Chapter 1

Software Architecture and Quality

Lawrence Bernstein
Stevens Institute of Technology

COPYRIGHTED MATERIAL

This chapter of the book illustrates Barry Boehm's many contributions to the technologies of software design, software quality, software project risk assessment, and software architecture.

Article 1–1. Software Design and Structuring
When I first read Barry Boehm's "Software Design and Structuring" in the 1970s I sent it to my colleagues and said this article was fundamental to our work. Today I assign it to my students. It remains relevant today despite several sea changes in software engineering technology. Barry lays the foundation for software architecture as well as the framework for quantitative analysis in software engineering. In the second paragraph in this article, he explains the need for data-driven conclusions when he cites the analysis of 220 types of software errors and finds that ". . . if more time had been spent on validating the design . . . prior to coding many of the conceptual errors would not have been committed to code." This finding is as true today as it was in 1975 and is only one of the many seminal ideas supported by data that Barry promoted.

Barry explains the application domains for bottom-up, top-down, and structured programming. The analysis approach explained in the article's appendix is a landmark use of risk analysis applied to software engineering tradeoffs. This approach can be found in most reputable software engineering textbooks.

Barry anticipates the current software engineering enthusiasm for model-driven design in this article, including the idea of continuous user involvement, a tenet of modern agile methods. He anticipates the Unified Modeling Language that is now the lingua franca of software architecture, but he wisely avoids advocating for automatic code generation, long a software engineering dream that so far has become a nightmare.

Software architects must read and understand Barry's findings, including the appendix, and apply them to their current and future software projects. Anything less is malpractice.

Article 1–2. Quantitative Evaluation of Software Quality
"Quantitative Evaluation of Software Quality," which constitutes the second article, is coauthored by Barry and others from TRW. It lays out the application of quality theory for software engineers and includes an extensive set of data with detailed analysis.

Copyright © 2007 IEEE Computer Society

Barry introduces the need for testable requirements. This concept is difficult to articulate and has proven elusive. This article identifies and evaluates many software metrics. It urges software engineers to set explicit and measurable quality objectives. It recommends that they be prioritized and compared with similar metrics from other companies. It points out that design and code inspections are a valuable practice. Studies in the 1980s at Bell Laboratories, followed by university and IBM research, validated his early insights.

The authors recognize that the article provides a "... framework for assessing the often slippery issues associated with software quality. ..." This framework has led to the evolution of several good software engineering processes now universally practiced.

Now, before reading further, skip to Barry's conclusions in his article and read them carefully. You will be drawn to read the analysis supporting his findings.

Article 1–3. Lessons Learned on an Early Application Generator
In 1996, Barry wrote about his early career as a rocket scientist. His story is compelling and reflects a common experience of software people pioneering software control of weapons and space systems. He explains that domain engineering is critical to establishing validated and quantifiable requirements. His observation about software product line architecting is frequently rediscovered. I first observed software product line approaches in practice in NASA's Apollo program and emulated them at Bell Laboratories as software began to dominate the many product and services businesses of the Bell System. When I visited Japan to see their software factories, I saw similar software product lines at Hitachi.

In the 1950s, before the idea of software engineering was born, Barry met early and often with his customers. This idea, captured in the phrase "management by walking around," has been embedded in the 1990s theory of agile programming.

Barry shows the effectiveness of using an application platform with many control parameters to execute rocket simulations. He advises using general-purpose and portable languages rather than a tailored language to avoid project dead ends. The article is a wonderful tour of our collective past with many insights posted along the way. By reading this article, you can learn to avoid software pitfalls without having to experience them and learn from the school of hard knocks. This is the essence of education.

Article 1–4. COTS Integration: Plug and Pray?
Commercial off the shelf (COTS) became the watchword in the 1990s without a critical understanding of its implications. Barry and Chris Abts studied the use of COTS in real projects and defined the conditions that lead to success and those that do not. This article is on COTS integration. They cite the danger that "COTS vendors do not change features ... in response ... to individual user needs." Large projects relying on COTS to reduce their costs and schedules often fall into these common traps: licensing constraints, high fees, poor computer resource use, lagging behind vendor upgrades, and little control over product life. When a software product will be widely deployed, the COTS fees can sometimes consume the product's profits. COTS use can make it difficult to converge on a reasonable software architecture by trapping developers into fragile and immutable software structures. The authors urge the use of Architecture Review Boards to avoid structural problems and ease the integration of COTS with custom-built modules in a system.

Article 1–5. Software Defect Reduction Top 10 List
In 2001, Barry collaborates with Vic Basili to transform the software engineering industry "from a fad-based practice to an engineering-based practice." They reported the top-10

steps that reduce defects. Professional software engineers would profit from this 2001 *Computer* article, as would all software engineering and computer science students. In brief:

1. Developers take 100 times less effort to find and fix a problem than one reported by a customer.
2. Half of software project work is wasted on unnecessary rework.
3. Twenty percent of defects account for 80% of the rework.
4. Twenty percent of modules account for 80% of the defects and half the modules have no defects.
5. Ninety percent of downtime comes from 10% of the defects.
6. Peer reviews catch 60% of the defects. This finding confirms Boehm's foresight concerning the effectiveness of inspections.
7. Directed reviews are 35% more effective than nondirected ones.
8. Discipline can reduce defects by 75%.
9. High-dependability modules cost twice as much to produce as low-dependability ones.
10. Half of all user programs contain nontrivial defects.

Article 1–6. COTS-Based Systems Top 10 List
In this 2001 article, Barry again collaborates with Vic Basili to hypothesize a set of decision criteria to evaluate your COTS strategy. They point out that empirical data on COTS adoption strategies are limited because of the immaturity of the approach. They offer their list as a starting point for formalizing the criteria for project adoption of COTS products within an architecture. I found the most interesting hypothesis to be that half the features in large COTS software products are never used. These features take up space and can slow the execution of the COTS integrated software product. Barry warned us against such "gold plating" in the 1970s. Turn to the article to read the rest of the list.

BARRY HAS IMPACT

Barry's contributions to the education of software engineers are legendary. He influences the daily activities of thousands of software engineers worldwide.

Article 1–1

Software Design and Structuring

Barry W. Boehm

INTRODUCTION

In 1974, the major leverage point in the software process is at the software design and structuring stage. This stage begins with a statement of software requirements. Properly done, the software design and structuring process can identify the weak spots and mismatches in the requirements statement that are the main sources of unresponsiveness of delivered software. Once these weak spots slip by the design phase, they will generally stay until during or after delivery, where only costly retrofits can make the software responsive to operational needs.

The software design and structuring phase ends with a detailed specification of the code that is to be produced, a plan for testing the code, and a draft set of users' manuals describing how the product is going to be used. Software costs and problems result from allowing coding to begin on a piece of software before its design aspects have been thoroughly worked out, verified, and validated. Figure 1 illustrates this quite well. It summarizes an analysis of 220 types of software errors found during a large, generally good (on-cost, on-schedule delivery) TRW software project [19,20]. The great majority of the types of errors found in testing the code had originated in the design phase. Even more significantly, the design errors are generally not found until later in the test process. Of the 54% of the error types that were typically not found until during or after the acceptance test phase, only 9% were coding errors. The other 45% were design errors.

A particularly important stage is design verification and validation. Many of the design errors could have been caught before the coding phase by using better design-review procedures or better tools for design consistency checking, some of which are now under development at TRW [20,21].

Table 1 provides another perspective on the importance of thorough software design specification and review. It shows data collected on a 24 ODD instruction Navy command and control software project that was analyzed for the CCIP-85 study [22]. Here, conceptual errors accounted for 61% of the total number of errors detected.

The most significant correlation, though, is the following. On the "Executive" components of the project, only 37% of the time was spent on analysis and design, and the ratio

From E. Horowitz (Ed.), *Practical Strategies for Developing Large Software Systems,* Addison-Wesley, 1975, 103–128. © 1975 Addison-Wesley. Reprinted by permission.

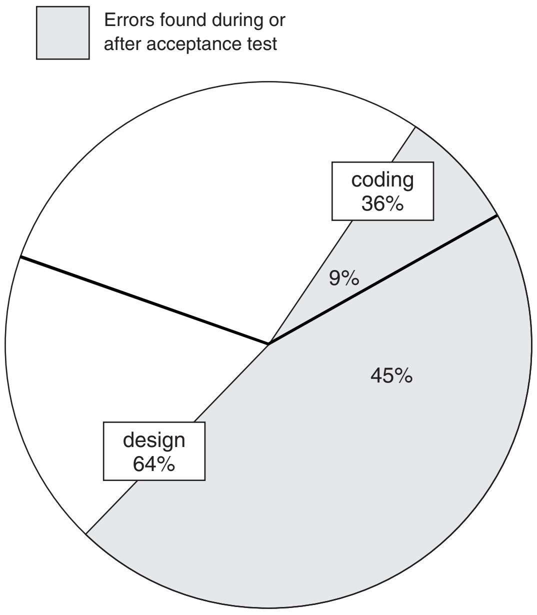

Figure 1. Software error sources.

of conceptual to clerical errors was 4:1. The other two components of the project spent 54% and 59% of their time in analysis and design; their ratios of conceptual errors to clerical errors were 0.6 and 0.5.

Some of the difference is probably due to the fact that executive software is conceptually more difficult. However, there can be little doubt that if more time had been spent on validating the design of the executive prior to coding, many of the conceptual errors would not have been committed to code. Similarly, better design tools and techniques

Table 1. Distribution of software error causes

Causes of error	Hardware diagnostics (%)	Executive (%)	User programs (%)	Total (%)
Unexpected side effects of changes	5	25	10	19
Logical flaws in design				
Original design	5	10	2	8
Changes	5	15	8	12
Inconsistencies between design and implementation	5	30	10	22
Total: conceptual errors	20	80	30	61
Clerical errors	40	20	50	28
Inconsistencies in hardware	40	—	20	11
	100	100	100	100
Total errors detected in 3 year sample	36	108	18	162
Number of instructions	4K	10K	10K	24K
% Analysis and design	59	37	54	

should catch or avoid many of the design errors early without increasing the cost of the design phase.

An increasing number of techniques for improving the design process are now becoming available. It is becoming harder and harder to sort out just what each technique does and does not provide.

This article provides a classification of software design and structuring techniques into a few main categories that can currently be considered as software design alternatives. For each alternative, it presents a short description and a balance sheet indicating the advantages and difficulties in using the technique.

In an overall comparison of the alternative techniques, none of them provides positive assurance of satisfying all the criteria, but each criterion is satisfied by at least one of the techniques. This would lead one to believe that an appropriate synthesis of the techniques might prove successful.

The following major design and structuring alternatives will be described and evaluated: bottom-up, top-down stub, top-down/problem statement, structured programming, and model-driven design.

Most software design and structuring techniques fall generally within one of these categories. Some other variations exist, such as iterative multilevel modeling and prototyping; these will be discussed below in the context of the other concepts.

BOTTOM-UP

The typical bottom-up software development begins with the development of several computational routines whose function is considered important to the application. Sometimes, the process involves modification of related previously existing computational routines in an attempt to avoid duplication of effort (Phase I in Figure 2).

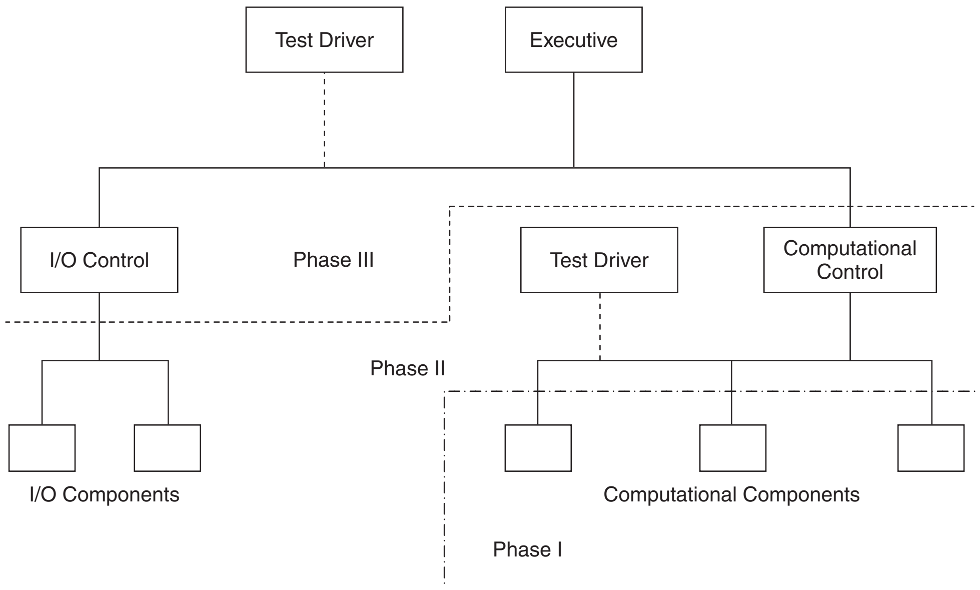

Figure 2. Typical bottom-up approach.

Once these are developed, the need is seen for a test driver to support testing of the modules and their interfaces, a control program to determine the sequence in which the computational routines are called, and some input–output routines to service both. These are then built in Phase II.

Once they are built, the need is seen for an input–output control capability to sequence the input and output routines, perform initializations, conversions, error checks, and so on. Also needed is an executive to moderate the interfaces between the input–output sector and the computational sector, and a test driver to test the entire system. These are then built, and the entire system is tested and subsequently modified as errors are detected in Phase III.

Advantages

High-risk Components. These are low-level component tasks for which there is a high risk as to whether or not they can be performed as specified; for example, processing sensor data in real time and automating human discrimination functions. They are well handled since the bottom-up process generally begins by concentrating on them.

Reusable Modules. Being component oriented, the bottom-up approach tends to do well here, but often at the expense of other considerations.

Disadvantages

Integration, Verification, and Validation. These tend to be very costly because of the lack of early attention to interface problems and overall system requirements.

Visibility and Tracking. Tracking the progress of individual components is straightforward, but the lack of well-defined overall user requirements and system interfaces makes overall visibility difficult.

Maintainability. Often, different assumptions made during the development of individual bottom-level components lead to a higher-level control structure and a data structure that are simply "kludged-up" to make the components work together in their current form. Such constructs are very difficult to modify to meet changing requirements.

User and Data Structuring Issues. The lack of early attention to overall system requirements mentioned above and the generally "kludged-up" data structures are the sources of these difficulties.

TOP-DOWN STUB

The top-down/stub approach, as described in References 1 and 2, proceeds as follows. The designer begins by determining what overall functions will be performed by the software system, in what order, and under what conditions. He then proceeds to develop a working top-level computer program, containing all the logic controlling the sequencing between functions, but inserting dummy programs or "stubs" for the functions. He then proceeds to test this top-level program thoroughly before proceeding to the next step.

The succeeding steps consist of fleshing out the stubs into a lower-level sequence of control logic, computation, and subfunctions, each of which is again represented by a stub. Each subprogram is developed to replace a stub; it can be tested immediately, not only by itself, but also as a part of the software subsystem and system it joins by replacing the stub.

The top-down/stub approach to software design is often enhanced by several optional aids. One is the HIPO (hierarchical input–process–output) technique [3], which provides designs with a standard way to represent the inputs, processes, and outputs of each module and the module's relationship to the overall control hierarchy. Currently, no automated aids are available for developing or checking HIPO descriptions. Another is the walkthrough, an element of "egoless programming" [4,5] in which the designer–programmer presents his design and its underlying rationale to his team leader and a group of peers to eliminate as many bugs as possible before coding begins. Another is the chief programmer team [6], a management approach in which a single individual (aided by a backup) produces the bulk of the design, writes a good deal of the code, and reviews all of it personally. Another is structured programming, to be discussed in more detail later.

In top-down/stub design, changes to the software are first worked out at the highest level affected, and then propagated downward as appropriate. The process thus concludes with not only a completed design but a completed and tested program, in which the lower levels of the control structure were not designed until the upper levels were programmed and tested.

Advantages

Early Integration. Committing the top-level design early to testable code focuses attention on interface problems at a stage at which they are relatively easy to resolve. (It also creates a useful additional deterrent to arbitrary changes of scope.)

Parallel Testing. Testing can proceed much faster when one can immediately test the program at the unit, subsystem, and system levels. (Experience has shown that one must prepare to handle an overload of test shots at this stage while somehow preserving good turnaround time.)

Visibility and Tracking. The well-defined interfaces and stub organization make it relatively easy to track progress and isolate trouble spots.

Maintainability. Traceability from high level requirements to low level code is maintained at least partially through the hierarchical control structure.

Disadvantages

High-Risk Components. Proceeding in strictly top-down fashion can lead to a large investment in top-level structure before discovering that a low-level component cannot do its job, for example, to process inputs in real time or to automatically discriminate friend from foe, at which point one must virtually start another top-down design from scratch.

Common or Reusable Components. Particularly when several designers are involved, people have found that several branches of the control hierarchy terminate with a requirement for a similar capability, say, for sorting, but that a great deal of rework must be done at the higher levels to accommodate the requirement with a single routine.

Machine dependence. As it begins by developing an executing version of the top-level control structure, the top-down/stub approach unfortunately commits the project to a specific hardware system right away before the hardware implications of various design decisions are completely understood. One may avoid this difficulty by using the top-down/ stub approach more as a programming technique than as the complete design process.

User and Data-Structuring Issues. The top-down/stub approach is a formalism for organizing a program's control structure. If the user's requirements are poorly understood, or if the data structure implications are not also considered in advance, the delivered software may be unresponsive or hard to maintain. With an experienced, thoughtful designer, these will generally not be problems. But if considered as a "cookbook" approach by inexperienced designers, the top-down/stub approach can lead to problems; one group using it for the first time had an 83% overrun on a job budgeted at 450 man-hours.

TOP-DOWN PROBLEM STATEMENT

Several variants exist on the problem statement approach but, in general, each attempts to define a special problem domain and establish for it a set of constructs and capabilities for expressing, analyzing, and ideally generating programs within the problem domain. In the domain of business system problems, ISDOS and its Problem Statement Language epitomize the problem statement approach [7]; a good review of business problem statement languages is Reference 8.

Another variant involves the use of iteratively refined simulations as successive problem statements that drive program design and development, as in Iterative Multilevel Modeling [9], SODAS [10], and DES [11]. Another is Glaser's LOGOS approach, centered around the problem domain of computer operating systems [12].

In the ballistic missile defense problem area, TRW has evolved a set of problem statement capabilities currently being used in the design and development of the large Site Defense of Minuteman (SDM) software project. These capabilities allow a designer to specify a top-down control structure via a hierarchical set of functional sequence diagrams (FSDs). The FSDs are specially structured time-sequenced flow charts whose elements may be expanded into attribute descriptions via a functional properties matrix, which may identify the various categories of activity (data handling, control, report, test, etc.) as a function of mode (preoperational, operational, or postoperational). Another associated design and data-structuring construct is the N-square chart, which indicates the flow of data to and from the component elements of the system.

Another design and development aid on the SDM project is a process construction language. This language is designed to be able to drive both a simulation of the SDM system and the building of the program itself. Thus, in one mode, the command "PERFORM RADAR SCHEDULING . . ." will call out a simulation of the radar-scheduling activity; in the other mode, the actual radar-scheduling code will be assembled for execution.

Advantages

Early Integration. This is provided by the emphasis on early specification of the sequence of functions and their interactions via data, and of a simulation model of the system to be developed.

Data Structuring. Most of these approaches provide explicit capabilities for assessing data implications and some degree of data structuring.

Maintainability. In general, the problem statement provides very good traceability from requirements to code, and a natural way of specifying modifications.

Visibility and Tracking. The problem statement provides a well-understood context within which software development can be monitored.

Disadvantages

Limited Applications Context. The price generally paid for the implicit contextual framework provided by the problem statement approach is a restriction of the problem domain for which such a framework is relevant. Specifying BMD software via ISDOS would be extremely awkward, as would the use of FSDs to specify a personnel–payroll data processing system.

Machine-dependent Support Software. Although the problem statements are generally machine dependent, at the present time the programs that operate on them generally are not.

STRUCTURED PROGRAMMING

Structured programming [13,14] is strictly more of a program-organization discipline than a design technique, but it can be used to significantly enhance most of the available design techniques. It is basically a set of standards for organizing the control structure of a set of computer programs. The key ideas in structured programming are:

1. Only '"proper programs" having one flow of control in and out of each unit should be developed.
2. Only three basic control structures are allowed: DOWHILE, IFTHENELSE, and SEQUENCE. These three structures are sufficient to express any proper program [15]. Two additional structures are optional: the DOUNTIL and the CASE.
3. Programs are organized according to a hierarchical, modular block structure.
4. Additional standards are imposed on the size of modules and the formatting of instructions, declarations, program commentary, and so on.

Advantages

The modularity, standard control structure, and format conventions of structured programming generally make programs easier to understand. Organizing programs in this way also generally requires more careful thought; this leads to many advantages, particularly in reducing the incidence of errors. Also, the structuring and limited number of logic components makes it easier to develop automated software development, test, and documentation tools. Enforcing any sort of programming standard generally leads to fewer interface problems. These factors combine to produce the following main advantages:

- Visibility and tracking
- Maintainability
- Testing and reliability
- Support of early integration

Disadvantages

Asynchronous Control and Error Exits. The natural way to handle these types of program controls involves the use of nonproper program logic. To express an interrupt capability with structured programming, for example, would require IFTHENELSE for each type of interrupt after every statement in the program.

Common or Reusable Components. Structured programming strictly leads to the use of duplicated segments of in-line code to handle many functions usually performed by common multipurpose components.

User and Data-structuring Issues. Structured programming does not address these. It is strictly a formalism for organizing the control aspects of a program.

Table 2 summarizes and extends the balance sheets given on the previous pages. For each of the approaches described, it provides an assessment for each of several software criteria or characteristics. Most of the ratings are straightforward; the "caution" rating implies either that the approach can lead the designer astray under some conditions, or that the approach says virtually nothing about how to achieve the characteristic.

As seen in the table, none of the techniques described provide positive assurance of satisfying all the criteria, but each criterion is satisfied at least to the "good" level by one of the approaches. This would lead one to believe that an appropriate synthesis of the techniques might prove successful. The concept of model-driven design evolving at TRW gives promise to providing such a synthesis. A rating of the near-term objectives for model-driven design is given in the rightmost column. The concept is summarized below.

MODEL-DRIVEN DESIGN

Model-driven software design (see Figure 3) begins with two models that are independent of the specific software capabilities to be designed. One is a "process model" of the software design and development process, best illustrated by such documents as the recent TRW

Table 2. Comparison of alternative design techniques

	Bottom-up	Top-down/ stub	Top-down/ problem statement	Structured programming	Model driven
Universality	VG	VG	C	G	G
User RQTS refinement	P	C	G	C	G
Design validation	P	G	G	G	VG
Early integration	P	VG	VG	G	VG
Data structuring	C	C	G	C	G
High-risk components	VG	C	G	C	VG
Machine independence	C	P	C	VG	VG
Reusable modules	VG	C	G	C	G
Efficiency	C	C	G	C	G
V & V assurance	P	G	G	G	G
Maintainability	P	G	VG	VG	VG
Visibility and tracking	P	VG	VG	G	VG

VG = very good, G = good, C = caution, P = problem source.

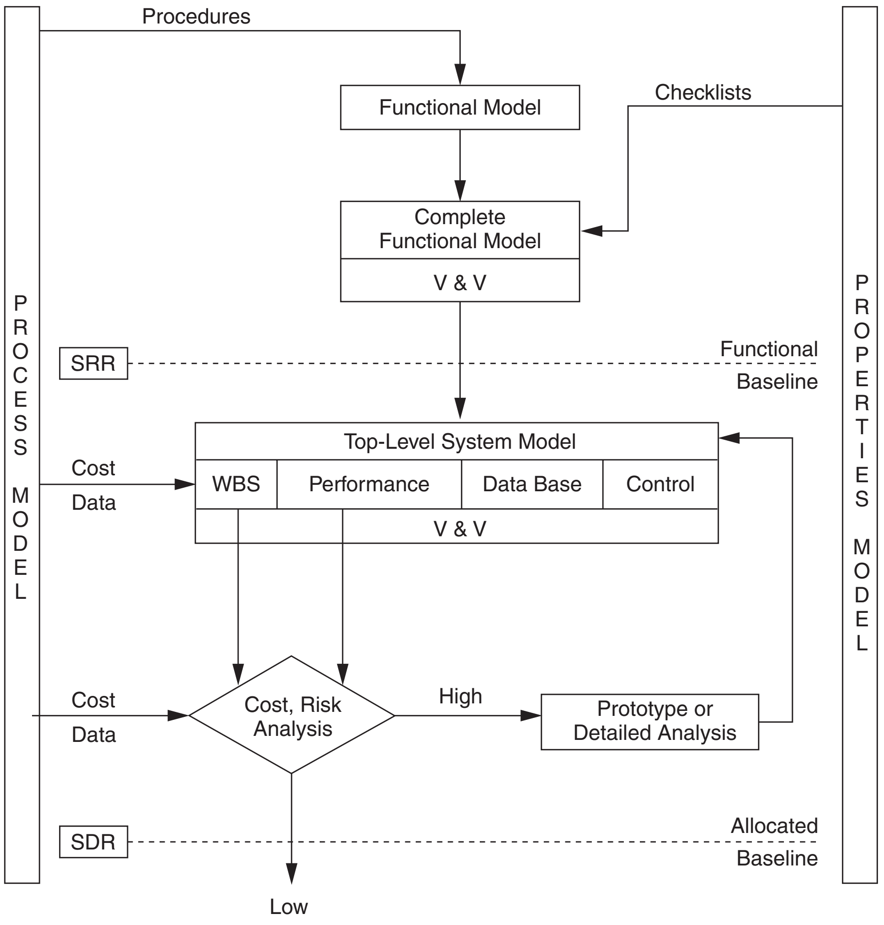

Figure 3. Model-driven software design.

Software Development and Configuration Management Manual [16]. The other is a "properties manual" of the characteristics of a good software product, best illustrated by such documents as the recent TRW report for NBS on characteristics of quality software [17].

The process model specifies the overall sequence of activities, which begins with the specification of another model, this time a specific-project-oriented model of the functions to be performed by the software. This model is checked for completeness via checklists derived from the properties model, using techniques along the lines of the requirements/properties matrix formalism being developed at TRW [18]. Additional walkthroughs or protocols are worked out with the prospective user to provide design verification and validation (V & V) followed by an official system requirements review and establishment of a functional baseline for the system.

The resulting functional model is then used to generate a top-level system model with four complementary components:

1. A model of the overall control structure, using an enhanced version of Functional Sequence Diagrams

2. A model of the database, using the N-square formalism and additional data structure models appropriate to the application (batch, online, real-time random/sequence access, etc.)
3. A model of system performance or processing efficiency (generally a rough analytic model will do at this stage, but more detail may be appropriate for performance-critical systems)
4. A rough model of the work breakdown structure (WBS), cost, and schedule required to develop the software, based on experimental data that form a part of the process model

After these models are checked for consistency, the WBS and performance models are exercised to determine whether there are any high-risk components within the software to be developed. If not, the design can proceed top-down. If so, however, some detailed simulation or bottom-level prototyping is needed to determine the right approach to be taken for the critical components. A good conceptual model for this process is given by statistical theory; an example is given in the appendix, which shows how this kind of risk analysis helps determine the appropriate sequence of design activities in a system involving real-time sensor data processing.

It is important that such risk analysis and design sequence activities include strong participation from the user, as both the risk assessments and the resulting design decisions generally involve assumptions about the user's needs and priorities. Thus, even though the system design phase is culminated by a thorough systems design review by the user, this fact should not lead either the user or the developer to succumb to the temptation to avoid user-developed interaction during this or other design phases. In fact, many successful software projects owe a good deal of their success to having been designed by a joint user-developer team, an option that deserves consideration for the future government software procurements.

The preliminary software design (see Figure 4) begins with a thorough description of the information-processing system to be developed (the allocated baseline) and an understanding of the residual risks involved in developing it. Sometimes, these risks are extremely low, in which case the allocated baseline can indeed be a basis for a firm determination of the resources required to complete the project, for example, a fixed-price-type contract. However, in most cases even the detailed analysis or prototyping activities described above will not eliminate a good deal of cost uncertainty that can only be resolved by proceeding to the next step or preliminary software design.

Often, then, the preliminary design step will involve a continuing refinement of the cost and risk models, and their supporting WBS and performance models. These are developed in conjunction with a top-down refinement of the models of the data and control structures. This process generally begins with a more detailed layout of the data structure, which then provides a general context for the refinement of the control structure. For large projects, the latter process can be assisted by a machine-readable but machine-independent description of the inputs and outputs of each module, along the lines of the HIPO technique but extended to accommodate automated consistency checking of the attributes of the inputs and outputs.

The preliminary design phase, and the detailed design phase to follow, are similarly characterized by:

- Continuous user involvement
- Explicit planning for and performance of design verification and validation activities

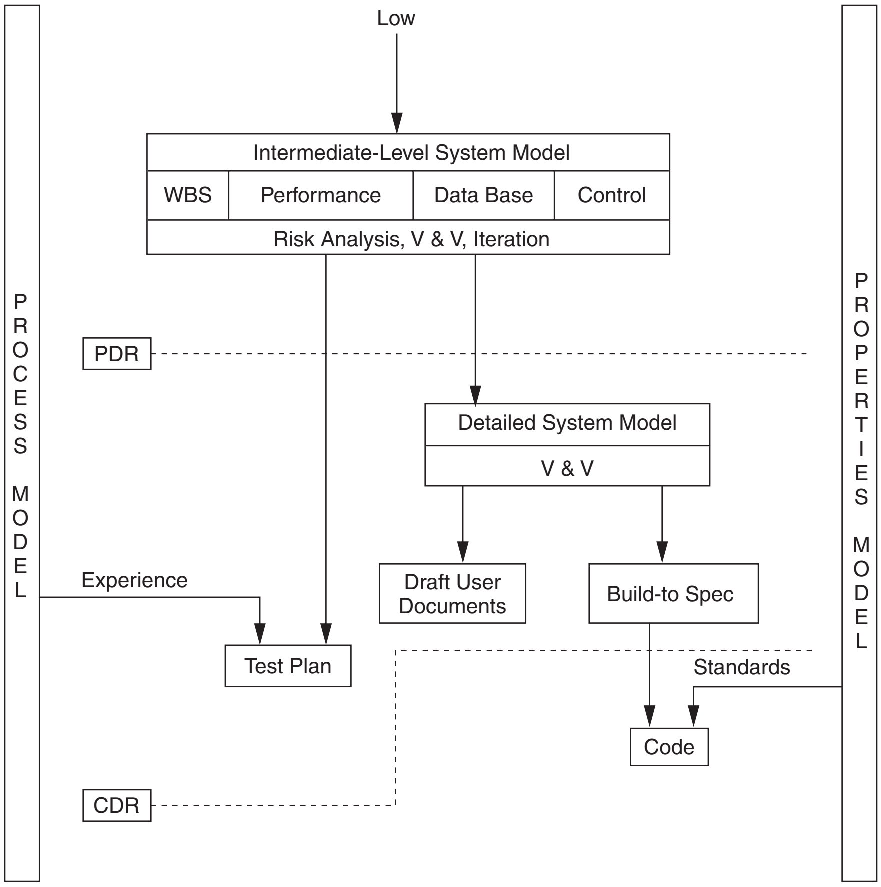

Figure 4. Model-driven software—preliminary and detailed design.

- Explicit consideration of risk and uncertainty resolution
- Thorough reviews to assure firm resolution of issues in each phase before proceeding to the next

The detailed design phase concludes not only with a build-to coding specification, but also a thorough test plan and draft user's manuals so that the user has an accurate picture of how the product will work.

Some additional points on model-driven preliminary and detailed design are:

- In order that user considerations rather than hardware constraints continue to be the major detriment of the design, a commitment to a specific hardware configuration should be delayed until as late as possible.
- At some point, the result of a risk analysis may be to factor the development into two or more phased-development activities (five "loops" in the TRW Site Defense project), in which the actual performance in the earlier developments can be used to guide the design of successive development portions.

Advantages

Emphasizes Design Validation. Model-driven design makes explicit provisions for design test planning and performance before proceeding to subsequent phases. The several models provide reference information that can be used to support design consistency and completeness checking.

Emphasizes Early Integration. Greater assurance of design integrity at early stages of development results from the formulation and cross-checking of successively more detailed models of control structure, data structure, performance, and work breakdown structure.

Avoids Premature Hardware Commitment. Model-driven design is basically a "software-first" approach. The performance model and optional prototyping activities provide the capability of resolving many of the hardware–software tradeoff issues before committing the project to a particular hardware configuration.

Explicit Risk Analysis for Design Decision. In general, the objective of model-driven design is to avoid building up a large inventory of code or detailed design specifications when there is a high risk that they may have to be scrapped as nonresponsive or expensively retrofitted. The explicit assessments of risk and the options to proceed in high-risk situations via higher-level design analysis or quick disposable prototypes both support this objective.

No Major Weak Spots. Although model-driven design has been synthesized from other design approaches to fulfill this objective, its full performance in practice has yet to be demonstrated. Currently, the Site Defense project at TRW is the closest approximation to complete model-driven design and, though performance to date is highly satisfactory, it has not reached its culmination. The major potential problem area is that inexperienced or model-happy designers will use model building as an alternative to thinking about and resolving design development issues, rather than a means to stimulate thinking about and resolving such issues. Thus, "model-driven software development" should admit to both of the following interpretations:

- The software development activities are driven by the formulation, analysis, and iterations of preliminary models.
- The model development activities are driven by the overall objective of producing a cost-effective software product.

Disadvantages

Exists Only for Limited Problem Domains. At the current time, the model-driven approach can be applied to real-time software projects with characteristics somewhat similar to ballistic missile defense. Some additional capabilities in the performance and data modeling areas in particular need to be provided to make it generally applicable.

Extension Requires New Developments. The main capabilities needed to provide general model-driven facilities are:

- A language for expressing software requirements design specifications, and performance characteristics for simulation
- A set of formal and automated aids for checking the completeness and consistency of software requirements and designs
- A set of capabilities for top-down data structuring

Requirements/Design/Simulation Language. A language is needed for specifying the components and characteristics of software systems in ways that support performance analysis and automated aids to consistency checking and traceability of requirements into design and code specifications, completeness checklists, and, most importantly, a clean and common understanding among the software builders, buyers, and users of what is to be built. Components of such a language would include:

- Precise terminology (Are response time, reaction time, turnaround time, port-to-port time, and delay time the same or different concepts?)
- A taxonomy of elementary information-processing functions (send, receive, store, query, sort, process, tally, etc.)
- A means of describing performance or specifying a performance or specifying a performance model
- Facilities to characterize software via inputs and outputs, via properties, or via data structures and dynamics

Requirements Analysis and Design Verification and Validation Aids. These include completeness checklists and capabilities for consistency checking of software designs, such as the module description consistency checker being built at TRW along the lines described in Reference [17].

Hierarchy of Performance Modeling and Simulation Tools. Capabilities are needed to support smooth evolution of performance models to levels of greater detail or of higher aggregation; for example, along the spectrum from analytic models through semianalytic simulations to bit-level simulations.

Top-down Data Structuring. Capabilities are needed to accumulate a number of statements of data usage (existing files, required outputs and response times, access control, etc.) and organize them in ways to support data structuring and performance analysis. ISDOS [7] is a good step in this direction.

Structured Programming Extensions. Language features are needed to extend the constructs of structured programming in ways that accommodate asynchronous control. Existing languages should be extended to easily accommodate structured programming.

Online Analysis and Design Aids. Providing such capabilities would not only reduce design concurrency problems in multiperson efforts but would also assure that the design is in machine-readable form for easier analysis.

REFERENCES

1. R. C. McHenry, “Management concepts for top down structured programming,” IBM Corporation Technical Report No. FSC 73-0001, February 1973 revision.
2. H. Mills, “Top down programming in large systems,” in *Debugging Techniques in Large Systems,* Randall Rustrn (Ed.), Prentice-Hall, Englewood Cliffs, NJ, 41–55, 1971.
3. “HIPO: Design aid and documentation tool,” IBM Corporation, Armonk, NY, SR20-9413.
4. G. F. Weinberg, *The Psychology of Computer Programming,* Van Nostrand Reinhold, 1971.
5. “Structured walk-through: A project management tool,” IBM Corporation, Bethesda, Maryland, August 1973.
6. F. T. Baker, “Chief programmer team management of production programming,” *IBM Systems Journal,* 11, 1, 1972.

7. D. Teichroew and H. Sayari, "Automation of system building," *Datamation,* pp. 25–30, August 17, 1971.
8. J. D. Cougar, "Evolution of business system analysis techniques," *ACM Computing Surveys,* 67–198, September, 1973.
9. F. W. Zurcher and B. Randell, Iterative Multilevel Modeling—A Methodology for Computer System Design, in *IFIP Congress,* Edinburgh, 1968.
10. D. L. Parnas, "More on simulation languages and design methodology for computer systems," in *Proceedings Spring Joint Computer Conference,* pp. 739–743, 1969.
11. R. M. Graham, G. L. Clancy, and D. B. DeVaney, "A software design and evaluation system," *ACM Communications,* 110–116, February, 1973.
12. E. L. Glaser, et al., "The LOGOS project," *Digest of Papers 72; Innovative Architecture Compcon 72,* IEEE, Catalog No. 72-CHO 59-3C, pp. 175–192, 1972.
13. E. W. Dijkstra, "Notes on structured programming," in *Structured Programming,* Dahl, Dijkstra, and Hoare (Eds.), Academic Press, London, 1972.
14. H. D. Mills, "Structured programming in large systems," IBM/FSD, Gaithersberg, November 1970.
15. C. Bohm and G. Jacopini, "Flow diagrams, Turing machines and languages with only two formation rules," *ACM Communications, 9,* 5, pp. 366–371, 1968.
16. *Software Development and Configuration Management Manual,* TRW Systems Group, Redondo Beach, California, December 1973.
17. "Characteristics of software quality," TRW Document No. 25201-6001-RU-00, TRW Systems Group, Redondo Beach, California, 28 December 1973.
18. B. W. Boehm, "Some steps toward formal and automated aids to software requirements analysis and design," TRW Systems Group, Redondo Beach, California, November 1972.
19. "Information processing/data automation implications of Air Force command and control requirements in the 1980's," U.S. Air Force, SAMSO/XRS-71-1, April 1972.
20. "Characteristics of software quality," TRW Document No. 25201-60001-RU-OO, December 28, 1973.
21. B. W. Boehm, "Some steps toward formal and automated aids to software requirements analysis and design," to be presented at IFIP Congress 1974.
22. J. McGonagle, *A Study of a Software Development Project,* J. D. Anderson and Associates, October 1971.

APPENDIX. SOFTWARE SYSTEM DESIGN AND DEVELOPMENT: TOP-DOWN, BOTTOM-UP, OR PROTOTYPE?

One difficulty which may occur in using the top-down structured programming approach to software development is that of high-risk modules. If one proceeds too routinely through a top-down structuring of a program, one may at times find oneself with a fifth-level module that is supposed to "understand natural-language queries," "process a megabit of sensor information in two milliseconds," or something equally impossible. And, in coming up with an acceptable compromise for handling this module's task, one may have to rework all of the structure that was painstakingly worked out at levels 1, 2, 3, and 4. It is such practical difficulties that have led same people to consciously reject the top-down approach in favor of the bottom-up approach, and led others to advocate building a throw-away prototype before proceeding to develop the production-engineered software system. However, in straightforward programming projects with no high-risk elements, the prototype approach would generally result in duplicated effort.

There is a formalism that can help determine how much bottom-up prototyping should precede the top-down specification of the overall program structure. This is the statistical decision theory approach, which takes explicit account of risk aspects. It is illustrated below by an example.

The Statistical Decision Theory Approach—An Example

Suppose you are organizing a software project that has within it a requirement to process 30,000 bits of sensor data within 5 milliseconds. You have an algorithm that you are pretty sure will work, and if you proceed top-down to develop the program and it works, you will have to spend an estimated $40,000 in performance penalties, ending up with a loss of $50,000 on the job. The other alternatives are (1) to develop the program initially with the less efficient algorithm, accepting a loss of $10,000 on the project, or (2) to spend $10,000 to quickly develop a bottom-level prototype and test it in a way that gives a much stronger assurance, though no guarantee, that it will work in practice if it passes the tests. How do you decide what to do?

Using statistical decision theory, you would begin by quantifying such words as "pretty sure" and "stronger" into probabilistic terms. This is a very subjective step, but there are various techniques involving expert polling that can help. To begin with, let us suppose that you established a probability of 0.65 that the efficient algorithm would work. From this, we can see that the expected value of proceeding top-down with the efficient algorithm (Approach A) is:

$$0.65(\$30{,}000) + 0.35(-\$50{,}000) = \$2{,}500$$

which is clearly better than the $10,000 loss associated with proceeding top-down with the less efficient algorithm (Approach B). However, the positive expected value using Approach A is not completely reassuring, as most software performer organizations have a highly asymmetric utility function such as that shown in Figure 5. It is generally fairly linear on the profit side, but drops off steeply if the project begins to encounter losses. Thus, the expected utility of Approach A is more like

$$0.65(0.4) + 0.35(-0.9) = -0.055$$

which is, again, clearly better than the –0.4 utility resulting from Approach B, but still not very satisfactory.

In order to evaluate the expected value and utility of the bottom-level prototype approach (Approach C), we first need to estimate the effectiveness of the prototype as a predictor of the outcome of the success of the project. We do this by estimating two probabilities:

1. The probability that the prototype is successful, given that the project is tractable. Suppose we estimate this as 0.95; that is, 5% of the time, the prototype would be badly done and fail, but the project would succeed if attempted.
2. The probability that the prototype is successful, but that the project would fail (due perhaps to problems of scale, or unrepresentativeness of the prototype). Suppose we estimate this as 0.15.

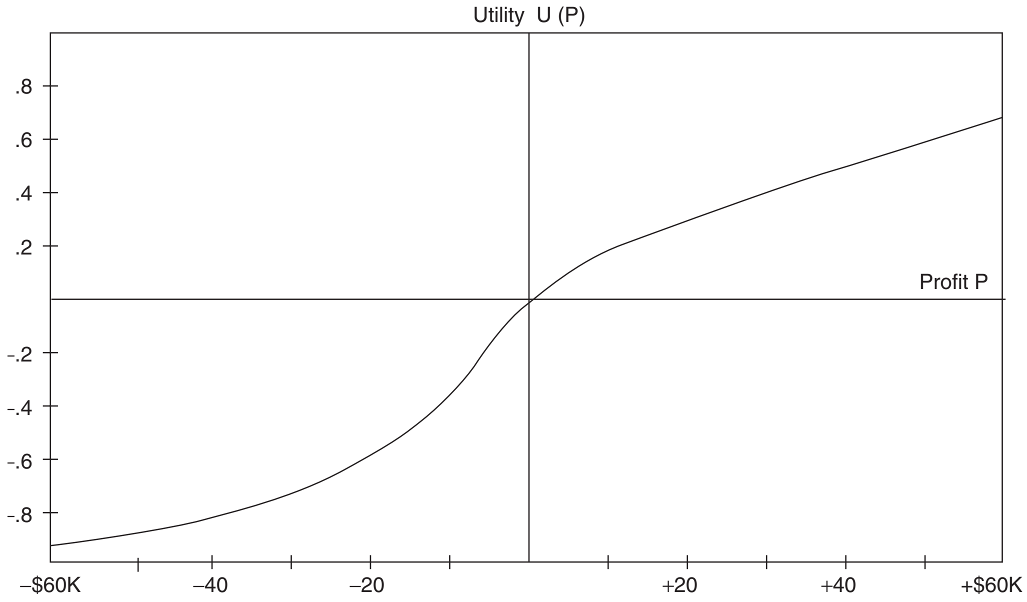

Figure 5. Utility function for software project outcome.

Then we can use Bayes' formula to calculate the probability that the project will succeed, given that the prototype succeeds:

$$P = \frac{(0.95)(0.\ 65)}{(0.95)(0.65) + (0.15)(0.35)} \doteq \frac{0.617}{0.670} \doteq 0.92$$

Thus, the expected value associated with Approach C is:

$$0.670[0.92(\$30\text{K}) + 0.08(-\$50\text{K})] + 0.330(-\$10\text{K}) - \$10\text{K} = \$2.5\text{K}$$

where 0.670 is the probability of the prototype being successful and the decision being made to use the efficient algorithm. The expected utility is

$$0.670[0.92U(\$20\text{K}) + 0.08U(-\$60\text{K})] + 0.330U(-20\text{K}) = +0.009.$$

Thus, in this situation, an expenditure of $l0K on a prototype can change the expected utility of the outcome from –0.055 to +0.009.

If we can estimate costs and probabilities with reasonable accuracy, then the statistical decision theory approach can help us steer a proper course between top-down, bottom-up, and prototype activities. Currently, given the scanty firm knowledge we have on software cost estimation, and our lack of experience in expressing possible software outcomes in probabilistic terms, the accuracy of the estimates will not support cookbook applications of the method. However, it still has considerable value as a conceptual framework for software development decision making, and provides another reason to press for more collection and analysis of quantitative software data to support cost estimation.