A Survey of Software Design Techniques

STEPHEN S. YAU, FELLOW, IEEE, AND JEFFERY J.-P. TSAI, MEMBER, IEEE

Abstract—**Software design is the process which translates the requirements into a detailed design representation of a software system. Good software design is a key to produce reliable and understandable software. To support software design, many techniques and tools have been developed. In this paper, important techniques for software design, including architectural and detailed design stages, are surveyed. Recent advances in distributed software system design methodologies are also reviewed. To ensure software quality, various design verification and validation techniques are also discussed. In addition, current software metrics and error-resistant software design methodologies are considered. Future research in software design is also discussed.**

Index Terms—**Design methodologies, design representation, design verification and validation, distributed software system design, error-resistant software design, software design technique, software metrics.**

I. INTRODUCTION

IN the 1950's, software development was done by hiring a group of programmers to write a largely undocumented code. This approach often resulted in delay of the software development schedule, required high software maintenance cost, and even unusable software since it could not be properly maintained. Much progress has been made in software design techniques since the mid-1960's. Various systematic approaches have been developed for software design in order to produce reliable and maintainable software systems, especially for large-scale software. These systematic ways to software design are actually a process which translates software requirements into a detailed design representation. The activity involved in this kind of design process consists of two major stages, namely *architectural* and *detailed* design stages [1]–[3].

The input of the design phase is the user's requirement specification. In the requirement specification, a system analyst should be able to specify and plan the externally observable activities and concepts of a software system. Based on the design objects, performance requirements, and planning documentations as well as the specification of very high level product structure and various report forms of user interface layouts, a software designer can perform the architectural and detailed design.

Manuscript received September 30, 1985. This work was supported by the Office of Naval Research under Contract N00014-80-C-0167.

S. S. Yau is with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60201.

J. J.-P. Tsai was with the Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, IL 60201. He is now with the Department of Electrical Engineering and Computer Science, University of Illinois at Chicago, Chicago, IL 60680.

IEEE Log Number 8608193.

The architectural design stage, also called the *preliminary* or *general* design stage, is responsible for decomposing requirement specifications to form a system structure. It emphasizes the module-level system representations which can be evaluated, refined, or modified in the early software development process.

The detailed design stage is responsible for transforming the system structure produced by the architectural design stage into a procedural description of a software system. This stage emphasizes the selection and evaluation of algorithms to implement each module. At this stage, all the details and decisions of each module are well defined and can be easily implemented.

Distributed computing systems represent a wide variety of computer systems, ranging from a centralized star network to a completely decentralized computer system. The design of software for distributed systems is more complicated due to many design constraints and interactions of software components of the system [4]. In this paper, we will also examine current advances in distributed software design methodologies.

The purpose of design verification and validation is to avoid transforming design errors into coding phase. Undetected design errors will cost a lot to correct in testing phase [5]. It is impossible to debug all errors at the code-level, especially for large-scale software. We will briefly survey the techniques for software design verification and validation.

Software metrics are used to predict and control the quality of the product of the software development process [6]. Current software metrics are primarily applied to the code-level and used to measure the sizes of various program objects, the comprehensibility of control structure, and the use of data structure. Those measures are also very important in the software design phase.

To provide high reliability and continuous availability of software system, the topic of error-resistant software designs is very important. Error-resistant software is a piece of software which can resist the adverse effects of errors [7]. In order to accomplish this goal, the program must possess the capabilities to detect errors, to locate and contain the propagation of errors, and to recover from the errors. Current techniques in this area will be discussed.

We will discuss the following aspects of software design in this paper: important design techniques in architectural and detailed design stages, recent advances in distributed software system design methodologies, various design verification and validation techniques, software metrics, error-resistant software designs, and future research in software design.

0098-5589/86/0600-0713$01.00

II. Architectural Design Techniques

In the architectural design stage, requirement specification is transformed into a system structure. Techniques in architectural design stage can be divided into two groups: *process-oriented* and *data-oriented* approaches [2]. The process-oriented design technique emphasizes the process of decomposition and structure in creating a software architecture. The data-oriented technique emphasizes the data design component of software systems and the techniques for deriving the data design. Let us first discuss *process-oriented design techniques.*

A. Process-Oriented Design Techniques

We will discuss the following process-oriented design techniques in this section: modular programming approach, functional decomposition method, data flow design methods, data structure design methods, HIPO, and module interconnection languages.

1) Modular Programming: The basic concepts of modular programming are as follows:

1) Each module implements a single independent function.

2) Each module has a single entry/exit point and the module size is as small as possible.

3) Each module can be designed and coded by different programmers in a team and can be tested separately.

4) The whole system is built by modules.

This approach essentially divides a complex system into several parts, which may be developed by different programmers simultaneously. Each module only performs a single function and the module size is small so that the testing is manageable and thorough. After all modules are coded and tested, then they are integrated and the whole system is tested. In the maintenance phase, we only debug and test the module which cannot function properly. Obvious advantages are easier to write and test programs, and less costly to maintain programs.

2) Functional Decomposition: Functional decomposition is essentially based on divide-and-conquer strategy for solving software design problems. Parnas [8], who attempted to formalize the procedure of functional decomposition using *stepwise refinement technique* [9], presented the concept of *information hiding* as a criterion to decompose a software system. Using this criterion, each module is characterized by a designer's decision. Only certain information of this module is needed by other modules, and communications between modules are through well-defined interfaces. Later, Parnas [10] explored the idea to design software for ease of extension and contraction. The important concept here is the identification of usable subsets as part of the preliminaries to software design. Using this concept, a designer will be able to tailor the software to fit the needs of a large variety of users. Another important idea is to design software as a set of virtual machines instead of the conventional flowchart approach. The advantage of functional decomposition is in its applicability, and its disadvantages are unpredictability and variability [11].

3) Data Flow Design Methods: The data flow design method uses information flow as a driving force for software design process. It uses various mapping functions to transform information flow into software structure. In this section, we will discuss two data flow design methods.

Structured design is based on the concept originated by Yourdon and Constantine [12] and extended by Meyers [13]. It is also called "composite design" or "transform-centered design." It intends to make up the shortcoming in the functional decomposition approach which fails to tell if the decomposition of a function is good or bad [14]. This approach consists of the concepts of structured design, guidelines for composite design and refinement of a design, measurement criteria, and design analysis techniques. The approach is to map the data flow of a problem into its software structure by using some design analysis technique. The structured design procedure can be summarized as follows:

1) Identify the flow of data in the problem and draw a data flow graph.

2) Identify the incoming, central, and outgoing transform elements.

3) Factor the incoming, central, and outgoing transform elements branches to form a hierarchical program structure.

4) Refine and optimize the program structure formed in step 3.

The structured design does aid in the rapid definition and refinement of the data flows. It is widely used because it allows a software designer to express his perception of the design problem in terms of data flow and transformation diagrams and because it also provides means to evaluate a design [14]. This approach is more suitable for a design problem with no data structure available.

Structured Analysis Design Technique (SADT) [15] is based on Structured Analysis (SA) developed by Ross [16]. SA is a graphical language used for explicitly expressing hierarchical and functional relationships among any objects and activities. The system structure represented graphically will highlight interfaces among software components. The analysis of the system structure can be conducted in a top-down, structured, modular, and hierarchical way. Furthermore, the SADT methodology includes management planning and configuration control procedures as well as means for organizing workers into a team and review procedures for allowing work to be communicated. SADT has been successfully applied to a very broad range of application areas. It is particularly effective in the early and late stages of the system development life cycle, and less effectively used for actual detailed design of software system [17]. However, allowing each designer to develop independent diagrams may cause additional difficulties in the review cycle.

4) Data Structure Design Methods: Data structure design methodologies emphasize the structure of a problem. There are two different data structure design methodologies, which were developed independently by Jackson [18] and Warnier [19] and are used to construct architectural and detailed design concurrently. These two approaches use the same representation technique and each

treats the problem of translation of the data hierarchy in its own way.

In *Jackson's Design Methodology* [18], data structure is used as a key to good software design. The basic structure of a software system is determined by the structure of the data it processes, and software is viewed as a mechanism by which input data are transformed into output data. Using input and output structures as a guide, we will be able to design a well-structured software system. The main advantages of Jackson's Methodology are that the quality of the software design results does not depend on a designer's experience, each step in the software design process can be verified, and different designers working independently on the same problem will obtain the same result. However, Jackson's methodology does not tell a designer how to structure data. This methodology can be summarized as follows:

1) Identify and draw the structure of the input data and output data.

2) Draw a program structure by merging these data structure diagrams.

3) Derive and allocate the discrete operations composing the program.

4) Convert the program structure text and program text.

More detailed discussion of this methodology, including case studies, can be found in [20].

Warnier's Methodology [19] is similar to Jackson's Methodology in that it also assumes data structure as the driving force to good software design. However, this approach provides more detailed procedures to software design than Jackson's Methodology.

It uses four kinds of design representations: data organization diagram, logical sequence diagram, instruction list, and pseudocode. Data organization diagram describes input and output data. Logical sequence diagram represents its logical flow. Instruction list contains instructions used in the design. Pseudocode is the final description of the design. Warnier's Methodology can be summarized as follows:

1) Identify all input data of a software system.

2) Organize the input data into a hierarchical form.

3) Define detailed format of each item of input file and note the number of their occurrences.

4) Do steps 1 and 2 for the output data.

5) Specify the details of the program by identifying the types of instructions contained in the design in the following order: read instruction, branch instruction, computations, outputs, and subroutine calls.

6) Using flowchart-like diagrams to display the logical sequence of instructions using special symbols to represent begin-process, end-process, branching, and nesting.

7) Number the elements of the logical sequence and expand it through the instructions listed in step 5.

5) HIPO: HIPO (*Hierarchy, plus Input, Process, Output*) [21], developed by IBM, consists of a set of diagrams and is originally used as a documentation tool. Its main characteristics are as follows:

1) The ability to represent the relationship between input/output data and software process.

2) The ability to decompose a system in a hierarchical way without involving logic details.

3) In HIPO, there are three kinds of diagrams to represent input, process, and output. Process is specified in a central processing box and connects to input and output boxes.

Basic software design procedure using HIPO [14] can be summarized as follows:

1) Start from the highest level of abstraction.

2) Identify the input, output, and process of a system.

3) Connect each input and output to their related process.

4) Document each element of the system using HIPO diagrams.

5) Refine the system diagram by repeating steps 1–4.

HIPO has been widely used and its advantages are that it is easy to learn and use. This technique can also be used in the detailed design, testing, and maintenance phases.

6) Module Interconnection Languages: Programming in a large-scale software system is quite different from programming in a small software system. Most current programming languages do not have the ability to specify interconnections among modules. DeRemer and Kron [22] first addressed this problem and proposed a *module interconnection language* (MIL) to formally specify the module structure and linkage. Besides describing intramodule connections and module attributes, MIL can also serve as a project management tool and a support tool for design process. Other important MIL related research includes the System Version Description and Generation Tool developed by Tichy [23]. The significance of this MIL is that it addressed two important issues: module interface control and system version control. Another MIL is *graph description language* (GDL) developed by Yau and Tsai [24]. The GDL uses graph grammars to formally specify the interconnection of a large-scale software system in order to facilitate the user interface design. They also employed automated reasoning techniques to perform the consistency checking of component interconnection [25]. ADAPT [26] developed by IBM also has the MIL feature and is reported very successful for overall system design and decomposition.

B. Data-Oriented Design Techniques

A *data-oriented design technique* emphasizes the data design components of a software system and the techniques for deriving the data design. The important techniques are object-oriented design technique [27] and conceptual database design methodology [28]. Since these techniques are related to formal specification method, we will first discuss the concept of formal specification methods.

Programs can be built systematically from a formal specification of the data they deal with. Based on formal specification, the techniques of automated programming and proving program correctness may be developed. Liskov and Zilles [29] pointed out the importance and properties of specification techniques for data abstractions. Shaw [30] also introduced the relations between formal

specification, abstract techniques, and program verfication. The mathematical foundation of correctness proofs using *algebraic* and *axiomatic specification* can be found in [31]. These techniques have been used for program hierarchical design and its proof [32]. They have also been used for specification and design of large-scale software systems [33]. Kemmerer [34] used formal specification and verification techniques to detect software design errors in the early stage of the software life cycle in order to guarantee a system to meet its requirement. Hayes [35] applied formal specification techniques to existing software and obtained the benefit of recovering design documentation errors.

Object-oriented design methodology [27], [36], [37] has become popular recently. It is based on the concept of Parnas's information hiding and Guttag's abstract data type. This approach views all the resources, such as data, module, and system as objects. Each object encapsulates a data type inside the set of procedures, which knows how to manipulate the data type. Using this methodology, a designer can create his own abstract type and map the problem domain into these designer-created abstractions instead of mapping the problem domain into the predefined control and data structures of an implementation language. This approach is much more natural than process-oriented design techniques since various different kinds of abstract types may be created in a design process [38]. In this way, a designer can concentrate on the system design without worrying about the details of the data objects used in the system. The feature of complete separation between specification and implementation in Ada and Modula-2 is most suitable for object-oriented design methodology. This methodology can be summarized as follows:

1) Define the problem.

2) Develop an informal strategy which actually is a general sequence of steps that satisfy the system requirement.

3) Formalize the strategy.
   a) Identify objects and their attributes.
   b) Identify operations on the objects.
   c) Establish the interfaces.
   d) Implement the operations.

Another important type of data-oriented approach is *conceptual database design methodology* which can guide a designer in the process of translating data and requirement specification into a database conceptual schema [39]. This approach aims to establish a unified *conceptual model* inheriting richer semantic meaning and using the data astraction concept to facilitate software design. In fact, it is a kind of knowledge representation which ranges from the real-world problem to machine executable code. The software design process can be viewed as a process of building a data model. Borgida, Mylopoulas, and Wong [39] have proposed a method for the construction of conceptual models based on *generalization/specialization* techniques. This method suggests that a designer should start by defining the most *general* naturally occurring classes of objects and events in the domain. Further details of the software system are then introduced in successive iterations by describing subclasses of already presented classes and *specializing* transactions in order to deal with the objects in these classes. However, the relations among conceptual modeling technique, knowledge representation, and abstraction mechanism need more research.

III. Detailed Design Techniques

The modular programming design method discussed before is at the module-level. Here, we discuss a code-level design method *Structured Programming* formalized by Dijkstra [40], [41]. It attempts to address a question "Is the code within a module easier to read, write, and maintain when it is constructed from a fixed number of basic control structures without GOTO?" [42]. As a matter of fact, it was shown that any complex system can be represented using three basic control structures: *sequence*, *iteration*, and *selection* [43]. Gries [44] explored the meaning of structured programming and, furthermore, decomposed it into four closely related research topics: programming methodology, program notation, program correctness, and program verification. The success of structured programming has been exemplified by the practical projects [45]. It is noted that structured programming alone does not improve programming efforts in a large software system design too much, and that in order to achieve maximum reliability improvement and cost saving, it needs to integrate the structured programming technique with a software development methodology, including chief programmer team, top-down development, and development support libraries.

The software design process is a human problem-solving process, similar to other scientific and engineering problem-solving processes. Because of the limited capacity of the human mind for technical detail, there should be certain design representation that can be used as a communication tool and from which source code can be directly and simply derived. Design representation techniques used in detailed design stage can be classified into two categories: graphical representation techniques and language representation techniques. In the following sections we will discuss these detailed design representation techniques.

A. Graphical Representation Techniques

The *flowchart* is the oldest, widely used, and most misunderstood graphical representation technique. It was developed by Von Neumann who intended to use it as a program documentation tool. For each program structure, there is a corresponding graphic chart. A user can use these charts to develop a program. The main advantage of this method is its simplicity and visual aid. However, there are also many disadvantages in using charts [46]:

1) Its notation is inconsistent with the notations used in program specification and implementation.

2) Input to the computer is not straightforward, nor is

output from it, and very few automated tools are supported.

3) There is no effective way to control the level of details contained within each flowchart.

4) Its notations are insufficient to design large-scale software systems.

Nassi–Shneiderman Diagram (N-S Diagram) [47] was developed to support structured programming. There are special rectangular diagrams for *sequence*, *for*, *if*, *case*, and *goto* statements. A program can be described using these rectangular diagrams. A rectangular diagram may contain either a succession of simple statements or other rectangular diagrams structured statements. Its main characteristics [1] are as follows:

1) The functional domain is well defined.

2) It does not allow arbitrary control transfer.

3) It is easy to determine the scope of local and global data.

4) It is easy to represent recursive feature.

To provide visual-aid program development, several software development systems have used N-S diagrams as their graphical representations, such as Graphics-based Programming-Support System [48] and Graphical Interactive Program Monitor [49].

Hierarchical Graphs (HG's) are an extension of ordinary directed graphs and are used for direct modeling of programs and data structures with the desired degree of intuitiveness and formal simplicity [50]. With HG's, a user can easily represent a large-scale software system in a hierarchical way. Yau, Grabow, and Tsai [24], [50] have used HG's and Codd relations to represent the behavior and structure of a large-scale software system. Control flow, data flow, and the structure of program objects are uniformly described by a collection of HG's and their relational equivalence. Graph representation serves as a high-level view of source codes for users and Codd relations are used as a common internal data representation. Based on these representations, a graph-based software development/maintenance environment [51] has been established to integrate different software tools for software creation and modification.

B. Language Representation Technique

Program Design Language (PDL) [52] is a language-based design tool using English-like statements. This technique is based on the fact that if the user-interface is friendly, then the errors made by the designer will be reduced. Its design principle includes iterative refinement procedure, top-down design philosophy, structured design, and automated aids.

IV. Distributed Software System Design Methodologies

Distributed computing systems represent a wide variety of computer systems, ranging from a centralized star network to a completely decentralized computer system. The design of software for distributed systems is more complicated due to many design constraints and interactions of software components of the system. Yau, Yang, and Shatz [4] have presented an approach for developing the design specification for a distributed software system. In their approach, the data and functional components are considered separately and all interactions among the functional components are allowed only through the access of shared resources. A precise description of all aspects of software design, including the data and functional components, their structural relations, and interactions, is developed. They also proposed a method to estimate the performance of the resultant software.

Since attributed grammars have an ability to combine syntactic and semantic approaches to problem solving, Lu, Yau, and Hong [53] have developed a formal methodology using *attributed grammars* for multiprocessing-system software development. This methodology includes the design representation and validation of the software design using attributed grammars. The objective of this methodology is to provide continuity between the development phases so that design analysis and system validation can be automated. The continuity is provided through the use of a model for representing the control and data flows of a software system. The model is used for automated test-case generation and automated validation of the execution of the software system.

Another design representation of distributed software systems is to use *Petri nets* [54] and their modification [55]. In this approach, a model was developed for representing and analyzing the design of a distributed software system. The model enables one to represent the structure and the behavior of a distributed software system at a desired level of details of design, especially communications among processors. Behavioral properties of the design representation can be verified by translating the modified Petri nets into equivalent ordinary Petri nets. The model emphasizes the unified representation of control and data flows, partially ordered software components, hierarchical component structure, abstract data types, data objects, local control, and distributed system state.

V. Design Verification and Validation Technique

In general, the software properties we want to verify and validate are as follows [56]:

1) Completeness: A design is complete if each part of requirement specification is fully developed.

2) Consistency: A design is consistent if no conflict exists among portions of the design.

3) Correctness: A design is correct if its input and output relation can be proved true or false.

4) Traceability: A design is traceable if terms in a design have antecedents in earlier specification.

5) Feasibility: A design is feasible if it can be implemented and maintained so that the expected life-cycle benefits of the system would exceed the expected life-cycle cost of the system.

6) Equivalence: Two designs are equivalent if they have the same behavior.

7) Termination: A design is terminated if it is sufficiently detailed for implementation.

Verification and validation techniques can be manual or automated, simple or mathematical. Manual techniques include: review, walkthroughs, inspections [57], etc. The formal verification techniques using mathematics are based on *inductive assertions*, *function semantics*, and *explicit semantics* [58]. Other mathematical techniques are based on static analysis [59], symbolic execution [60], etc. Examples of automated verification systems are the program verification system developed by the Boyer–Moore theorem prover [61], Standard verifier [62], Gypsy Verification Environment [63], and Edinburgh LCF system [64]. Yau and Wang [65] developed an approach to the verification of communications in distributed system software.

VI. Software Metrics

Software metrics are used to measure and predict software quality. Several software metrics have been developed to measure various kinds of software properties, such as the complexity measure [66], [67], stability measure [68], [69], reliability measure [70], reusability measure [71], etc.

There have been a number of program complexity metrics proposed for various applications, such as to estimate programming effort, understandability, and testability. Among these, the two most well-known complexity metrics are *Halstead's software science metric* [66] and *McCabe's complexity metric* [67]. Halstead's metric is primarily used for estimating programming effort and understandability. The metric E is based on the size of the program in terms of the counts of the operators and operands of the program. It is given by

$$E = [n_1 N_2 (N_1 + N_2) \log_2 (n_1 + n_2)]/(2n_2),$$

where n_1 is the number of distinct operators, n_2 is the number of distinct operands, N_1 is the total number of occurrences of operators, and N_2 is the total number of occurrences of the program.

McCabe's metric is based on graph theory and the decision structure of a program to measure the understandability and testability of the program. The metric $C(G)$ is essentially the cyclomatic number of the program graph $+p$, where p is the number of strongly connected components of the program graph. It is given by

$$C(G) = e - n + 2p$$

where e is the number of edges and n is the number of vertices of the program graph. When the program is structured, $p = 1$, then $C(G)$ can be given as follows:

$$C(G) = \text{the number of predicates} + 1;$$

Stability measure is one of the most important factors affecting software maintainability, modifiability, reusability, and ease of redevelopment. Due to the fact that there are logical and performance ripple effects, there are two types of stability: *logical stability* and *performance stability*. The logical stability of a module is a measure of the resistance to the impact of a modification of a module on other modules in a program in terms of logical considerations. The performance stability of a module is a measure of the resistance to the impact of a modification of a module on other modules in a program in terms of performance considerations. Yau and Collofello [68] established metrics for module and program stability measures. They have also developed normalized stability metrics and designed experiments to validate these metrics. In addition, they have also extended the logical stability metrics to the design phase [69].

Regarding the metrics for program reliability, current approaches are mainly based on the failure history of a program. The major analytical models are *failure count model*, *times between failures model*, *input domain based model*, and *fault seeding model* [70]. In the failure count model, the number of failures in a certain time interval is assumed to follow a stochastic process with a time dependent failure rate. The equations of a stochastic process are usually obtained from experimental observation and are then used to determine the reliability of a software system. In the times between failures model, the time between the current failure occurrence to the next failure occurrence is assumed to follow a statistical distribution and use this distribution to estimate software reliability. The parameters of the statistical distribution are also obtained from the observed values of times between failures. In the input domain based model, a set of test cases which represents the operational usage of a program is first generated and the software reliability is obtained by observing the symbolic execution of the test cases. In the fault seeding model, the software reliability is estimated by inserting a number of seeds of errors into a program and observing the fault count of testing result.

VII. Error-Resistant Software Design Techniques

Error-resistant software, sometimes called *fault-tolerant software*, is a piece of software which can resist the adverse effects of errors [72]. In order to accomplish this goal, the program must possess the capabilities to detect errors, to locate and contain the propagation of errors, and to recover from the errors. Many existing software systems contain errors but are reliable in terms of producing correct results. High reliability and continuous availability, even in the presence of software errors and hardware faults, are very important for many real-time systems.

The capability of error detection, and to a certain extent, the capabilities of error containment and recovery can be implemented using *self-checking software* [73], which contains software redundancy in the program to check the dynamic behavior for proper operation during its execution. When an abnormal behavior is detected, it will be interpreted as an error and the error will be isolated to minimize its propagation. A recovery procedure is then attempted to correct the abnormal behavior. Some transient errors can be corrected by repeating the operation. Permanent damage can be repaired by reconstructing

the value of the mutilated items from redundant information stored in the program or from a safe backup copy stored at an earlier state. Some rollback and retry operations are usually involved. Some errors can be corrected by a duplicated copy of hardware for hardware faults or a different version of algorithm for software errors.

A number of techniques are available to design self-checking software [73]–[75]. However, error containment and error recovery capability largely depend upon the structure of the software system [72], [76], [77]. In general, the approach is to restrict and verify the interactions among various components of the software system and to restructure the system so that a different version of the part of the software system can be executed. Some techniques toward the design of fault tolerant operating systems [78], [79] are also available. The amount of redundancies and increase of program complexity and overhead of error-resistant software must be carefully evaluated for the amount of gain in reliability and availability. Some evaluation experiments for error-resistant software have been conducted [80].

VIII. Conclusion and Further Research

In this paper, important techniques for software design, including architectural and detailed design stages, are surveyed. Recent advances in distributed software system design methodologies are also reviewed. Various design verification and validation techniques are also examined. In addition, software metrics and error-resistant software design methodologies are discussed.

In this section, we will discuss some of the research problems which may have a major impact on the improvement of software design methodologies.

A user requirement specification can be directly translated into suitable design specification using an *intelligent system*, and then the design specification can automatically be transformed to code. To develop an intelligent software design system, we need to study the following problems:

1) What kind of software information is required to support the translation from the requirement phase to design phase?

2) What kind of knowledge representation is most suitable for the design of an intelligent system and its user interface?

3) What kind of automated visual-aid tool is needed to facilitate man–machine interaction?

4) Based on a knowledge representation, what kinds of verification and validation techniques can be used?

Reusable software design methodology is another way to reduce design cost. Using reusable software design methodology, we will be able to use existing design documentation, design specification, design structure, modules, data, and so forth. To make software reusable, standardization of software components is essential. If we can standardize software components, the concept of parts and assembly can be applied to software production as it was applied to automated hardware assembly, and the software design cost can then be reduced. In addition, maintenance cost is the largest single cost item contributing to the high cost of software, and software maintenance is an evolution process which involves different phases in the software life cycle. Design of easily maintainable software is important in reducing software cost.

In software design, *object-oriented design technique* is characterized by data abstraction, program abstraction, and protection domains [36]. This type of technique has advantages over conventional software design methods in better productivity, maintainability, data integrity, and program security. Productivity is addressed by abstraction mechansim which allows the construction of reusable components. Maintainability is achieved by information hiding which hides the effect of modifications inside a software component. Data integrity and program security are achieved by protection domains which define access rights and operations that are available to a specific user. This approach is more suitable to control-dominant or real-time applications which are not well addressed in conventional design techniques [5]. However, its theoretic foundation needs more study and more applications need to be developed.

Design of *distributed system* software is more complicated due to many design constraints and interactions of software components of the system. Currently, various design methodologies and representations for distributed systems have been proposed [53]–[55] and some techniques have been developed to verify the design of distributed software system [53], [65], [81]. However, more research is needed in this area.

Most current metrics are used to measure and predict software quality at the code-level, implementation or maintenance phase. It is obvious that the design process can be much cost effective if it is guided by using certain metrics. A new design methodology called *metric-guided design methodology* has been proposed [82] to reduce the complexity in a software design process. It is a good start and more research is needed in this direction. In addition, most current software metrics are used for nonreal-time system software. It is important to have some meaningful metrics for real-time system software [83].

Although the reconfigurability of software systems is essential for error-resistant software, its effectiveness largely depends upon the quality of the error detection tests used in the system (on both thoroughness and overhead produced). More effective tests need to be developed in order to make error-resistant software practical.

References

[1] R. S. Pressman, *Software Engineering: A Practitioner's Approach*. New York: McGraw-Hill, 1982.

[2] P. Freeman and A. I. Wasserman, Eds., *Tutorial: Software Design Techniques*. Washington, DC: IEEE Computer Society Press, 1983.

[3] R. E. Fairley, *Software Engineering Concepts*. New York: McGraw-Hill, 1985.

[4] S. S. Yau, C. C. Yang, and S. M. Shatz, "An approach to distributed computing system software design," *IEEE Trans. Software Eng.*, vol. SE-7, pp. 427–436, July 1981.

[5] C. V. Ramamoorthy, A. Prakash, W. T. Tsai, and Y. Usuda, "Soft-

ware engineering: Problems and perspectives," *Computer*, pp. 191-209, Oct. 1984.
[6] B. Curtis, "Software metrics," *IEEE Trans. Software Eng.*, vol. SE-9, pp. 637-638, Nov. 1983.
[7] S. S. Yau, "On error-resistant software designs," in *Proc. 1978 Army Numer. Anal. and Comput. Conf.*, 1978, pp. 1-19.
[8] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053-1058, Dec. 1972.
[9] N. Wirth, "Program development by stepwise refinement," *Commun. ACM*, vol. 14, no. 4, pp. 221-227, Apr. 1971.
[10] D. L. Parnas, "Designing software for ease of extension and contraction," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 128-138, Mar. 1979.
[11] G. D. Bergland, "A guided tour of program design methodologies," *Computer*, vol. 14, no. 10, pp. 13-37, Oct. 1981.
[12] E. Yourdon and L. L. Constantine, *Structured Design*. Englewood Cliffs, NJ: Prentice-Hall, 1979.
[13] G. J. Meyers, *Composite/Structure Design*. New York: Van Nostrand, 1978.
[14] L. J. Peters, *Software Design: Methods and Techniques*. New York: Yourdon, 1981.
[15] D. T. Ross and K. E. Schoman, Jr., "Structured analysis for requirements definition," *IEEE Trans. Software Eng.*, vol. SE-3, no. 1, pp. 69-84, Jan. 1977.
[16] D. T. Ross, "Structure analysis (SA): A language for communicating ideas," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 16-34, Jan. 1977.
[17] ——, "Applications and extensions of SADT," *Computer*, vol. 18, no. 4, pp. 25-35, Apr. 1985.
[18] M. A. Jackson, *Principles of Program Design*. New York: Academic, 1975.
[19] J. D. Warnier, *Logical Construction of Programs*. New York: Van Nostrand Reinhold, 1976.
[20] J. Cameron, *Tutorial: JSP and JSD: The Jackson Approach to Software Development*. Washington, DC: IEEE Computer Society Press, Nov. 1983.
[21] J. F. Stay, "HIPO and integrated program design," *IBM Syst. J.*, vol. 15, no. 2, pp. 143-154, 1976.
[22] F. DeRemer and H. H. Kron, "Programming-in-the-large versus programming-in-the-small," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 80-86, June 1976.
[23] W. F. Tichy, "Software development control based on module interconnection," in *Proc. 4th Int. Conf. Software Eng.*, Sept. 1979, pp. 29-41.
[24] S. S. Yau and J. P. Tsai, "A graph description language for large-scale software specification in a maintenance environment," in *Proc. COMPSAC 84*, Nov. 1984, pp. 397-407.
[25] S. S. Yau, J. P. Tsai, and R. A. Nicholl, "Knowledge representation of software life-cycle information using first-order logic," in *Proc. COMPSAC 85*, Oct. 1985, pp. 268-277.
[26] J. L. Archibald, B. M. Leavenworth, and L. R. Power, "Abstract design and program translator: New tools for software design," *IBM Syst. J.*, vol. 22, no. 3, pp. 170-187, 1983.
[27] G. Booch, "Object-oriented design," in *Tutorial: Software Design Techniques*, P. Freeman and A. I. Wasserman, Eds. Washington, DC: IEEE Computer Society Press, 1983, pp. 420-436.
[28] R. King and D. McLeod, "A unified model and methodology for conceptual database design," in *On Conceptual Modeling*, M. Brodie *et al.*, Eds. New York: Springer-Verlag, 1984, pp. 312-327.
[29] B. H. Liskov and S. N. Zilles, "Specification techniques for data abstraction," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 7-19, Mar. 1975.
[30] M. Shaw, "Abstraction techniques in modern programming language," *IEEE Software*, vol. 1, no. 4, pp. 10-27, Oct. 1984.
[31] D. Gries, "An illustration of current ideas on the derivation of correctness proofs," *IEEE Trans. Software Eng.*, vol. SE-2, no. 4, pp. 238-244, Dec. 1976.
[32] J. M. Spitzen, K. Levitt, and L. Robinson, "An example of hierarchical design and example," *Commun. ACM*, vol. 21, no. 12, pp. 1064-1075, Dec. 1978.
[33] F. W. Beichter, O. Herzog, and H. Petzsch, "SLAN-4—A software specification and design language," *IEEE Trans. Software Eng.*, vol. SE-10, pp. 155-162, Mar. 1984.
[34] R. A. Kemmerer, "Testing formal specification to detect design errors," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 32-42, Jan. 1985.
[35] I. J. Hayes, "Applying formal specification to software development in industry," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 169-178, Feb. 1985.
[36] G. D. Buzzard and T. N. Mudge, "Object-based computing and the Ada language," *Computer*, vol. 18, no. 3, pp. 11-19, Mar. 1985.
[37] B. J. Cox, "Message/object programming: An evolutionary change in programming technology," *IEEE Software*, vol. 1, no. 1, pp. 50-62, Jan. 1984.
[38] R. Wiener and R. Sincovec, *Software Engineering With Modula-2 and Ada*. New York: Wiley, 1984.
[39] A. Borgida, J. Mylopoulas, and H. Wong, "Generalization/specialization as a basis for software specification," in *On Conceptual Modeling*, M. Brodie *et al.*, Eds. New York: Springer-Verlag, 1984, pp. 87-114.
[40] E. W. Dijkstra, "The humble programmer," *Commun. ACM*, vol. 15, no. 10, pp. 859-866, Oct. 1972.
[41] ——, *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1973.
[42] G. D. Bergland and R. D. Gordon, "Software design strategies," in *Tutorial: Software Design Strategies*, G. D. Bergland and R. D. Gordon, Eds. Washington, DC: IEEE Computer Society Press, 1981, pp. 79-92.
[43] C. Bohm and G. Jacopini, "Flow diagrams, turing machines and language with only two formation rules," *Commun. ACM*, vol. 9, no. 5, pp. 366-371, May 1966.
[44] D. Gries, "On structured programming," in *Programming Methodology*. New York: Springer-Verlag, 1978, pp. 31-52.
[45] F. T. Baker, "Structured programming in a production programming environment," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 241-252, June 1975.
[46] A. I. Wasserman, "Information system design methodology," *J. Amer. Soc. Inform. Sci.*, Jan. 1980.
[47] I. Nassi and B. Shneiderman, "Flowchart techniques for structured programming," *ACM SIGPLAN Notices*, vol. 8, no. 8, pp. 12-26, Aug. 1973.
[48] H. P. Frei, D. L. Weller, and R. William, "A graphics-based programming-support system," in *Proc. ACM SIGGRAPH Conf.*, Aug. 1978, pp. 43-39.
[49] B. E. J. Clark and S. K. Robinson, "A graphically interacting program monitor," *Comput. J.*, vol. 26, no. 3, pp. 235-238, 1983.
[50] S. S. Yau and P. C. Grabow, "A model for representing programs using hierarchical graphs," *IEEE Trans. Software Eng.*, vol. SE-7, pp. 556-574, Nov. 1981.
[51] S. S. Yau and J.-P. Tsai, "A graph-based software maintenance environment," in *Dig. Papers, COMPCON 84 Spring*, Feb. 1984, pp. 321-324.
[52] S. H. Caine and E. K. Gordon, "PDL: A tool for software design," in *Proc. 1975 Nat. Comput. Conf.*, vol. 44, 1975, pp. 272-276.
[53] P. M. Lu, S. S. Yau, and W. Hong, "A formal methodology using attributed grammars for multiprocessing system software development, Part I: Design representation; Part II: Validation," *J. Inform. Sci.*, vol. 30, pp. 79-123, 1983.
[54] S. S. Yau and S. M. Shatz, "On communication in the design of software components of distributed computer systems," in *Proc. 3rd Int. Conf. Distributed Comput. Syst.*, Oct. 1982, pp. 280-287.
[55] S. S. Yau and M. U. Caglayan, "Distributed software system design representation using modified Petri nets," *IEEE Trans. Software Eng.*, vol. SE-9, pp. 733-745, Nov. 1983.
[56] B. W. Boehm, "Verifying and validating software requirements and design specifications," *IEEE Software*, vol. 1, no. 1, pp. 75-88, Jan. 1984.
[57] G. M. Weinberg and D. P. Freedman, "Reviews, walkthroughs, and inspections," *IEEE Trans. Software Eng.*, vol. SE-10, pp. 68-72, Jan. 1984.
[58] R. S. Boyer and J. S. Moore, "An overview of automated reasoning and related fields: Program verification," *J. Automated Reasoning*, vol. 1, no. 1, pp. 17-23, 1985.
[59] S. S. Yau, P. C. Grabow, and J. P. Tsai, "Modification and analysis of programs based on the hierarchical program model," *J. Inform. Sci.*, to be published.
[60] R. B. Dannenberg and G. W. Ernst, "Formal program verification using symbolic execution," *IEEE Trans. Software Eng.*, vol. SE-8, pp. 43-52, Jan. 1982.
[61] R. S. Boyer and J. S. Moore, *A Computational Logic*. New York: Academic, 1979.
[62] S. Igarashi, R. L. London, and D. C. Luckham, "Automatic program verification I: A logical basis and its implemetation," *Acta Inform.*, vol. 4, pp. 145-182, 1975.
[63] D. I. Good, "The proof of a distributed system in GYPSY," Inst. Comput. Sci. and Comput. Application, Univ. Texas at Austin, Tech. Rep. ICSCA-CMP-30, 1982.

[64] M. Gordon *et al.*, "A metalanguage for interactive proof in LCF," Dep. Comput. Sci., Univ. Edinburgh, Tech. Rep. CSR-16-77, 1977.
[65] S. S. Yau, and R. S. Wang, "Verification of communication in distributed system software," in *Proc. Pacific Comput. Commun. Symp.*, Oct. 1985, pp. 432–446.
[66] M. H. Halstead, *Elements of Software Science*. New York: Elsevier North-Holland, 1977.
[67] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 208–220, Dec. 1976.
[68] S. S. Yau and J. S. Collofello, "Some stability measure for software maintenance," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 545–552, Nov. 1980.
[69] ——, "On design stability measure for software maintenance," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 849–856, Sept. 1985.
[70] A. L. Goel, "Software reliability models: Assumptions, limitations, and applicability," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 1411–1423, Dec. 1985.
[71] P. E. Presson, "Software interoperability and reusability guidebook for software quality measurement," Rome Air Development Center, Griffiss Air Force Base, NY, Rep. RADC-TR83-174, July 1983.
[72] S. S. Yau, R. C. Cheung, and D. C. Cochrane, "An approach to error-resistant software design," in *Proc. 2nd Int. Conf. Software Eng.*, San Franciso, CA, Oct. 1976, pp. 429–436.
[73] S. S. Yau and R. C. Cheung, "Design of self-checking software," in *Proc. 1975 Int. Conf. Reliable Software*, Mar. 1975, pp. 450–457.
[74] J. R. Kane and S. S. Yau, "Concurrent software fault dectection," *IEEE Trans. Software Eng.*, vol. 1, pp. 87–99, Mar. 1974.
[75] S. S. Yau and F. C. Chen, "An approach to real-time control flow checking," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 126–137, Mar. 1980.
[76] B. Randell, "System structure for software fault tolerance," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 220–232, June 1975.
[77] A. Avizienis, "The *N*-version approach to fault-tolerant software," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 1491–1501, Dec. 1985.
[78] P. J. Denning, "Fault-tolerant operating system," *ACM Comput. Surveys*, vol. 8, no. 4, pp. 359–389, Dec. 1976.
[79] T. A. Linden, "Operating system structures to support security and reliable software," *ACM Comput. Surveys*, vol. 8, no. 4, pp. 409–445, Dec. 1976.
[80] T. Anderson, P. A. Barrett, D. N. Halliwell, and M. R. Moulding, "Software fault tolerance: An evaluation," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 1502–1510, Dec. 1985.
[81] B. S. Chen and R. T. Yeh, "Formal specification and verification of distributed system," *IEEE Trans. Software Eng.*, pp. 710–721, Nov. 1983.
[82] C. V. Ramamoorthy, W. T. Tsai, and T. Yamaura, "Metrics guided methodology," in *Proc. COMPSAC 85*, Oct. 1985, pp. 111–120.
[83] H. A. Jensen and K. Vairavan, "An experimental study of software metrics for real-time software," *IEEE Trans. Software Eng.*, pp. 231–234, Feb. 1985.

Stephen S. Yau (S'60–M'61–SM'68–F'73) received the B.S. degree from the National Taiwan University, Taipei, Taiwan, China, in 1958, and the M.S. and Ph.D. degrees from the University of Illinois, Urbana, in 1959 and 1961, respectively, all in electrical engineering.

He joined the faculty of the Department of Electrical Engineering, Northwestern University, Evanston, IL, in 1961, and is now Professor and Chairman of the Department of Electrical Engineering and Computer Science. He is currently interested in distributed computing systems, software engineering, and reliability and maintainability of computing systems. He has published numerous technical papers in these and other areas.

Dr. Yau is a Life Fellow of the Franklin Institute from which he received the Louis E. Levy Medal in 1963; he also received the Golden Plate Award of the American Academy of Achievement in 1964, the first Richard E. Merwin Award of IEEE Computer Society in 1981, and the IEEE Centennial Medal in 1984. He is also a Fellow of the American Association for the Advancement of Science. He was the President of the IEEE Computer Society in 1974–1975, the Division V (Computer Society) Director of the IEEE in 1976–1977, Chairman of IEEE Technical Activities Board Development Committees in 1979, and Editor-in-Chief of *Computer* magazine in 1981–1984. He has been a Director of the American Federation of Information Processing Societies (AFIPS) since 1972 and is now the President of AFIPS. He was Conference Chairman for the First Annual IEEE Computer Conference, Chicago, 1967, General Chairman of the 1974 National Computer Conference, Chicago; General Chairman of IEEE Computer Society's First International Computer Software and Applications Conference, Chicago, 1977 (COMPSAC '77), and Chairman of the National Computer Conference Board in 1982–1983. He is also a member of the Association for Computing Machinery, the Society for Industrial and Applied Mathematics, the American Society for Engineering Education, Sigma Xi, Tau Beta Pi, and Eta Kappa Nu.

Jeffery J.-P. Tsai (S'83–M'85) received the B.S. degree in computer science from Tamkang University, Taiwan, China, in 1976, the M.S. degree in computer engineering from National Chiao-Tung University, Taiwan, in 1978, and the Ph.D. degree in computer science from Northwestern University, Evanston, IL, in 1985.

He taught at Air Force Communication and Electronic College, Taiwan, from 1978 to 1980. From 1980 to 1982 he was a member of the Technical Staff at Digital Equipment Corp., Taiwan, where he did technical support for DEC's operating systems VMS, RSX-11M, and RSTS/E. In the same period, he was a part-time instructor at Tamkang University and Ming-Chung College. He joined the faculty at the University of Illinois at Chicago in 1985 as an Assistant Professor in the Department of Electrical Engineering and Computer Science. His current research interests are expert system, logic, and software engineering, man-machine interface design, and software development/maintenance environment.

Dr. Tsai is a member of the Association for Computing Machinery and the American Association of Artificial Intelligence.