the many other facilities provided by SSDE.

12.3.4 General Summary.

The replies received were overwhelmingly positive towards SSDE. Its success was confirmed by the complete absence of negative feedback. The automated design support with editing at the high- and low-level design stages, the database facilities and the code generation capability were well received. They mentioned that SSDE was a "great idea" and that it demonstrated "great potential" as a software development environment and that it was easy to learn. See appendix C for the complete set of responses.

12.3.5 Histogram.

The following histogram summarizes the responses to some of the more important questions answered by the evaluators. It can be seen from this histogram that the responses were very positive in general.



rating: 1 (not at all)
2 (very little)
3 (average)
4 (quite a lot)
5 (very much indeed)

legend: Q1 (productivity improved)
Q3 (automated design preferred over manual)
Q12 (efficiency of menus)
Q18 (code generation useful)
Q21 (HL diagrams contribute towards understanding)
Q32 (LL diagrams contribute towards understanding)

Along the X-axis are the ratings for these questions.

Along the Y-axis are the number of responses.

12.4 Advantages Offered by SSDE.

1. An inherent design methodology is present which guides the system architect through a design cycle starting at the high-level (broad outline) of the system and proceeding to a low-level (detail) logic design.

2. Beyond this, SSDE aims at flexibility, so the designer can follow his own methodology.

3. Graphical representation clarifies a users perception of a system and makes it easier to commit to a particular design option or not. Complex systems can be simplified by abstraction.

4. Menus allow for easy diagram manipulation and movement within the system. They are simple enough for even novices to use, yet allow fast navigation for experts.

5. Ease of communication is made possible by the graphics facilities and the documentation provided.

6. Increased productivity is provided by the automation of design components, e.g. diagram production, documentation, editing at all levels, code generation, etc.

7. Defects in the system have been minimized by using structured techniques. Furthermore it is suggested that modules which are designed in a hierarchy normally form a good structure.

8. Incremental system design is possible. Design, implement and run a partial system and then the system can "grow" as new components are added.

9. The reuse of existing software in the construction of new applications is made possible.

10. The manipulation of program constructs in a language independent manner is supported.

11. Maintenance costs should also be reduced since a design consistency for all systems will be maintained by using SSDE.

12. Novice analysts and system designers can be shown how to accomplish system design and decomposition by experimenting with the environment.

13. Since the complete design information of the system is stored in a database accessed and maintained via SSDE, an organization's reliance on the loyalty and expertise of a few individuals is greatly reduced.

14. SSDE and any designs produced by it are completely portable and can with minimum effort be transferred to any PC running DOS.

15. For the user of a fixed disk, there is no noticeable delay in processing as SSDE loads and runs different programs to accomplish its tasks and when data is retrieved / written to secondary storage.

16. The drawing of high-level and low-level diagrams is done quickly and the designer is not delayed in any way.

12.5 Summary.

SSDE provides automated support for the different phases of the software development life-cycle. Facilities such as on-line databases, pictorial presentations and code generation help the analyst to create a conceptual model, to define the functional components which constitute the proposed solution, to spell out the detail logic required to accomplish these, and to compare and analyze alternative designs.

CHAPTER 13:

DESIGN METHODOLOGIES

13.1 Definition and Introduction.

A Methodology can be defined as a collection of methods, procedures, working concepts, rules and postulates applied by a science, art or discipline.
The term strategy is often used interchangeably as a synonym for methodology in the literature [Bergland et al:1981, Gomaa :1979, Parker :1978]. An important engineering principle is that if one pursues a method or algorithm then the chances are very good that a correct solution will be arrived at.

13.2 Classification of Methodologies.

The following three classification groups for methodologies currently exist: Functional Decomposition Methodologies, Data-Oriented Methodologies and Prescriptive Methodologies.

13.3 Functional Decomposition Methodologies.

These methodologies emphasize the approach whereby the system is divided into a number of smaller sub-systems with the resulting rudimentary sub-systems being less complicated and therefore not so difficult to design and install. The actual system functions are the major interest to the designer, hence the name "functional method".

Examples of Functional Decomposition Methodologies are:

13.3.1 Top-Down approach:

In this approach the top-most level decisions are made first and then successively lower levels are done, the bottom level decisions being made last. It is at present considered the finest method for systems design [Aktas :1987 p136]. Examples of methodologies which implement this approach are, HIPO and Stepwise Refinement. The top-down approach is often also called "decision analysis".

(a) HIPO (Hierarchy plus Input-Process-Output)

Initially IBM developed this as a documentation tool and it is often now referred to as a design methodology [Aktas :1987 p137]. HIPO can also be used to document other designs.

(b) Stepwise Refinement. (SR)

Here a constant problem statement is available and a number of design solutions and alternatives are proposed. After selecting an optimal solution from amongst the same level solutions, then the next level alternative solutions are proposed and the process is repeated [Ledgard :1973] [Wirth :1971].

13.3.2 Bottom-Up approach

In this approach the bottom level decisions are made first then successively higher levels, ending with the top level last. This approach is often also called "data analysis".

13.4 Data-Oriented Methodologies.

Here the characteristics of the data to be processed are the main concern.
Two data-oriented methodologies can be identified, viz., Data-Flow Oriented Methodologies and Data-Structure Oriented Methodologies. The former approach decomposes the system into a number of modules by defining the data element types and their logical conduct within a system. The data flow logic and the functional association amongst the modules of the system, formulates the logical organization of the particular system. The Data-Structure Oriented methodology approach focuses on the input/output data structures of a particular system. These input/output structures form the point of departure for the eventual structure of the system. Functional relationships amongst modules of the system are formulated in terms of this system structure.

Examples of Data-Oriented Methodologies are:

13.4.1 Data-Flow Oriented Methodologies

(a) SADT

Structured Analysis and Design Technique (SADT) is a methodology using graphical diagramming techniques which can be used for the analysis and design stages of a system development process. It is a tool which can be used in all phases of systems development. A system is seen as consisting of objects, documents or data, happenings performed by people, machines or software and their interrelationships. Activity and data diagrams are used here [Ross :1980, Conner :1980, Ross et al :1977, Ross et al :1976].

(b) Composite Design

Composite Design (CD) and Structured Design (SD) were initially intended to ease the latter stages of of the development process, viz., coding, debugging and modification. Later these strategies were extended to also include the systems development activities. The main difference between SD and CD is the emphasis on modularity in CD. Module coupling and module strength are used to achieve the required modularity [Myers :1975,1973, Stevens et al :1974].

(c) Structured Design

Structured Design is a popular and extensively used methodology for systems development activities. A data-flow diagram is produced indicating the transformation which the data experiences within a system. Transform analysis, Transaction analysis and decomposition are examples of tools which can be used in structured design. A limitation of this type of design is that it requires another tool for detailed design [Dickonson :1981, Stevens :1981, Yourdon :1981, Yourdon et al:1979, De Marco :1978].

13.4.2 Data Structure Oriented Methodologies

(a) Jacksons's Methodology

In this methodology the time dimension of a system or its dynamic quality is also considered. Basic terms relevant here are entity and action. An entity (an object from the real world) partakes in a time-ordered set of mechanisms. One or more entities can share an action within an event. It models real world environments and also models the functions of a system [Jackson :1987, 1983, Hicyilmaz :1985].

(b) Warnier/Orr Methodology

The starting point for this methodology is the definition of the output required in terms of Warnier diagrams. The different variations of the methodology are Logical Construction of

Programs (LCP), Logical Construction of Systems (LCS), Structured Systems Development (SSD) and Data Structured Systems Development (DSSD). It is a data-centred design approach.

13.5 Prescriptive Methodologies.

These are generally automated facilities which support system software development efforts. An important objective of this type of methodology is to free the system designer from many of the manual, time-consuming technicalities of the design effort. This is accomplished by providing him with a prescriptive approach which will capture his logic design layout and possibly generate the necessary software. Thus the designer can concentrate more on the actual logic design.

Examples of Prescriptive Methodologies are:

(a) Nassi-Shneiderman or Chapin's Approach.

This approach can represent low-level program constructs such as sequence, selection and iteration and in general the program constructs have one entry and one exit point.
This technique is well suited for designing low-level (detail) program logic. Using this as a design technique, results in structured programs, with all the associated advantages thereof.

(b) Object-Oriented Design.

Here the concept of an object is fundamental in the decomposition process associated with a system. An object is an entity whose conduct is determined by the actions performed on it as well as its requirements with respect to other objects. The object, its properties and its its interface are identified. Objects communicate by message-passing. The object can then be implemented [Booch :1986].

(c) Problem Analysis Diagram

A system can be described in terms of the activities present within the system as well as the flow of data between these activities. A description such as this can be used in constructing a problem analysis diagram (sometimes also called a requirements diagram). Symbols (or elements) e.g an activity, form part of this diagram and it is possible that these elements could be understood by the user, allowing further user input to occur [Gilbert 1983: pp33+].

13.6 Design Methodologies and their use in SSDE.

(a) Functional Decomposition.

SSDE encourages this design approach and clearly encourages the use of top-down, HIPO or Step-wise strategies. Step-wise refinement can be employed in the designing of systems in an incremental manner. Suppose a medium sized system is being

constructed which has a clearly defined input stage. Now using the top-down approach, only the input stage (its functional components) can be designed at the high-level facility and then the detailed code for this can be constructed at the low-level facility. (At this stage no high- or low-level design has occurred for any of the other stages of this system.)
Although SSDE does not encourage a bottom-up approach, this can still be utilized in conjunction with top-down development in SSDE.

(b) Data-Oriented Methodologies.

SSDE can be used with a data-structure oriented approach and even with data-flow oriented strategies as these also result in functional decomposition of a system. Naturally it is less helpful in the latter case as it does not support the data flow diagramming techniques used in these methodologies, so they would have to be constructed manually.

(c) Prescriptive Methodologies.

The very nature of these methodologies is such that a general software engineering environment cannot support all of these. Those chosen in SSDE are the HIPO / Structure Chart and the Nassi-Shneiderman techniques.

(d) SSDE and These Methodologies.

SSDE aims at providing general tools that are widely applicable and do not force users to learn and apply specific strategies. Thus SSDE supports a flexible approach to software design which incorporates high-level and low-level definitions and encourages structured design. If one wishes to standardize a methodology amongst designers, the SSDE code could be altered to force a particular approach. Certain menu options need to be altered or removed to reduce the flexible design methodology which SSDE permits.

(e) Partial Definitions.

A process of partial definitions can be accomplished using an incremental design methodology. Sometimes it is necessary, or within certain organizations it is the accepted methodology, to design only a subset of the high-level definition and then to proceed immediately with an associated low-level design in order that a partial system can be developed, tested and demonstrated. The methodology then proceeds with another partial high-level definition and the process is repeated until finally the partial definitions together constitute the complete envisaged system.

13.7 Summary.

Despite the number of methodologies available to the designer, be they manual or automated strategies, much of the success of the development process still depends on the expertise and

experience of the designer. SSDE is a valuable aid to the system designer allowing a wide rang of methodologies to be followed.

CHAPTER 14:

CONCLUSIONS AND FUTURE WORK

SSDE is considered in terms of the original aims and objectives. Areas within SSDE which could be further investigated are discussed.

14.1 Re-Statement of Objectives.

The main thrust of this research effort has been to produce an automated environment which will assist an analyst in the construction of software logic. The facility had to provide a framework wherein structured design techniques can be used in drawing up the blueprint for the high-level and low-level design. This research aimed at providing an automated facility which strives to reduce the number of manual tasks the systems designer has to perform, and which is cheap and easy to use.

14.2 Evaluation in terms of Hypotheses / Objectives.

SSDE gives automated assistance for many of the tasks associated with the design of software. A functionalized menu system provides ease of navigation within SSDE with no design information being lost.

The hierarchical definition of the system tasks and their inter-relationships is accomplished using the high-level design facility. This allows the components of the hierarchy to be fully defined in terms of their input, processing and output

parts. The diagramming techniques provide an effective way in communicating the design of the system to other interested parties, particularly users.

Components in the high-level definition which require detailed design can be considered using the low-level design facility.

The code generation facility automatically writes code from this design information.

Several databases are used as a central repository to capture various aspects of the system design and are accessible via on-line query.

The high-level and low-level designs that have been created using SSDE can be printed for documentation and maintenance purposes.

Structured techniques are encouraged throughout to ensure that the design proceeds in a disciplined manner. In the low-level design only structured constructs are permitted. The modular design approach and the use of databases permits the software re-use concept to be applied.

No rigid methodology is forced upon the designer. SSDE provides a design framework that can support a variety of design methodologies.

The evaluation of SSDE showed that software designers prefer an automated tool or environment above the normal manual methods. Design automation results in systems being designed much quicker than would otherwise be the case, increasing the designer's productivity.

SSDE has met its initial goals in that it is easy to learn, low cost, enforces structured techniques and provides diagrams which are clear and easy to modify. As much as possible of the software development task has been automated, error detection is included and the environment will be useful throughout the system life-cycle. End-users should be able to understand all output produced by SSDE and should themselves be able to create high-level designs as well as low-level designs comprising abstract (uninterpreted) entries. This is possible particularly because all aspects of SSDE are optional so complex ones (eg. input/output specification) can simply be omitted by end-users.

14.3 Areas for Future Work.

14.3.1 Improvements.

Diagrams and control information which is presented on the screen may be improved by using special graphics and a colour scheme. To improve productivity even further, a mouse or touch support could be used in the selection of menu options and also in editing the design. These facilities have not been included due to a lack of suitable hardware.

SSDE allows designers to follow a flexible design approach. The reason for this is that designing software is considered a creative task. However, SSDE code could be altered to force a particular design methodology. The way to accomplish this would be to change or remove menu options, since it is the menu system which permits the flexibility.

The documentation facilities can be extended in order to provide an even more comprehensive service. For example: list all low-level designs which has have their code tested; list high-level design with (or without) a general comment; etc. These are trivial extensions which can easily be included.

The default array size is set to twenty at code generation time. It can be set more accurately by searching through an entire module (eg. looping structures) to determine the precise array size.

Validation procedures in the high- and low-level design parts can be extended, eg. misspelled words can be rectified. Care must be exercised that the validation does not distract the analyst's design thoughts however.

14.3.2 Extensions.

It would be desirable if SSDE could allow members of a team to work on a single design in a multi-user environment.

Artificial Intelligence techniques could be used to provide SSDE with a knowledge base of software engineering expertise and opportunities for consulting this.

Software developed for real-life applications are subject to budget controls and time schedules. These aspects can be accommodated within SSDE so that management and project leaders can evaluate the progress being made in the context of these constraints.

An "export" and "import" facility would be useful to allow sharing of data between SSDE and other applications.

It is possible to generate code in other programming languages, as the operations and data types used in the low-level design were specifically chosen to be the most common across the spectrum of imperative languages.

A data flow diagram (DFD), or a bubble chart is a logical model which shows the overall data flow through a system. In addition to Structure Charts, designers often prefer to draw a DFD illustrating the data flow in a system.

14.4 Summary.

The evaluation exercise clearly showed that the initial goals set for SSDE were accomplished. Several insights, as regards design automation, were gained as a result of the research.

Scope for future enhancements exist but these must not be incorporated if it is at the expense of SSDE's ease of use, flexibility, low cost and efficient performance.

Appendix A:

USER MANUAL FOR SSDE.

A.1 Overview: Designing a System using SSDE.

When large computer applications are developed, then a careful design strategy or methodology is normally followed to improve the chances of finding a correct solution. SSDE permits top-down and structured decomposition.
The design strategy or methodology encouraged by SSDE can be divided into two distinct parts.

A.1.1 High-Level Conceptual Design.

The first part of the design is concerned with the functional components which will constitute the system. This conceptual design shows these components (or modules) and their relationships with each other and also the movement of data. This design activity is often referred to as high-level design. The high-level diagramming technique used in SSDE, is the familiar Structure Chart technique combined with aspects from the HIPO diagramming technique. These techniques form a strict hierarchy i.e all modules (except the top one) have only one parent.

A.1.2 Low-Level Logic Definition.

The second part of the design is concerned with the more detail outline of the logic required to accomplish the tasks outlined in the high-level design. This activity is commonly called the low-level design task. SSDE uses the Nassi-Shneiderman diagramming technique to accomplish this.

A.2 Starting SSDE.

To start the system, type in START and press <enter>.

A.3 Using the Main Menu.

```
        1.  Design
        2.  Database
        3.  Documentation
        4.  Exit

H=help
```

Enter the option number you require or "h" for help.

Option: Design

Define high-level conceptual components or low-level detail logic.

Option: Database

Perform database related activities.

Option: Documentation

Generate of the high-level and low-level documentation.

Option: Exit

Return to DOS.

Option: h

Short and detailed information about the options on the present screen.

A.3.1 Using the Design Sub-menu.

```
  1. High-Level Design
  2. Low-Level Design
  3. Verify HL-LL Link
  4. Generate LL code
  5. Return - Main Menu

H=help
```

Enter the option number you require or "h" for help.

(N.B Whenever you need to enter a module number, enter the two parts with a space between. e.g. module 21 is entered as 2 1<enter>.)

Option: High-Level Design

Start a new high-level design or continue with an existing one.

Option: Low-Level Design

Start a low-level design or continue with an existing one.

Option: Verify HL - LL Link

Very that variables defined in the high-level component are referenced in the associated low-level design.

Option:Generate Code

Generate skeleton Pascal programming code for the low-level design currently in memory and write to a ASCII file.

Option: Return - Main Menu

Return to Main Menu.

Option: h

Short and detailed information about the options on the present screen.

A.3.2 Using the Database Sub-menu.

```
1. High-Level Systems
2. Low-Level Systems
3. Return - Main Menu

H=help
```

Enter the option number you require or "h" for help.

Option: High-Level Systems

Perform database operations and queries related to the high-level conceptual definition.

Option: Low-Level Systems

Execute database operations and queries related to the low-level definition.

Option: h

Short and detailed information about the options on the present screen.

(a) Using the High-Level Systems Sub-Menu.

```
  1. Save
  2. List
  3. Select
  4. Delete

  5. Find Variable
  6. HL Components with no LL
  7. List Deletions
  8. Return - Main Menu

H=help
```

Enter the option number you require or "h" for help.

Option: Save

Save all the high-level design data.

Option: List

List of all the high-level designs in database.

Option: Select

Choose an existing high-level design from the database.

Option: Delete

Delete a high-level system. The system is not physically deleted and can be recovered.

Option: Find Variable

Within a high-level system definition, list all hierarchical components which reference a particular variable.

Option: HL Components with no LL

Within a system, list all high-level "tip" modules which have no corresponding low-level design.

Option: List Deletions

List all high-level systems which have been deleted and you can subsequently recover such a design.

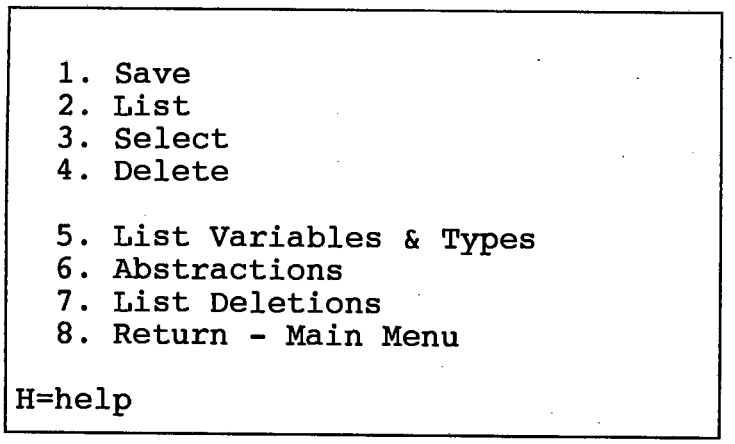
Option: Return - Main Menu

Return to Main Menu.

Option: H

Short and detailed information about the options on the present screen.

(b) Using the Low-Level Systems Sub-Menu.



Enter the option number you require or "h" for help.

Option: Save

Save all the low-level design data and variable definitions.

Option: List

List of all the low-level designs in database.

Option: Select

Choose an existing low-level design from the database.

Option: Delete

Delete a low-level system. The system is not physically deleted and can be recovered.

Option: List Variables and Types

All the variables and their types for a particular low-level design are listed from the database.

Option: Abstractions

A low-level design will be checked to see if any abstraction entries exist in the design.

Option: List Deletions

List all low-level systems which have been deleted and you can subsequently recover such a design.

A.3.3 Using the Documentation Facility sub-menu.

```
        PRINTER OPTIONS

  1.  Print Current HL Design
  2.  Print Current LL Design
  3.  Print all LL Designs

  4.  Generate Manual
  5.  Print Cross-Reference
  6.  Return - Main Menu

H=Help
```

Option: Print Current HL Design

The high-level hierarchical structure is printed together with all the detail associated with each component.

Option: Print Current LL Design

The low-level construct diagram is printed together with the text it contains.

Option: Print all LL Designs.

This prints out all the low-level designs of the current system.

Option: Generate Manual

This will generate a skeleton system manual write-up from the high-level definition of the system.

Option: Print Cross-Reference.

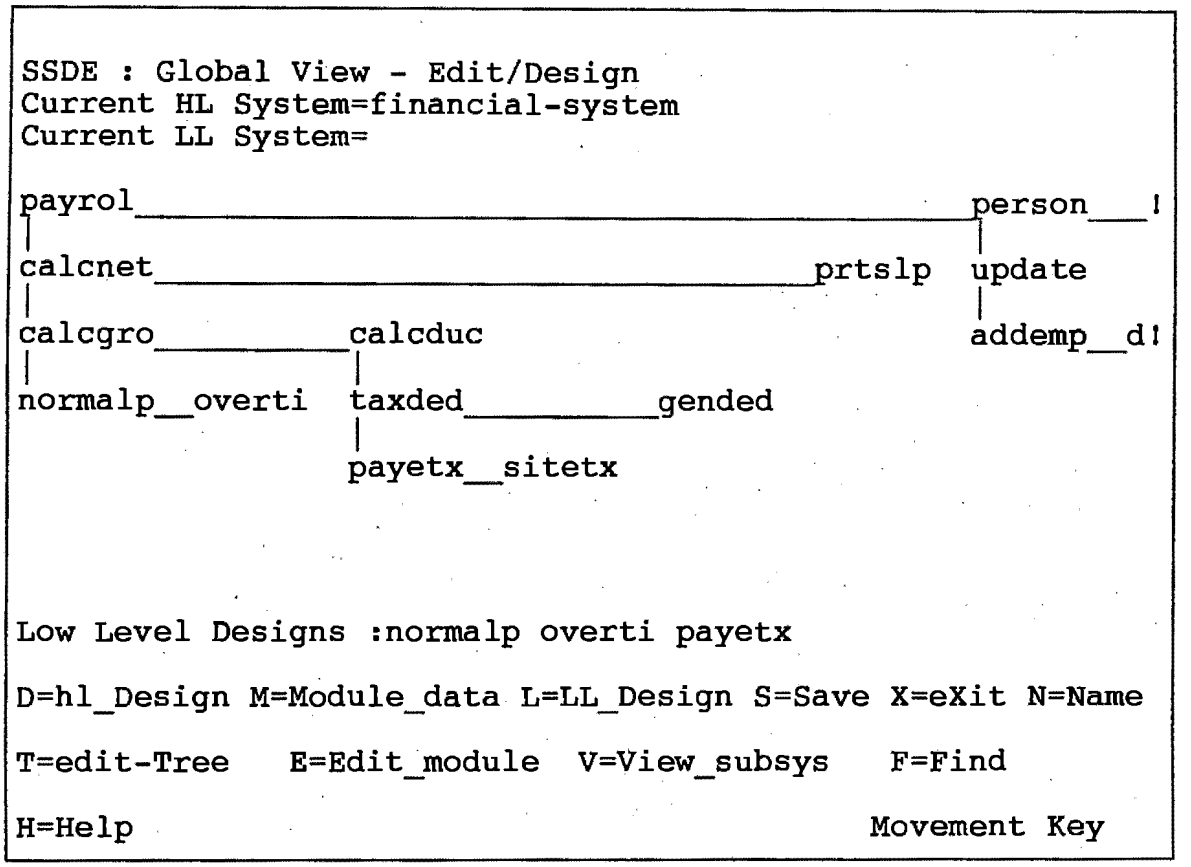
A particular variable may be entered and all high-level functional components which have a reference to that variable are listed.

A.4 The Global View of the High-Level Design.

Control is immediately passed to this facility (i.e the Global View) as soon as a box has been created on the HIPO/Structure chart (local design view).

This is done to enable you to view your design done so far and in order to assist you in designing the next part of the high-level design. Other options are also available.
The global view displays all the high-level modules designed to date with some control information and also presents a sub-menu to choose from. The control information consists of current high-level system name, current low-level system name (if any) and a display of the low-level modules which have been designed up to this point (if any).

The global view sub-menu is presented as follows:



A.4.1 Using the sub-menu presented on the Global High-Level View.

This menu makes the following options possible:

Option: D. hl_Design.

This option continues with the high-level component definition task. Detailed information about surrounding components, is also provided.

Option: M. Module_data.

All the relevant data about a module name shown in the hierarchy can be extracted. The module name, number, input and outputs and the general comment are shown.

Option: L. LL_Design.

This will transfer design flow to the low-level definition facility. The low-level design will be associated with one of the lower components in the high-level definition.

Option: S. Save.

The high-level definition is saved in the relevant databases.

Option: X. eXit.

The main menu options are displayed.

Option: N. Name.

The high-level system name is changed.

Option: T. edit-Tree.

The high-level definition can be edited or even designed using this option.

Option: H. Help.

Provides brief and detailed help screens.

Option: E. Edit_module.

The data items which constitute a functional component module may be modified.

Option: V. View_subsys.

A sub-set of the complete hierarchy can be displayed only.

Option: F. Find.
A search process that will locate a module in the hierarchy with a particular name or portion thereof.

A.4.2 Creating a HL Functional Component.

(a) Mandatory Entry.

SSDE will prompt you for a functional component name. Enter an appropriate and meaningful name for the particular functional component being designed.

(b) Optional Entries.

1. Next you will be required to enter the input parameters for this component. Enter these input parameters and separate them with a ":" character.

2. Next you will be required to enter the output parameters for this component. Enter these output parameters and separate them with a ":" character.

3. Enter a number of process steps. Press <enter> on a blank line to bypass or when all steps have been entered.

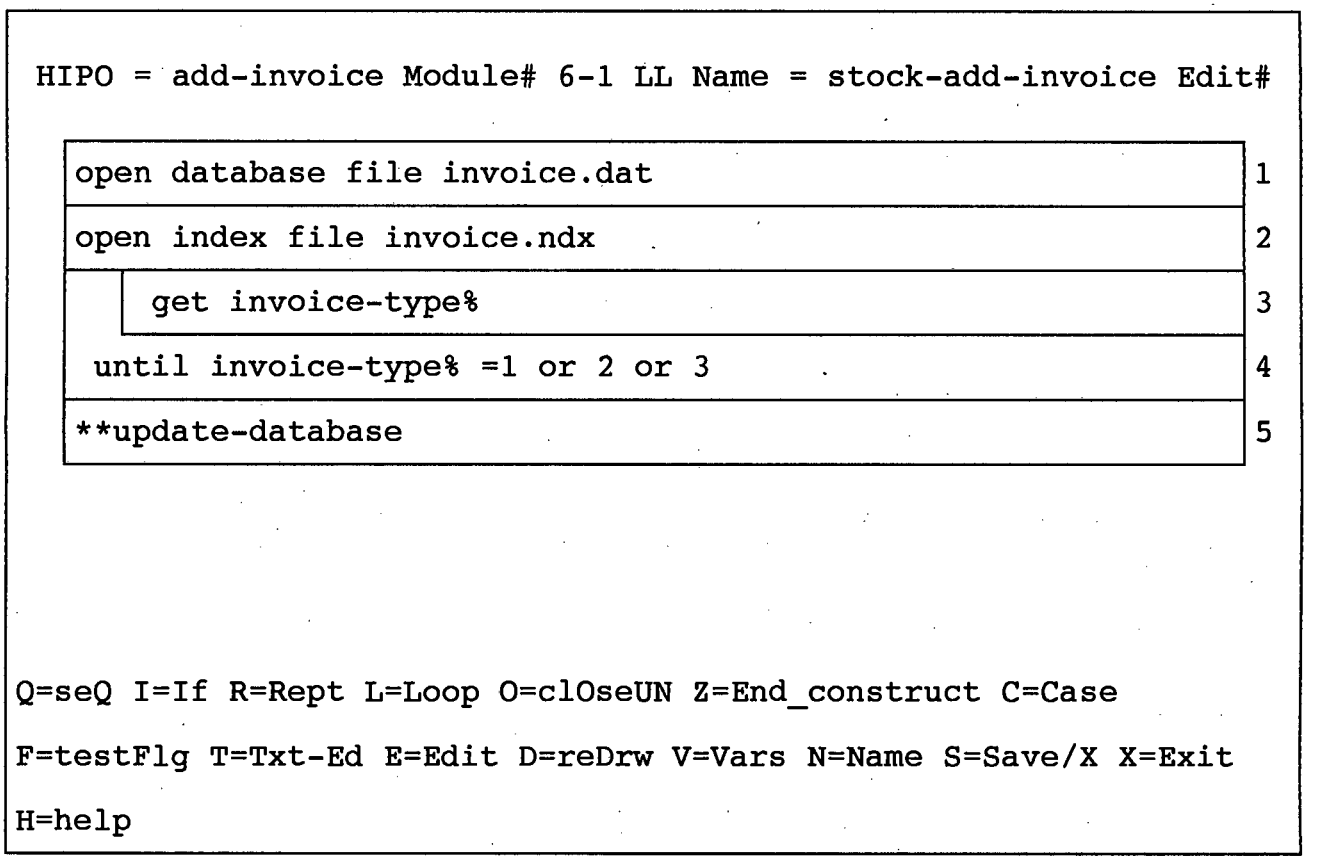
4. Finally you are required to enter any general informal comment about this particular component.

A.5 The Low-Level Design

Upon entering the low level design facility from the global view of the high-level design to date, you are presented with the low-level design environment menu.
All the structures in terms of the Chapin diagramming technique, can now be selected via a choice from the menu and the appropriate text associated with each structure may also be entered.
The structures available are the standard structures used in structured program design.
If the logic at a particular point is very complex, and you wish to postpone its consideration for a later stage, then he can do this in terms of a secondary diagram.

A.5.1 The Sub-Menu on the Low-Level Design Screen.

This menu is presented as follows:



Option: Q. seQ (Sequence).

This option will draw a sequence construct and obtain its text.

Option: I. If (decision)

A decision construct is drawn.

Option: R. Rept (While/Repeat). (Iterative Construct)

This option will draw a WHILE/REPEAT construct and obtain its related text.

Option: L. Loop (Until). (Loop Construct)

This option will draw the start of a UNTIL construct diagram and obtain its text.

Option: O. clOseUN (Terminate UNTIL).

This option will draw a "close" UNTIL construct diagram and obtain the UNTIL condition.

Option: F. TestFlg.

A flag can be set to indicate that this particular low-level designs' code has been generated and tested.

Option: T. Txt_ed.

This option will allow you to alter any existing text which is already displayed on the screen. Enter the edit number alongside the text entry you want changed.

Option: Z. End_construct.

The end of a repetition structure is noted. No structure is drawn.

Option: C. Case.

A CASE construct is drawn.

Option: D. reDrw. (Re-draw LL design)

This option will clear the low-level display on the screen and re-draw the low-level design from the beginning of the diagram.

Option: V. Vars. (Display Symbol table)

This option will display a list of all the variables and their types. See below for full discussion.

Option: N. Name a current Low-Level system.

Give the low-level design in memory a new name.

Option: H. Help.

Obtain brief and detailed help about the options available and their functions.

Option: S. Save/End.

Save the low-level design information in the appropriate databases and return to the main menu.

Option: X. Exit.

Return to the main menu without saving the low-level design. (No low-level design information is lost).

Option: E. Edit.

Re-structure the low-level logic.

A.5.2 The Syntax for Entering Text.

The text entered will be validated in two ways:

* a keyword - all text entered must have a recognized keyword. these keywords are those normally used in a high-level programming language (e.g open, display, get, etc.)

* variables - all variable names must be preceded by a space and must end with either "%" (numeric items) or "$" (alphanumeric items).

The text entry is completely rejected if no recognizable keyword is found. You may enter a blank line however. The latter case is mostly applicable within an IF construct where there is no "else" part or if you wish to enter the text at a later stage.

N.B Entries you do not want validated can be entered by starting such an entry with a "?" character.

APPENDIX B:

EXAMPLE.

B.1 Introduction.

The problem presented in Appendix C is designed using SSDE. The complete high-level definition and a few of the low-level designs where completed including code generation.

B.2 Defining a Data Structure.

The data structure is defined by using the high-level design facility.

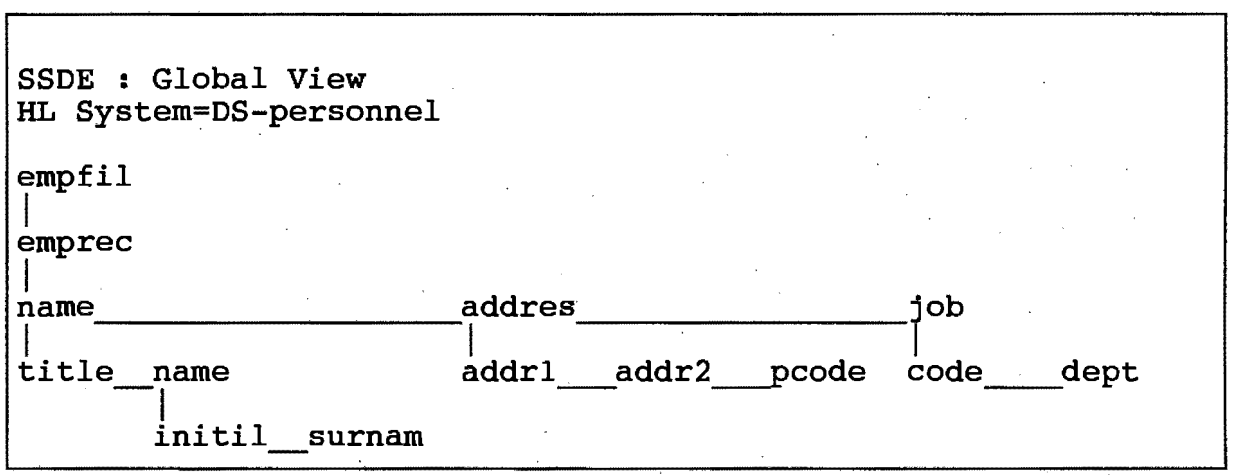


Figure B.1 Global View of High-Level Components

B.3 The High-Level Definition of the System.

The high-level functional components, together with their input/output and comment parts, are designed. During this design activity the relationships between these components are also established.

```
SSDE : Global View
HL System=financial-services

payrol_________credit__________econom__bank
|              |                |
netsal         points__action   withdr__dposit__transf
|
gross__deduct


Low Level Designs :gross, deduct, transf

Functional Components which have been Designed:

System name = financial-services

Components which have been designed :

payrol > Module #=21
         Full name=process payroll
         Input=empname,empn,hrs,prate
         Output=empname,empn,gross,deduct,net
         Comment=calculate payroll for hourly employed workers

credit > Module #=22
         Full name=credit evaluation
         Input=apllnum,age,address,income,job
         Output=apllnum,actionmsg
         Comment=Evaluate an applicants credit worthiness

econon > Module #=23
         Full name=economic indicators
         Input=userid,indicate-type
         Output=userid,indicate-type,reply
         Comment=Supply Information about Economic Indicators

bank   > Module #=24
         Full name=process transactions
         Input=accnum,transaction,amount
         Output=accnum,amount,balance
         Comment=Provide a Banking Service

netsal > Module #=31
         Full name=net salary
         Input=nhrs,ohrs,prate
         Output=net
         Comment=calculate the net salary

points > Module #=32
         Full name=total points
```

```
          Input=age,address,income,job
          Output=pointotal
          Process=get fields
                  award points per field
                  total points
          Comment=Calculate the total points for an applicant

action >  Module #=33
          Full name=action response
          Input=pointotal
          Output=actionmsg
          Comment=Produce the appropriate reply message

withdr >  Module #=34
          Full name=withdrawal
          Input=accnum,transaction,amount
          Output=balance
          Comment=Process a withdrawal

dposit >  Module #=35
          Full name=deposit
          Input=accnum,transaction,amount
          Output=balance
          Comment=Process a Deposit

transf >  Module #=36
          Full name=transfer
          Input=accnum,transaction,amount
          Output=balance
          Comment=Process a inter-account transfer

gross  >  Module #=41
          **coded
          **tested
          Full name=gross salary
          Input=hrs,prate
          Output=gross
          Process=get hours worked
                  get pay rate
                  calculate gross pay
                  accumulate all gross
                  produce pay slip
          Comment=calculate the gross salary

deduct >  Module #=42
          **coded
          **tested
          Full name=deductions
          Input=gross
          Output=deductions
          Process=get gross
                  calculate tax
                  calculate pension deduction
                  calculate medical deduction
```

total deductionas Comment=calculate the total deductions

Figure B.2 Global View of High-Level Components

B.3.1 A Cross-Reference Listing.

The cross-reference listing for "prate" looks as follows:

Variable	Referenced
prate	payrol netsal gross

B.4 The Low-Level Definitions of the Required Logic.

When the logic required to accomplish one of these high-level functional components needs further definition, then a low-level construct diagram is designed.

The low-level design of "gross" is as follows:

```
zero ct
open employee file
while more employees
    get emprec.hrs[1], emprec.hrs[2];
    get emprec.prate
    gross=emprec.hrs[1]*emprrec.prate+emprec.hrs[2]*2*
                                                emprec.rate
    add 1 to ct
    gross2(ct) = gross
    print out gross
close employee file
```

Figure B.3 The Low-Level Design

The variable table associated with this design looks as follows:

Variable	ArrayVar	Field	ArrayFld	Type
ct				int
emprec		hrs	Y	int
emprec		ohrs		int
emprec		prate		int
gross		-		int
gross2	Y	-		int

Figure B.4 The Variable Table

B.5 The High-Level to Low-Level Linkage Check.

This function will determine that all variables specified in a high-level functional component has in fact been referenced in the corresponding low-level design.

The following list will be checked against the variable table above: hrs, prate, gross. (This list is the input and output variables of "gross" as shown above).

The Linkage Check displays the following:

```
    High-Level to Low-Level Linkage Check

Structure Chart Name=gross being compared with
                                financialservices-gross

 Variable          Response

 ct                -
 hrs               input
 prate             input
 gross             output
 gross2            -
```

Figure B.5 The High- to Low-Level Linkage Report

B.6 Code Generation.

For a low-level construction, skeleton Turbo Pascal code is now generated and placed in an ASCII file. The low-level design in figure B.2 together with the variable table B.3 are used to produce the code. The code for the diagram "gross" above, has been generated as follows:

```
Program gross(input,output);
Type
  array1 = array[1..20] of integer; {**check size**}
  rec1 = record
         hrs : array[1..20] of integer; {**check size**}
         prate : integer;
  end;
Var
```

```
  gross2 : array1;
  empr : file of rec1;
  emprrec : rec1;
  ct :integer;
  gross :integer;

{******************program body starts here***************}

Begin
  ct := 0;
  assign(empr,file1);          {**fix**}
  reset(empr);
  while {(there are employees)} do  {**fix**}
  begin
    read(empr,emprrec);
    read(emprrec.hrs[1], emprec.hrs[2]);
    read(emprrec.ohrs);
    read(emprrec.prate);
    gross:=emprrec.hrs[1]*emprrec.prate+emprrec.hrs[2]*2*
                                          emprrec.prate;
    ct := ct + 1;
    gross2[ct]:=gross;
    writeln('Gross=',gross);
  end;
  close(empr);
End.
```

Figure B.6 SSDE Generated Code

The designer can now edit this code which then becomes:

```
Program gross(input,output);
Type
  array1 = array[1..100] of integer;
    rec1 = record
         hrs : array[1..2] of integer;
         prate : integer;
  end;
Var
  gross2 : array1;
  empr : file of rec1;
  emprrec : rec1;
  ct : integer;
  gross :integer;

{******************program body starts here***************}

Begin
  ct:=0;
  assign(empr,'payroll.dat');
  reset(empr);
  while not eof (empr) do
  begin
    read(empr,emprrec);
```

```
    gross:=emprrec.hrs[1]*emprrec.prate+emprrec.hrs[2]*2*
                                                emprrec.prate;
    ct := ct + 1;
    gross2[ct]:=gross;
    writeln('Gross=',gross);
  end;
  close(empr);
End.
```

Figure B.7 SSDE Generated Code Modified

B.7 Writing the Manual.

From the high-level definition of each functional component data is extracted which will form the basis for writing a skeleton manual.

The following skeleton information has been extracted from the high-level design:

```
System : Financial-Services

Module = payrol
         Description: calculate payroll for hourly employed
                      workers
     Input Required:
          empname -
          empn    -
          hrs[1]  -
          hrs[2]  -
          prate   -

     Output Generated:
          empname -
          empn    -
          gross   -
          deduct  -
          net     -
.                                                              .
. similar entries for netsal, gross, and deduct (with process  .
.                                                steps listed  .
.                                                              .
```

```
Module = credit
         Description: Evaluate an applicants credit
                      worthiness
     Input Required:
          apllnum -
          age     -
          address -
          income  -
          job     -

     Output Generated:
          apllnum -
          actionmsg -

. similar entries for points, action .

Module = econom
         Description: Supply Information about Economic
                      Indicators
     Input Required:
          userid -
          indicate type -

     Output Generated:
          userid -
          indicate type -
          reply -


Module = bank
         Description: Provide a Banking Service
     Input Required:
          accnum -
          transaction -
          amount -

     Output Generated:
          accnum -
          amount -
          balance -

. similar entries for withdr, dposit, transf .
```

Figure B.8 The Skeleton Manual Written by SSDE

The above text can now be extended or edited to provide a more comprehensive user manual. Below is an example of the above diagram in a more complete write-up:

System : Financial Services

1. Payrol.
   Description: calculate payroll for hourly employed workers

   Input Required:
   - empname - employee name (30 chars max.)
   - empn - employee number (4 numeric chars)
   - hrs[1] - normal hours worked
   - hrs[2] - overtime hours worked
   - prate - the pay rate for this employee

   Output Generated:
   - empname - see above
   - empn - see above
   - gross - the gross salary earned (with overtime)
   - deduct - the total deductions
   - net - the gross minus deductions

2. Credit.
   Description: Evaluate an applicants credit worthiness

   Input Required:
   - apllnum - applicant number (8 digits)
   - age - applicants age in years (round months)
   - address - number of years at same address
   - income - annual income
   - job - # of years at the same job (round off)

   Output Generated:
   - apllnum - see above
   - actionmsg - indicates the credit limit awarded

3. Econom.
   Description: Supply Information about Economic Indicators

   Input Required:
   - userid - a 8 character user number
   - indicate type - code for indicator required

   Output Generated:
   - userid - see above
   - indicate type - see above
   - reply - the value of the requested indicator

4. Bank.
   Description: Provide a Banking Service

```
Input Required:
     accnum - 9 digit account number
     transaction - the transaction code
     amount - the amount involved with this transaction

Output Generated:
     accnum - see above
     amount - see above
     balance - the balance remaining in the account
```

Figure B.9 The Skeleton Manual Extended

B.8 Summary.

SSDE is a useful environment for high-level conceptual designing as well as for low-level construct definition. For low-level constructs code can be generated and a skeleton manual is written.

APPENDIX C:

USER SURVEY RESULTS

C.1 Problem Addressed.

C.1.1 Problem Statement.

The following problem statement was presented to the evaluators:

Design a financial information system which will provide the following services for its users.

(a) A Payroll Service.

Design a payroll system. The employee input record consists of an employee name, employee number, normal-hours worked, overtime-hours worked and pay-rate. Assume the pay-rate for overtime is double the normal rate. Tax deductions are 10% if the wage is less than or equal to R1000, otherwise the tax rate is 20%. Medical aid contributions is 2.5% if wage is less then or equal to R1000, otherwise it is 5%. Output the employee name, employee#, hours worked, net, deductions and gross.

(b) Process a Credit Card Application.

A credit card company bases its evaluation of card applicants on four factors: the applicants age, how long the applicant has lived at his/her current address, the annual income of the applicant and how long the applicant has been working at the

same job. For each factor points are added to a total as follows:

Factor	Value	Points Added
Age	20 and under	-10
	21-30	0
	31-50	20
	Over 50	25
At current address	Less than 1 year	-5
	1-3 years	5
	4-8 years	12
	9 or more years	20
Annual income	R15000 or less	0
	R15001-25000	12
	R25001-40000	24
	Over R40000	30
At same job	Less then 2 years	-4
	2-4 years	8
	More than 4 years	15

On the basis of the point total, the following action is taken by the company:

Points	Action
-19 to 20	No card issued
21 to 35	Card issued with R500 credit limit
36 to 60	Card issued with R2000 credit limit
61 to 90	Card issued with R5000 credit limit

Design a system which will accept an applicants number, age, years at current address, annual income and years at the same job. The system should evaluate the applicant's credit worthiness and produce an appropriate phrase describing the companys' action.

(c) Display Important Economic Indicators.

The input will be the user-id and the indicator type. The output is the user-id, indicator type and the reply message. The following indicators can be requested:
indicate=1 means the latest gold price (goldprice)
indicate=2 means the latest rand-dollar exchange (exchangerate)
indicate=3 means the latest inflation rate (inflationrate)
indicate=4 means the latest prime rate (primerate)
indicate=5 means the BA rate (barate)
indicate=6 means industrial index (indusindex)
indicate=7 means the long bond rate (lgbondrate)

These indicators are kept in a database file and the field name is given in parenthesis.

(d) A Small Banking System.

Assume that the bank's clients can have up to two accounts. A cheque account and a savings account. Input consists of an account number, transaction type and amount.
Transaction type can have the following values:
transtype=1 means a withdrawal from cheque account
transtype=2 means a withdrawal from savings account
transtype=3 means a deposit to cheque account
transtype=4 means a deposit to savings account
transtype=5 means transfer from savings to cheque account
transtype=6 means transfer from cheque to savings account
Output the account number, amount, transaction amount and the account balance.

C.2 Evaluation Questionnaire.

Note that redundant questions were deliberately included to check on respondents.

```
Scale :1=not at all
       2=very little
       3=average
       4=quite a lot
       5=very much indeed
```

Circle the number of your choice according to the above scale.

General:

In the questionnaire each question had the following scale below it:

1 2 3 4 5

1. To what extent was your design productivity improved:

2. To what extent was it easy to learn how to use SSDE:

3. To what extent do you prefer this automated design above a manual design method:

4. To what extent did you use less time to complete your design (as compared to a manual method):

5. To what extent was it in general easy to use the whole environment:

6. To what extent was it easy to move between the various parts of the system:

7. To what extent could you follow your own design methodology:

8. To what extent did SSDE force you to follow a specific methodology:

9. To what extent could you concentrate on the actual design, rather than on the workings of SSDE:

10. To what extent was SSDE flexible:

11. To what extent was SSDE rigid:

12. To what extent were the menus well organized:

13. To what extent were the database access times satisfactory:

14. In totality, to what extent would you prefer an automated environment such as SSDE:

High-Level Design:

15. To what extent were you previously familiar with the Structure Chart Technique:

16. To what extent did you use the edit facility of the High-Level Design:

17. To what extent was the High-Level to Low-Level Verify useful:

18. To what extent was the Low-Level Code Generation useful:

19. To what extent was it easy to edit the High-Level Design:

20. To what extent was the High-Level Diagrams readable:

21. To what extent did the Global High-Level view contribute towards your understanding of the hierarchical structure:

Low-Level Design:

22. To what extent were you previously familiar with the Nassi-Shneiderman / Chapin Diagram Technique:

23. To what extent did you use the edit facility of the Low-Level Design:

24. To what extent did you use the secondary diagram facility of the Low-Level Design:

25. To what extent did you use the Low-Level abstraction facility:

26. To what extent was it easy to edit the Low-Level Design:

27. To what extent was the Low-Level Diagrams readable:

28. To what extent was your executable code generated:

29. To what extent did your executable code have fewer bugs:

30. To what extent did SSDE improve your debugging process:

31. To what extent was the Low-Level Diagrams readable:

32. To what extent did the Low-Level Diagrams contribute towards your understanding of the detail logic:

Written Comment:

33. General Summary Concerning the use of SSDE:

34. Why would you (*prefer / prefer not) to use such an automated environment for your program and system design activity:
(*delete that which is not applicable)

35. Features you found useful (give reasons(s)):

36. Features you would rather not use (give reasons(s)):

37. Features which you would like to see added to SSDE:

C.3 Response Analysis - Rating Answers.

The table below summarizes the ratings chosen by the evaluators for each particular question (there were six evaluators). It should be noted that the help facility had not been implemented at the time.

Question 1: (productivity)

1-none 2-none 3-none 4-2 5-4

Question 2: (learnability)

1-none 2-none 3-5 4-1 5-none

Question 3: (automated design preference)

1-none 2-none 3-none 4-1 5-5

Question 4: (time saved)

1-none 2-none 3-2 4-2 5-2

Question 5: (ease of use)

1-none 2-none 3-2 4-3 5-1

Question 6: (movement within SSDE)

1-none 2-none 3-none 4-3 5-3

Question 7: (own design methodology)

1-none 2-none 3-2 4-4 5-none

Question 8: (SSDE force a methodology)

1-none 2-none 3-2 4-4 5-none

Question 9: (design concentration)

1-none 2-none 3-5 4-1 5-none

Question 10: (flexibility)

1-none 2-none 3-3 4-3 5-none

Question 11: (rigidity)

1-none 2-2 3-3 4-none 5-1

Question 12: (menu organization)

1-none 2-none 3-none 4-2 5-4

Question 13: (database access times)

1-none 2-none 3-none 4-5 5-1

Question 14: (SSDE preference)

1-none 2-none 3-none 4-2 5-4

Question 15: (Structure Chart familiarity)

1-none 2-none 3-none 4-1 5-5

Question 16: (use edit facility)

1-1 2-none 3-3 4-2 5-none

Question 17: (high- to low-level verify useful)

1-none 2-none 3-2 4-4one 5-none

Question 18: (code generation)

1-none 2-none 3-1 4-5 5-none

Question 19: (ease of High-level edit)

1-none 2-none 3-3 4-3 5-none

Question 20: (diagrams readable)

1-none 2-none 3-none 4-3 5-3

Question 21: (contribution towards understanding)

1-none 2-none 3-none 4-2 5-4

Question 22: (Nassi-Shneiderman familiarity)

1-none 2-none 3-none 4-1 5-5

Question 23: (use low-level edit)

1-none 2-none 3-3 4-2 5-1

Question 24: (use secondary diagrams)

1-none 2-none 3-3 4-2 5-1

Question 25: (low-level abstraction)

1-none 2-none 3-3 4-1 5-2

Question 26: (ease of edit for low-level)

1-none 2-none 3-3 4-3 5-none

Question 27: (low-level diagrams readable)

1-none 2-none 3-2 4-2 5-2

Question 28: (code generated)

1-none	2-none	3-4	4-2	5-none

Question 29: (fewer bugs in code)

1-none	2-none	3-6	4-none	5-none

Question 30: (improve debugging)

1-none	2-none	3-4	4-2	5-none

Question 31: (readable diagrams)

1-none	2-none	3-none	4-4	5-2

Question 32: (contribution towards understanding logic)

1-none	2-none	3-none	4-3	5-3

C.4 Response Analysis - Written Answers.

C.4.1 Question 33.

The average replies here were overwhelmingly positive.
Many evaluators said:

* "good idea"
* "great idea"
* saving of time, a major advantage, made possible by SSDE
* a "powerful" design tool
* has "great potential"

* easy to learn
* code generation very helpful
* improved design productivity possible

Because these students are inexperienced in the use of software development tools, their evaluation was done from a narrow perspective. However, their comments indicate that a tool such as SSDE could provide valuable assistance when developing software. (The development of software is a task with which they are familiar.)

C.4.2 Question 34.

All respondents indicated that they would prefer an automated design environment such as SSDE.

They preferred it because:

* it saved design time and effort (fast)
* easy to use
* it generated PASCAL code from the low-level design
* designing in a language-independent manner
* it promotes a structured design methodology
* the databases provided data retention
* high- and low-level designs can be easily edited (re-structured)
* drew all the diagrams automatically

C.4.3 Question 35.

Features which were particularly useful according to the evaluators:

* editing facilities saves time
* code generation
* the database for quick updating, saving and retrieval of systems
* similar systems can designed from previous designs in the database

C.4.4 Question 36.

Features which were they would rather not use were difficult to determine as they have been exposed to SSDE for only a short period of time.

C.4.5 Question 37.

Features that they would like to see added to SSDE:

* a on-line "help" facility in addition to the manual
* generate code in any programming language
* more "user-friendliness" in a few places

C.5 Summary.

It is clear from the evaluation exercise that SSDE is a useful and valuable environment for high-level system definition and low-level program code construction. No adverse comments where recorded. The evaluators found SSDE a refreshing new way to design systems and enjoyed the research exercise. SSDE is an

environment which manifests considerable potential as a framework for system design and code generation.

APPENDIX D:

GLOSSARY.

Algorithm - a set of well defined steps for solving a problem in a finite number of operations.

Application - the user task accomplished with the help of a computer.

Applications Software - the software required to carry out the applications function.

Artificial Intelligence - using computers to solve unstructured problems which normally would only be solved by human intelligence.

Automation - accomplishing a task using computers with little or no human intervention.

CAD/CAM - (Computer-Aided Design/Computer-Aided Manufacturing) - a general term applied to the efforts being made to automate design and manufacturing operations.

CASE - (Computer-Aided Software Engineering) - an automated system development environment consisting of a combination of software tools and methodologies which support the complete software development life-cycle.

Conceptual Model - the definition of a system, component-wise, in broad logical terms which are not computer-related. Interrelationships amongst components included.

Database - a stored collection of data that are needed by organizations and individuals to meet their information processing and retrieval needs.

Data Design - the design of the data structures required by a particular software system.

Data Flow Diagram (DFD) - a graphic representation depicting a network of related components.

Design Methodology - a systematic approach to creating a design, consisting of the ordered application of a specific set of tools, techniques and guidelines.

Documentation - the documents, that describe such things as the system, the programs prepared, and the changes made at later dates.

Engineer - a person who uses knowledge of science and mathematics to design and implement machinery and systems (see Software Engineering).

Engineering - activities performed by an engineer.

Expert System - a software package that uses a knowledge base to answer questions in some problem domain as a human expert would.

Functional Decomposition - a method of designing a system by breaking it down into its components in such a way that the components correspond directly to system functions and sub-functions.

Functional Specification - defining software in terms of the functions it must perform.

Hierarchy - grouping or arranging system elements into a set of successively subordinate / superordinate classes.

Hierarchical Data Structure - a logical approach to structuring data in with a single root data component or "parent" may have subordinate elements or "children" each of which, in turn, may "own" any number of other elements (or none). Each element, except the root, has a single parent.

High-Level Design - a systematic approach to defining, in broad terms, the functional components which show the main tasks that will be accomplished by the system, and their interrelationships.

Implementation - the process of converting detailed software design into program code. Also, the assembling of various components to make a system work.

Life Cycle - the sequence of stages involved in software development (generally from requirements analysis to maintenance).

Low-Level Design - a systematic approach to defining, in deliberate detail, the bottom level operations necessary to perform a specific task.

Maintenance - the task of changing an operational software system. This involves correcting, updating and enhancing.

Methodology - a systematic plan achieving a series of objectives.

Modelling - simulation of a system by manipulating various variables and parameters.

Modularization - the splitting of a software system into smaller manageable sections (called modules) to ease the task of designing, coding, etc.

Project Management - a systematic approach for completing a project. This involves analyzing, organizing, documenting, etc.

Prototype - an initial solution; experience with this is used in building the final working solution.

Prototyping - the process of developing a mock-up of a system to give the developers an opportunity to review their requirements.

Requirements Analysis - analyzing a user's requirements and converting them into a statement of needs (prior to specification).

Software - a collection of one or more programs which enables the hardware to accomplish specific tasks.

Software Development Life Cycle - see Life Cycle.

Software Engineering - the development and use of systematic strategies for the production of good quality software. This process notes constraints such as budgets and timescales.

Software Development Environment - an integrated collection of software tools which assist in achieving the various stages of the software development life cycle.

Structured Design - a disciplined approach to software design that adheres to a specified set of rules based on the principles such as top-down design, stepwise refinement and data flow analysis.

Structured Programming - an approach or discipline used in the design and coding of computer programs.

Systems Design - the process of defining the overall architecture of a software system.

Systems Software - software which manages the hardware resources, e,g an operating system.

Testing - the process of executing software with test data to check if it satisfies its specification.

Tools - Aids which assist in performing any part(s) of the software development life cycle.

Validation - the process of checking a specific piece of life cycle notation and the conversion from one piece to another.

Verification - the process of proving that software or a program meets its specification.

APPENDIX E:

CASE Products.

Product	Type	Supplier / Sponsor
Aims Plus	Workstation-based automated development system for Wang equipment	Aims Plus Inc
Analyst / Designer Toolkit	Workstation-based automated development system	Yourdon Inc
Application Factory	Integrated application development and maintenance for DEC VAX	Cortex Corp
CorVision	Workstation-based automated development system	Cortex Corp
DesignAid	Workstation-based automated development system	Nastec Corp
Design Machine	Workstation-based automated development system	Ken Orr & Ass.
Developer Workstation	Workstation-based automated development system	DBMS Inc.
Excelerator	PC-based workbench for systems analysis, design, documentation	Index Technology Corp.
Information Engineering Workbench	AI-based expert development system	KnowledgeWare
Life Cycle Manager	Project Manager workbench and analysis tool kit	Nastec Corp
Life-cycle Productivity Management	A collection of tools for IBM PC's and mainframes	American Systems Inc
Maestro	Software Engineering Tools	Softlab Inc
ManagerView	Workstation-based	Manager Software

	automated development system	Products
Micro-Caps	COBOL development on Convergent Technologies equipment	Software Research Inc
Micro-CICS	Creates, tests CICS programs on AT	Unicorn Systems Co.
MultiPro	AT-, XT-based development project control	Cap Gemini Software
PacBase	Generates production systems from specs	CGI Systems Inc
PC/Hibol	Develops CICS programs for mainframes, pc's; downloads from mainframe to pc	Matterhorn Inc
Progress	Automated development system/4GL for various micro's	Data Language Corp.
ProMod	Integrated software development tool for DEC VAX and IBM PC's	Promod Inc
Teamwork/SA	Automated development system for Apollo, DEC and IBM	Cadre Technologies Inc
Multi/CAM	Workstation-based automated development system	AGS Mgt. Systems Inc
Information Engineering Facility	Workstation-based automated development system [Stamps :1987 p56-57	Texas Instruments Inc
PCSA	Computer-Aided structured analysis tool [Communications of ACM :1987 pA-1]	Cadre Technologies
The Computer Aided Development and Evaluation System (CADES)	Supports the system development life cycle	ICL (UK)
The ADA Program Support Environment (APSE)	Implementation of complex real-time software	U.S Department o Defense

Naval Standard Software Engineering Environment (NSSEE)	Support for software development life cycle [Hoffnagle & Beregi :1985 p109]	U.S Navy
Software Requirements Engineering Methodology (SREM)	Formalize and automate software requirements	TRW Huntsville Laboratory
Systems Requirements Engineering Methodology (SYSREM)	Formalize and automate software requirements	TRW Huntsville Laboratory
Technology for the Automated Generation of Systems (TAGS)	Create system or software specification	Teledyne Brown Engineering Inc.
ISDOS System Methodology Encyclopedia Manager [Computer :1985 p36-70]	Prototyping -	
Software Technology for Adaptable, Reliable Systems (STARS)	Developing of Computer Software and Software Reuse	U.S DoD
Strategic Computing	Developing of Software and incorporating AI techniques	U.S Department of Defense
A Software Documentation Support Environment (SODOS) [Horowitz :1986 p1076]	Manages software documentation for developing software	-
Gandalf Prototype	Software Development Environment	-
GNOME	Programming Environment	Carnegie-Mellon University
SMILE (Incorporating ALOEGEN, ARL and DBGEN) [Haberman et al :1986 p1120]	Internal Development Tool	-
Classic/AL	Application development without programmers	Goal Systems [Datamation :1987
PSL/PSA	Automating Requirements Engineering :Systems Analysis & Design	ISDOS [Computer 1985 April p115]
Structured Architect	Automates Analysis and Graphics for SA	ISDOS [Computer 1985 April p117]
unknown	Software Development	SofTech & Mitre

	Environment: SADT & Ada	[Ross :1985 p33]
TAGS	Software Engineering [Sievert et al :1985 p57]	Teledyne
Knowledge WorkBench	Application Development Environment [Computer :April 1986 p114]	Silogic
Information Engineering WorkBench [Computer :April 1986 p115]	Systems Development	KnowledgeWare
PECAN	Program Development System [Reiss :1985 p276]	Brown Univ.
Rn	Environment for Developing large Fortran Programs. [Carle et al :1987 p75]	Rice Univ.
Interactive Development Environments (IDE) [Computer :1987 p22]	Computer Aided Software	IDE
SADT	Documenting Technique	SoftTech [Ross :1977/85]
Gandalf	Support SDLC	[Haberman et al :1985]
Genesis	Support SDLC	[Ramamoorthy et al :1985]
Saga	Support SDLC	[Kirlis et al :1985])
Graphics-based Programming Support System [Frei et al:1978]	Automate Nassi-Scheiderman	
Graphical Interactive Monitor	Automate Nassi Scheiderman	[Clark et al: 1983]

REFERENCES

Andersen, Arthur & Co., 1988, "FOUNDATION: Three Year Development Plan", Advertisement Document, August 1988

Archibald J.L, Leavenworth B.M & Power L.R, 1983, "Abstract Design and Program Translator: New tools for Software Design", IBM Systems Journal, Vol. 22, No. 3, 1983, pp170-187

Aktas A.Z, 1987, Structured Analysis and Design of Information Systems, Prentice-Hall, 1987

Aron J.D, 1969, "The Superprogrammer Project", Software Engineering Techniques, NATO Scientific Affairs Division, 1969

Bϕ K, 1982, "Guest Editor's Introduction :Human-Computer Interaction", Computer, Vol. 15, No. 11, Nov 1982, pp9-11

Baecker R.M, 1975, "Two Systems Which Produce Animated Representations of the Execution of Computer Programs", ACM Sigcse Bulletin, Vol. 7, No. 1, Feb 1975, pp158-167

Baker F.T, 1972, "Chief Programmer Team Management of Production Programming", IBM Systems Journal, January 1972

Balzer R.M, 1969, "EXDAMS - Extendable Debugging and Monitoring System", AFIPS Joint Spring Computer Conf., AFIPS Press, Palo Alto, Calif., 1969, pp567-580

Basili V.R, Katz E.E, Panlilio-Yap N.M, Ramsay C.L & Chang S, 1985, "Characterization of an Ada Software Development", Computer, Vol. 18, No. 9, September 1985, pp53-65

Boehm B.W, 1977, "Software Reliability : Measurement and Management", International Software Management Conference, London, 1977

Boehm B.W, 1987, "Improving Software Productivity", Computer, Vol. 20, No. 9, September 1987, pp43-57

Booch G, 1986, "Object-Oriented Development", IEEE Transactions on Software Engineering, Vol. Se-12, No. 2, February 1986, pp???

Borland, 1985, TurboPascal - Database Toolbox, Reference Manual 1985

Borland, 1986, TurboPascal - The Ultimate Pascal Development Environment, Reference Manual Version 3.0, 1986

Bornman C.H, 1989, "Introduction to CASE", SAIEE - CASE Workshop, unpublished, University of the Witwatersrand, January 1989

Bromberg H, 1984, "In Search of Productivity", Datamation, August 1984, pp74-76

Brooks F.P, 1987, "No Silver Bullet : Essence and Accidents of Software Engineering", Computer, April 1987, pp10-19

Brown G.P, Herot C.F, Kramlich D.A & Souza P, 1985, "Program Visualization: Graphical Support for Software Development", Computer, Vol. 18, No. 8, Aug 1985, pp27-35

Brown M.H & Sedgewick R, 1985, "Techniques for Algorithm Animation", IEEE Software, Vol. 2 No. 1, January 1985, pp28-39

Carle A, Cooper K, Hood R, Kennedy K, Torczon L & Warren S, 1987, "A Practical Environment for Scientific Programming", Computer, Vol. 20, No. 11, 1987, pp75-89

Carlyle R.C, 1987, "High Cost, Lack of Standards is Slowing Pace of CASE", Datamation, Aug 15, 1987, pp23-24

Carter HW, 1986, "Computer-Aided Design of Intergrated Circuits", Computer, Vol. 19, No. 4, 1986, pp19-36

Cerveny R.P, Garrity E, Hunt R, Kirs P, Saunders G & Sipior J, 1987, "Why Software Prototyping Works", Datamation, Aug 15, 1987, pp97-103

Chapin N, 1974, "New Formats for Flowcharts", Software-Practice and Experience, No. 4, 1974, pp341-357

Clark B.E.J & Robinson S.K, 1983, "A Graphically Interactive Program Monitor" Computer Journal, Vol. 26 No. 3, pp235-238, 1983

Communications of the ACM, 1987, Advertisement, Communications of the ACM, January 1987, Vol. 30 No. 1, pA-1

Computerworld, 1983, "Productivity Aid Boosts Total Plants' Access Time", Computerworld, Vol. 17 No. 37, September 12, 1983

Computing SA, 1989a, "CASE Tools are a Key to Efficient Organizations", Computing SA, Vol. 9, No. 14, April 10, 1989

Computing SA, 1989b, "Methodologies: New Approach to Software Developments", Computing SA, Vol. 9, No. 14, April 10, 1989

Connor M.F, 1980, "Structured Analysis and Design Technique", SoftTech Inc, May 1980

Computer, 1985, Computer, Vol. 18 No. 4, April 1985

Computer, 1986, Computer, Vol. 19 No. 4, April 1986

Computer, 1987, Computer, Vol. 20 No. 10, Oct 1987

Dart S, Ellison R, Feiler P, Habermann A, 1987, "Software Developemnt Environments", Computer, Vol. 20, No. 11, Nov 1987, pp18-28

Datamation, 1987, Advertisement, Datamation, OEM Edition, June 15,1987, pp119

Datamation, 1989, New Products Section - Software, Datamation, Jan 1, 1989, p71]

David E.E & Fraser A.G, 1969, "Software Engineering", Panel Discussion - Brussels (NATO), 1969

Davis W.S, 1986, Fundemental Computer Concepts, Addison-Wesley, 1986

DeMarco T, 1981, Structured Analysis and Systems Specifications, Yourdon Inc, 1981

Dhar V & Jarke M, 1988, "Dependency Directed Reasoning and Learning in Systems Maintenance Support", IEEE Transactions on Software Engineering, Vol. 14 No. 2, Feb 1988, pp211-227

Dijkstra E.W, 1969, "Structured Programming", Software Engineering Techniques NATO Scientific Affairs Division, 1969

Donaldson J, 1973, "Structured Programming", Datamation, December 1973

Dionne M.S & Mackworth A.K, 1978, "ANTICS: A System for Animating LISP Programs", Computer Graphics and Image Processing, Vol. 7, No. 1, 1978, pp105-119

Draper S.W & Norman D.A, 1985, "Software Engineering for User Interfaces", IEEE Transactions on Software Engineering, Vol. SE-11 No. 3, March 1985, pp252-268

Flaherty M.J, 1985, "Programming Process Productivity Measurement System for the IBM/370", IBM Systems Journal, Vol. 24, No. 2, June 1985, pp168-175

Frei H.P Weller D.L William R, 1978, "A Graphics-based Programming Support System", Proc. ACM SIGGRAPH Conference, August 1978, pp43-49

Frenkel K, 1985, "Toward Automating the Software-Development Cycle", Communications of the ACM, Vol. 28, No. 6, June 1985, pp578-589

Galley S.W & Goldberg R.P, 1974, "Software Debugging: The Virtual Machine Approach", Proc. ACM Annual Conf., ACM New York, 1974, pp395-401

Gibson M.L, 1989, "The CASE Philosophy", Byte, April 1989, pp209-218

Gilbert P, 1983, Software Design and Development, SRA Computer Science Series, 1983

Glaser S, 1983, "Application Generators : Automating the Art of Programming", Mini System Vol. 16 No. 6, May 1983

Goguen J & Moriconi M, 1987, "Formalization in Programming Environments", Computer, Vol. 20, No. 11, Nov 1987, pp55-64

Goldberg R, 1986, "Software Engineering : An Emerging Discipline", IBM Systems Journal, 1986, pp334-353

Gomaa H, 1986, "Software Development of Real-Time Systems", Communications of the ACM, July 1986, pp657-668

Grafton R.B & Ichikawa T, 1985, "Guest Editors Introduction : Visual Programming", Computer, August 1985, pp6-9

Graham R.M, 1969, "Software Engineering", Panel discussion - Brussels (NATO), 1969

Gutz S, A.I Wasserman & M.J Spier, 1981, "Personal Development Systems for the Professional Programmer", IEEE Computer, Vol. 14, No. 4, April 1981, pp45-53

Haberman A.N et al, 1985, Special Issue on Gandalf Project, J. Systems and Software, Vol. 5, No. 2, May 1985

Haberman A.N & Notkin D, 1986, "Gandalf: Software Development Environments" IEEE Transactions on Software Engineering, Vol. SE-12 No. 12, December 1986, pp1117-1127

Hartzband D.J & Maryanski F.J, 1985, "Enhancing Knowledge Representation in Engineering Databases", Computer, Vol. 18, No. 9, September 1985, pp39-48

Henderson P, 1987, ed. Proc. ACM SIGSoft/SIGPlan Software Engineering Symp. on "Practical Software Development Environments", Apr 1984, Appeared as joint issue of SIGPlan Notices, ACM, May 1984

Henderson P, 1984, ed. Proc. ACM SIGSoft/SIGPlan Software Engineering Symp. on "Practical Software Development Environments", Dec 1986, Appeared as joint issue of SIGPlan Notices, ACM, Jan 1987

Henderson P & Notkin D, 1987, "Integrated Design and Programming Environments", Computer, Vol. 20, No. 11, Nov 1987, pp12-16

Hewlett-Packard, 1987, "HP Teamwork", Hewlitt-Packard Publication, December 1987

Hewlett-Packard, 1988, "Hewlitt-Packard CASE Software Catalog", Hewlitt-Packard Publication, November 1988

Hoffnagle G.F & Beregi W.E, 1985, "Automating the Software Development Process", IBM Systems Journal, Vol. 24 No. 2, 1985, pp102-120

Horowitz E & Williamson R.C, 1986, "SODOS: A Software Documentation Support Environment - Its use", IEEE Transactions on Software Engineering, Vol. SE-12 No. 11, November 1986, pp1076-1087

IBM HIPO, 1974, "A Design Aid and Documentation Technique", (GC20-185D), White Plains, NY, IBM Corp., 1974

Ichikawa T & Hirakawa M, 1986, "ARES: A Relational Database with the Capability of Performing Flexible Interpretation of Queries", IEEE Transactions on Software Engineering, May 1986, Vol. SE-12 No. 5, pp624-634

Karimi J & Konsynski B.R, 1988, "An Automated Software Design Assistant", IEEE Transactions on Software Engineering, Feb 1988, Vol. 14 No. 2, pp194-210

Keren R, 1983, "The Most Important Software in the Business World", Software News, Vol. 3 No. 12, December 1983

Kirlis P.A et al, 1985, "The SAGA Approach to Large Program Development in an Integrated Modular Environment", Proc. GTE Workshop on Software Engineering Environments for Programming-in-the-Large, June 1985

Knowlton K.C, 1966, L6: Bell Telephone Laboratories Low-Level Linked List Language (films), Bell Telephone Lab., Murray Hill, New Jersey, 1966

Kowalski R, 1984, "AI and Software Engineering", Datamation, November 1984, pp92-102

Ledbetter L & Cox B, 1985, "Software-IC's", Byte, June 1985, pp307-316

Leavitt D, 1987, "Integrated Software Tools for Today", Datamation, July 1, 1987, pp48-52

Madhavji N.H, "Fragtypes: A Basis for Programming Environments",

IEEE Transactions on Software Engineering, Jan 1988, Vol. 14 No. 1, pp85-97

Manna Z, 1974, Mathematical Theory of Computation, McGraw-Hill, 1974

Martin J, 1982, Application Development Without Programmers, Prentice-Hall, 1982

Martin J & McClure C, 1985a, Structured Techniques for Computing, Prentice-Hall, 1985

Martin J & McClure C, 1985b, Action Diagrams, Prentice-Hall, 1985

Martin J & McClure C, 1985c, Diagramming Techniques for Analysts and Programmers, Prentice-Hall, 1985

Mathis R.F, 1986, "The Last 10 Percent", IEEE Transactions on Software Engineering, June 1986, Vol. SE-12 No. 6, pp705-712

McCracken D.D, 1973, "Revolution in Programming", Datamation, December 1973, pp50-52

McClure C, 1989, "The CASE Experience", Byte, April 1989, pp235-246

McWilliams G, 1988, "Users See a CASE Advantage in Reverse Engineering Tools", Datamation, February 1, 1988 pp30-36

Moran T.P, 1981, "Guest Editor's Introduction :An Applied Psychology of the User", Computing Surveys, Vol. 13, No. 1, March 1981, pp1-11

Morrissey J.H & Wu L.S-Y, 1979, "Software Engineering : An Economic Perspective", Proceedings of the 8th International Conference on Software Engineering, 1979, pp412-422

Moto-oka T, 1981, "Preliminary Report on Fifth Generation Computer Systems", keynote speech, Proc. Int'l Conf. 5th Generation Computer Systems, Tokyo, Japan, 1981

Nassi I & Shneiderman B, 1973, "Flowchart Techniques for Structured Programming", ACM SIGPLAN Notices, Vol. 8 No. 8, August 1973, pp12-26

NAVMAT Software Engineering Group, 1982, "Software Engineering Environment for the Navy", NAVMAT Software Engineering Group, Report A131941/7, Naval Material Command, Washington (DC), March 1982

Orr K, Gane C, Yourdon E, Chen P & Constatntine L, 1989, "Methodology: The Experts Speak", Byte, April 1989, pp221-223

Osterweil L, 1981, "Software Environment Research: Directions for the Next Five Years", IEEE Computer, Vol. 14, No. 4, April 1981, pp35-43

Parnas D.L, 1985, "Software Aspects of Strategic Defense Systems", Communications of the ACM, December 1985, pp1326-1335

Pomberger G, 1984, Software Engineering and Modula-2, Prentice-Hall International, 1984

Raeder G, 1985, "A Survey of Current Graphical Programming Techniques", Computer, Vol. 8, No. 8, Aug 1985

Ramamoorthy C.V, Prakash A, Tsai W & Usuda Y, 1984, "Software Engineering : Problems and Perspectives", Computer, Vol. 17 No. 10, October 1984

Ramamoorthy C.V et al, 1985, "Genesis - An Integrated Environment for Development and Evolution of Software", COMPSAC, 1985

Ramamoorthy C.V, Shashi S, & Garg V, 1987, "Software Development Support for AI Programs", Computer, Vol. 20 No. 1, Jan 1987, pp30-40

Reiss SP, 1985, "PECAN: Program Development Systems that Support Multiple Views", IEEE Transactions on Software Engineering, Vol. SE-11, No. 3, March 1985, pp276-285

Reps T & Teitelbaum T, 1987, "Langauge Processing in Program Editors", Computer, Vol. 20, No. 11, November 1987, pp29-40

Rogers G.R, 1983, "A simple architecture for consistent application program design", IBM Systems Journal, Vol. 22, No. 3, 1983, pp199-213

Roman G, 1985, "A Taxonomy of Current Issues in Requirements Engineering", Computer, Vol. 18, No. 4, 1985, pp14-22

Ross D.T, 1985, "Applications and Extensions of SADT", Computer, Vol. 18, No. 4, April 1985, pp25-34

Ross D.T, 1977, "Reflections on Requirements", Guest Editorial, IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, 1977, pp2-5

Ross D.T, 1977, "Structured Analysis (SA): A Language for Communicationg Ideas", IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, 1977, pp16-34

Ross D.T, 1977, "Structured Analysis for Requirements Definition", IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, 1977, pp6-15

Rushinek A & Rushinek S.F, 1986, "What Makes Users Happy", Communications of the ACM, July 1986, pp594-598

Rzepka W & Ohno Y, 1985, "Requirements Engineering Environments :Software Tools for Modeling User Needs", Computer, Vol. 18, No. 4, 1985 pp9-12

Scheffer P.A, Stone III A.H & Rzepka W.E, 1985, "A Case Study of SREM", Computer, Vol. 18, No. 4, 1985 pp47-54

Schindler M, 1981, "Technology Forcast : Software", Electron, January 1981

Shatz S.M & Wang J, 1987, "Introduction to Distributed Software Engineering", Computer, Vol. 20, No. 10, 1987, pp23-31

Sievert G & Mizell T, 1985, "Specification-Based Software Engineering with TAGS", Computer, Vol. 18, No. 4, 1985, pp56-65

Simons G, 1987, Introducing Software Engineering, NCC Publications, 1987

Smith D.R, Kotik G.B & Westford S.J, 1985, "Research on Knowledge-Based Environments at Kestrel Institute", IEEE Transactions on Software Engineering", Vol. SE-11, No. 11, November 1985, pp1278-1295

Sommerville I, 1985, Software Engineering, Second Edition, International Computer Science Series, 1985

Stamps D, 1987, "CASE Cranking Out Productivity", Datamation, July 1, 1987, pp55-58

STRADIS/DRAW, Reference Manual, MCAUTO, McDonnell Douglas Automation, Saint Louis, MO 63166

Sulgrove R, 1987, "The IEEE Struggles for a Standard", Datamation interview, Aug 15, 1987, pp24

Tazelar J.M, 1989, "CASE", Byte, April 1989, pp206

Tichy W, 1987, "What can Software Engineers Learn from Artificial Intelligence", Computer, Vol. 20, No. 11, 1987, pp43-54

Walker MG & McGregor J, 1986, "Computer-Aided Engineering for Analog Circuit Design", Computer, Vol. 19, No. 4, 1986, pp100-108

Wilk R, 1989, "The use of CASE Tools at BSW-Data", SAIEE - CASE Workshop, unpublished, University of the Witwatersrand, January 1989

Wirth No., 1971, "Program Development by Stepwise Refinement", Communications of the ACM, No. 4, April 1971, pp221-227

Wasserman A.I, 1980, "Information System Design Methodology", American Society for Information Science, Vol. 31 No. 1, January 1980, pp5-24

Wasserman A.I, 1981, "Automated Development Environments", IEEE Computer, Vol. 14, No. 4, April 1981, pp7-10

Wasserman A.I & Gutz S, 1982, "The Future of Programming", Communications of the ACM, March 1982, pp196-206

Yourdon E, 1975, Techniques of Program Structure and Design, Prentice-Hall, 1975

Yourdon E & Constantine L, 1979, Structured Design : Fundamentals of a Discipline of Computer Program and Systems Design, Prentice-Hall, 1979

Yourdon International, 1988, "The Yourdon Analyst / Designer Toolkit", Yourdon International LTD, Product Advertisement, 1988

Zvegintzov N, 1984, "Front-end Programming Environments", Datamation, August 15, 1984, pp80-88

BIBLIOGRAPHY

Alford M, 1985, "SREM at the Age of Eight: The Distributed Computing Design System", Computer, Vol. 18, No. 4, 1985, pp36-46

Adler M, 1988, "An Algebra for Data Flow Diagram Process Decomposition", IEEE Transactions on Software Engineering, Feb 1988, Vol. 14, No. 2, pp169-183

Balzer R, 1985, "A 15 Year Perspective on Automatic Programming", IEEE Transactions on Software Engineering, Vol. SE-11, No. 11, November 1985, pp1257-1268

Basili V.R, Selby R.W & Hutchens D.H, 1986, "Experimentation in Software Engineering", IEEE Transactions on Software Engineering, Vol. SE-12, No. 7, July 1986, pp733-743

Bergland G.D & Gordon R.D, 1981, "Tutorial: Software Design Strategies", IEEE Computer Society, 1981

Brady S.E, 1987, "Getting a hand on Maintenance costs", Datamation, August 15, 1987, pp62-71

Buxton J.N. & Randell B, 1969, "Software Engineering Techniques", Report on a Conference, Garmisch, 1968, Brussel", NATO Scientific Affairs Division, 1969

Card D.No., MC Garry F.E & Page G.T, 1987, "Evaluating Software Engineering Technologies", IEEE Transactions on Software Engineering, Vol. SE-13, No. 7, July 1987, pp845-851

Connell J & Brice L, 1984, "Rapid Prototyping", Datamation, August 15, 1984, pp93-100

Cordell R.Q, Misra M & Wolfe R.F, 1987, "Advanced Interactive Executive program Development Environment", IBM Systems Journal, Vol. 26, No. 4, 1987, pp361-382

Dickonson B, 1981, Developing Structured Systems, Yourdon Press, 1981

Doerflinger C.W & Basili V.R, 1985, "Monitoring Software Development through Dynamic Variables", IEEE Transactions on Software Engineering, Vol. SE-11, No. 9, September 1985, pp978-985

Dromey R.G, 1988, "Systematic Program Development", IEEE Transactions on Software Engineering, Jan 1988, Vol. 14, No. 1, January 1988, pp12-29

Druding F, 1984, "Looking for the right pond", Datamation, August 15, 1984, pp104-110

Freeman P, 1987, "A Conceptual Analysis of the Draco Approach to Constructing Software Systems", IEEE Transactions on Software Engineering, Vol. SE-13, No. 7, July 1987, pp830-844

Goldberg A.T, 1986, "Knowledge-Based Programming", IEEE Transactions on Software Engineering, Vol. SE-12, No. 7, July 1986, pp752-768

Gomaa H, 1979, "A Comparison of Software Engineering Methods for System Design", Proc. of National Electronics Conference, Chicago, October 1979, pp464-469

Good M.D, Whiteside J.A, Wixon D.R & Jones S.J, 1984, "Building a User-Derived Interface", Communications of the ACM, October 1984, Vol. 27, No. 10, pp1032-1043

Hicyilmaz C, 1985, Jackson System Development Methodology and Application, M.S Thesis, METU, Dept. Computer Eng., March 1985

Jackson M, 1975, Principles of Program Design, Academic Press, 1975

Jackson M, 1983, System Development, Prentice-Hall, 1983

Kant E, 1985, "Understanding and Automating Algorithm Design", IEEE Transactions on Software Engineering, Vol. SE-11, No. 11, November 1985, pp1361-1374

Kemerer C.F, 1987, "An Empirical Validation of Software Cost Estimation Models", Communications of the ACM, Vol. 30, No. 5, May 1987, pp416-429

Ledgard H, 1973, "The Case for Structured Programming", BIT, Vol. 13, 1973, pp45-47

Myers G.J, 1973, "Characteristics of Composite Design", Datamation, Vol. 19, No. 9, September 1973, pp100-102

Myers G.J, 1975, Reliable Software Through Composite Design, Petrocelli, 1975

Myers W, 1987, "Ada: First users - pleased; prospective users - still hesitant", Computer, Vol. 20, No. 3, March 1987, pp68-73

Naur P & Randell B, 1969, "Software Engineering", Report on a Conference, Rome; Brussel", NATO Scientific Affairs Division, 1969

Newton A.R & Sangiovanni-Vincentelli A.L, 1986, "Computer-Aided Design for VLSI Circuits", Computer, Vol. 19, No. 4, 1986, pp38-60

Orr K, 1977, Structured Systems Development, Yourdon Press, 1977

Orr K, 1981, Structured Requirements Definition, K. Orr and Associates, 1981

Parker J, 1978, "A Comparison of Design Methodologies", ACM SIGSOFT, Software Eng. Notes, Vol. 3, No. 4, October 1978, pp12-19

Parnas D.L, 1972, "On Criteria to be Used in Decomposing Systems into Modules", Communications of the ACM, Vol. 15, No. 12, December 1972, pp1053-1058

Parnas D.L, 1979, "Designing Software for Ease of Extension and Contraction", IEEE Transactions on Software Engineering, Vol. SE-5, No. 2, March 1979, pp128-137

Ramamoorthy C.V, Garg V & Prakash A, 1986, "Programming in the Large", IEEE Transactions on Software Engineering, Vol. SE-12, No. 7, July 1986, pp769-783

Ross D.T & Brackett J.W, 1976, "An Approach to Structured Analysis", Computer Decisions, Vol. 8, No. 9, September 1976, pp40-44

Ross D.T, Dickover M.E & McGowan C, 1977, Software Design using SADT, Auerbach Publishers, Inc., Portfoloi No. 35-05-03, 1977

Sneed H.M & Mérey A, 1985, "Automated Software Quality Assurance", IEEE Transactions on Software Engineering, Vol. SE-11, No. 9, September 1985, pp909-916

Stevens W.P, 1981, Using Structured Design, Wiley, 1981

Stevens W.P, 1985, "Using Data Flow for Application Development", Datamation, Vol. 10, No. 6, June 1985, pp267-276

Stevens W.P, Myers G.P & Constantine L.L, 1974, "Structured Design", IBM Systems Journal, Vol. 13, No. 2, 1974, pp115-139

Taylor R.No. & Standish T.A, 1985, "Steps to an Advanced Ada Programming Environment", IEEE Transactions on Software Engineering, Vol. SE-11, No. 3, March 1985, pp302-310

Teichrow D, et al, 1974, "An Introduction to PSL/PSA", IDOS Working Paper No. 86, University of Michigan, 1974

Teichrow D, & Hersley E.A, 1977, "PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems", IEEE Transactions on Software Engineering, SE-3, 1977

Verity J.W, 1985, "Empowering Programmers", Datamation, August 15, 1985, pp68-78

Warnier J.D, 1974, Logical Construction of Programs, Van Nostrand Reinhold, 1974

Warnier J.D, 1981, Logical Construction of Systems, Van Nostrand Reinhold, 1981

White S.M & Lavi J.Z, 1985, "Embedded Computer System Requirements Workshop", Computer, Vol. 18, No. 4, 1985, pp67-70

Yau S.S & Tsai J.J.-P, 1986, "A Survey of Software Design Techniques", IEEE Transactions on Software Engineering", Vol. SE-12, No. 6, June 1986, pp713-721

Yourdon Inc., 1981, Structured Analysis/Design Workshop, Edition 7.2, July 1981