AN INTEGRATED APPROACH TO SOFTWARE DEVELOPMENT

RUSSELL J. ABBOTT

AN INTEGRATED APPROACH TO SOFTWARE DEVELOPMENT

RUSSELL J. ABBOTT

A Wiley-Interscience publication
JOHN WILEY & SONS

New York • Chichester • Brisbane • Toronto • Singapore

Library of Congress Cataloging in Publication Data
Abbott, Russell J., 1942-
An integrated approach to software development.

"A Wiley-Interscience publication."
Bibliography: p.
Includes index.
1. Computer software—Development. I. Title.

QA76.76.D47A23   1985   001.64'25   85-9427
ISBN 0-471-82646-4

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

[illegible]

PREFACE

This book is intended as a text in software engineering courses and as a day-to-day working reference for practicing software engineers.

1. It could be used as a software engineering text in computer science departments and in systems analysis courses in business and engineering. In the latter capacity chapters 2 and 3 would be most relevant.
2. For practicing systems analysts, software engineers, and programmers this book can serve as a useful reference for the development of requirements and software design documents, system specifications, and testing documentation.

The format of this book makes it easy to adapt to both purposes. It is organized as a collection of annotated outlines for technical documents that are important to the development and maintenance of software. These documents are not just provided in bare form. Their annotations explain why they are organized as they are and why their contents are as shown. Thus the annotations are used for tutorial purposes when necessary. To varying degrees the documents rely on relatively sophisticated concepts and in some sections require a certain amount of formal notation. Therefore this material is presented in the context of the documents in which it appears rather than in the traditional academic manner. In that way its relevance is immediately apparent. In addition, the outlines may be used by practitioners as models for their own documents.

SIGNIFICANT FEATURES

1. A clear distinction is made between *requirements* and *behavioral specification*. Requirements refer to the needs of the end user, behavioral specifications, to a behavioral description of a system that meets those needs.

2. To express a user's requirements we must first understand the user's world. Techniques from the disciplines of artificial intelligence and databases serve as an approach to the analysis of an arbitrary world. The ideas inherent in these techniques provide the book with a unifying theme.

3. To specify the behavior of an arbitrary system requires precision and clarity. A method of expressing behavioral specifications is presented in a way that is both rigorous and intuitive. It also includes the following:
   a. The user's view of a system in terms of *user conceptual model*.
   b. Operations in road-map form available to the user for manipulating that conceptual model.

4. For an organization to operate a system requires *procedures* and *administration*. These topics are discussed as a complement to behavioral specifications.

5. The approach to system *design* stresses design in terms of reusable components and combines into a single methodology techniques from object-oriented programming, dataflow design, and stepwise refinement. A theory of interpretive systems is developed to show how programming languages and application systems alike may be understood as interpreters. A new construct, the *component*, captures the essense of modularity and unifies the notions of types, subprograms, tasks, and objects. This methodology places input and output in their proper perspective and is particularly oriented toward design in terms of reusable components and implementation in a modern programming language such as Ada.*

6. *Validation, verification,* and *testing* are integrated into the development procedures to rescue them from their roles as the expensive black sheep of software development.

7. The book contains recommended outlines of documents for requirements, behavioral specifications, procedural and administrative manuals, and system and component design documentation.

8. An appendix presents an easy to understand formal notation for expressing specification. It is structured to parallel the syntax of programming langauges. Thus it is comfortable for most programmers to read and write.

*Ada is a trade mark of the United States Department of Defense.

USE OF THIS BOOK IN LECTURE COURSES

The software engineering curriculum is still rather fluid. Schools package systems analysis, specification, design, and management in various ways. This book can be used in any of a number of senior or graduate level courses:

1. *Software Engineering.* It can serve as the primary text in a course in software engineering. The entire book can be covered in a single semester, although the pace may be somewhat rapid. Readings from the references may be used to explore in more depth topics of particular interest to the instructor. A supplementary software engineering survey book (e.g., Pressman, 1982; see the annotated bibliography at the end of this preface) may be used for breadth.
2. *Software System Design.* It may be used as the primary text in a software system design course in which only the chapters on system specification and design would be covered. Readings from the literature may be selected to explore particular topics in more depth.
3. *Software Architecture.* It may be used as the primary text in a course on software architecture and design in which only the chapters on design would be covered. It should be supplemented by readings from the literature.
4. *Systems Analysis.* It may be used as a primary text in a systems analysis course in which only the chapters on requirements and specifications would be covered. It should be supplemented by readings in the organizational use of computer systems.
5. *Software Engineering Project Management.* It may be used as a secondary text in a course on the management of software engineering projects.

The exercises throughout the book provide ample practice material and are structured to allow (if desired) concentration on a single system. A set of exercises is included for an appointment-scheduling system; if all are completed a full set of development documents for such a system will be written.

USE OF THIS BOOK IN A PROJECT COURSE

This book is ideally suited for a software engineering project course, in which the students compose their own requirements, behavioral specification, design, and text documents.

It is best to organize this class into four groups:

1. *Requirements.* This group is responsible for defining the requirements of the system to be built.
2. *Specification.* This group is responsible for explaining how a system that meets those requirements would look to the user.

3. *Design.* This group is responsible for designing a system that acts as the specification group says the system should act.
4. *Prototype.* This group is responsible for building a working prototype system to display the behavior that the specification group indicates. To the extent feasible the prototype should match the design created by the design group, although time limitations may force the two to diverge.

The instructor serves both as executive director to make sure that fundamentally the work stays on track and as "coach" to help the various groups to carry out their assignments. For the most part the instructor should avoid making project-related decisions because it is best for students to assume that responsibility.

Parallel Development

The major difficulty in teaching such a course is the limited time. Even in semester courses there is never enough to prepare the documents in the normal sequential manner. To accommodate the constraints documents must be developed in parallel. Starting out, of course, is difficult in that the students who initiate the behavioral specification, the design, and the prototype have nothing on which to base their work.

Yet parallel development is not so impossible as it seems. First, it takes a while for students to get used to the idea of producing large documents. Most computer science students, although comfortable in program preparation, are not accustomed to writing English and often have a rough time at the beginning. Second, it takes a while for most students to develop a feeling for these documents. The students must work with outlines and attempt to express their ideas within the framework that the documents provide before they can be confident enough to begin constructive work.

Third, in general it is not absolutely necessary to complete a requirements document before beginning a behavioral specification or to have a final behavioral specification before starting a design. With the basic idea in hand of what a system is intended to do it is generally possible to begin the behavioral specification and design. Finally, given a spirit of cooperation between groups and the appropriate perspective of each group's functions, parallel development can be a very positive approach. Each group will understand that the other groups are working out the details that its own group needs to do its work.

Parallel development can be considered constructive rather than disadvantageous. It is often said of large systems that "One should expect to throw the first version away." No matter how disciplined, the mistakes made the first time in a complex development are generally significant enough to determine that a second try would make a qualitative difference in the final product. Parallel development helps to mitigate this effect. If all the documents and a prototype version of the system are developed simultaneously each development group can benefit from the problems encountered and the lessons learned by the others.

Parallel development forces an orientation somewhat different from the usual "waterfall" chart approach. By the use of this methodology each document is driven by documents developed earlier and in turn drives documents developed later; that is, if the requirements document is fixed before work is begun on a specification the specification must be written to satisfy the requirements stated. Similarly, if the specification is fixed before the design is begun the design is forced to implement the system as specified. Of course, it never works out that way; problems that appear in later phases of development force compromises with decisions apparently made earlier once and for all. These compromises are frequently awkward to accommodate and often result in hard feelings.

On the other hand, if it is understood that the documents are to be developed in parallel, that no document drives any other absolutely, and that the primary constraint is that the documents (and the final system) be mutually consistent at the end of development, a spirit of cooperation and mutual respect can prevail. Each group represents an area of expertise that it brings to bear on the result and each group can represent its interests while understanding that the needs of the other groups must be heard.

1. Members of the requirements group think of themselves as experts in the definition of requirements. Although requirements are important, without the help of the other groups nothing would be accomplished except stating the problem. They recognize that the other groups are building a system to meet the needs that they discover and document. This, of course, is traditional.

2. Members of the specification group think of themselves as experts in the definition of user-friendly systems that are easy to understand and that fit the user's needs. All that is needed is to have someone tell them what the user is like and what the user-needs are. As far as they are concerned, these are details to be incorporated into their system-defining methodology. The specification group needs the requirements group to provide these details and the design and prototype groups to carry them out, but they do not see themselves as subservient to any other group any more than a photographer is subservient to the person whose portrait is taken or to the people who develop the prints.

3. Members of the design group think of themselves as experts in the design of clean, easily understandable, well organized software. All that is needed is to have someone tell them what the software is supposed to do. As far as they are concerned, that is a detail to which they apply their design methodology. The design group needs the specification group to work out the details of the appearance of the system and the requirements group to determine whether there are other constraints to be obeyed, but they do not see themselves as subservient to any other group any more than any other designers are subservient to their customers. Designers bring to their jobs a knowledge of design techniques that permit more to be accomplished than nondesigners might imagine and design constraints that set a limit to what is possible. Because of these talents designers often have the final say on what the product will do.

4. Finally, members of the prototype group think of themselves as experts in getting a computer to do what they want it to do. They can make the compiler do tricks, they can make terminals stand on their heads, they can make the operating system sit up and beg, and they have a bagful of other tools that turn building systems

into a video game. They see the computer as their instrument and they are its masters. All they need is to have someone give them specifications to animate. The prototype group needs the other groups to tell them what to make the system do, but they do not see themselves as subservient to any other group any more than a musician is subservient to the designer of the instrument to be played or to the composer of the music selected. Without the prototype group there would be nothing but paper; so in the end they are most important. Also, it is often the prototype group with their knowledge of making an actual system do real work that makes or breaks a design and the system that depends on it.

In developing documents in parallel, we expect a lot of rewriting, but we should expect a lot in any case. The rewriting due to too early commitments made in the behavioral specification and design is really no different from that required to remedy internal consistency or errors made in interpreting or ignoring parts of earlier documents.

The experience of working in the dark proves valuable. Students appreciate all the more the importance of a well defined set of requirements when developing a behavioral specification and the value of a behavioral specification when constructing a design. They learn from their own experience that the decisions they make from a limited perspective can sometimes get them into trouble and lead to extra work. They also learn how each part of the job depends on all the other parts.

Finally, it is important to point out the effects of working in groups. Even students who began with no particular affinity for the groups in which they were placed find themselves identifying with their thinking and defending them against the other groups whenever disagreements occur. Each group develops its own identity and way of working together and each group tends to think of itself as beset by the others. Groups bargain with one another to make their own jobs easier and to bend the system to their own points of view. It is the instructor's obligation to hold up a mirror so that the students who are emersed in the process can recognize and understand the interpersonal and intergroup interactions as well as the technical issues.

Choosing the Project

The class project is selected from proposals written by the students themselves. At the start of the semester each student submits a project proposal which consists of a one-paragraph summary of its contents and one-paragraph preliminary versions of the requirements, behavioral specification, and design documents. The class reviews these proposals and selects one as its semester project.

Project Organization

In addition to the four groups the class is divided into committees in a form of matrix organization. Each group has a representative on each committee. The groups are responsible for the technical work, the committees, for management and quality assurance.

1. *Management Committee.* The management committee directs project planning, scheduling (and rescheduling as necessary), configuration management, and any other management and administrative functions that can be handled by the students themselves.

The management committee must produce a development plan, which consists mainly of a schedule of milestones for the production and review of documents. A typical plan calls for the production during the semester of four review versions and one final version of the documents to reflect complete descriptions of partial versions (i.e., "builds") of the final system.

To the extent that the instructor wants to give the students this responsibility the management committee may also allocate class time. A great deal of class time may be devoted to reviewing documents. This is best if the class is scheduled in part as a laboratory and in part as a lecture. It is also best if the lecture and laboratory times run sequentially to allow work started in one period to continue into the other.

Scheduled laboratory time is also important for another reason: the students must be assured of having some time to work together in groups. Individual student schedules are often so different that whole groups can get together only during class time. A total of five to seven scheduled hours a week works reasonably well. For a three-unit class the hours are best scheduled as two hours of lecture and three hours of laboratory or as one hour of lecture and six hours of laboratory.

Management is also responsible for additional structuring; for example, it is often useful for each group to appoint a liaison with the other groups. It is the job of the liaison to meet periodically with groups to which he or she is assigned and to report back about the work of these groups. It is a good practice to schedule a time for these meetings to take place during the next to last minutes of the class session once a week. During that time the liaisons can make their reports to their home groups.

Finally, management is responsible for keeping track of the progress of all projects. A good method is to appoint one of its own members as *project historian*. The project historian's job is to record decisions made by the committees, to keep track of the memberships of the groups and committees, and to record and follow up on action items. The historian should submit three weekly reports to the management committee which would contain the following:

a. *The current project plan and the events scheduled until the end of the semester.*

b. *A record of decisions made and brief rationales for the decisions.* These records play an important part in all projects. Recording the decisions made saves a great deal of energy in preventing the same issues from being reargued and rethought. In large projects, decisions are frequently made but inadequate records are kept. The result is that later, when it is necessary to refer to a decision (What did we decide about that question last week?) there is no reliable means of confirming it. Either no one remembers clearly or everyone remembers clearly but differently. A definitive, authoritative record of decisions made will save a great deal of wasted time.

c. *A list of problem reports yet to be resolved and other items that require action, each item to include the name of the individual or group responsible for that item.* This list serves as a subsidiary project agenda, parallel to the schedule of document submissions and reviews of the project plan. Each action item should have a scheduled completion date. The actual date of completion should also be recorded as part of the project history. This list of action items provides a quick reference to trouble spots. Any document with an unusually large number of unresolved problem reports should receive extra attention.

2. *Quality Assurance.* The quality assurance committee is responsible for guaranteeing the overall quality of the project. It sets documentation standards and determines that each group's work is internally consistent and complete and that the groups are consistent with one another.

The quality assurance activity is concerned with consistency within and between documents and with document quality in general; for example, clarity of the writing.

As its first task quality assurance should define a problem report form for recording problems and a separate problem report should be issued for each problem. Problem reports may be initiated by anyone in the class (e.g., a member of a working group) as a way of anticipating a potential problem with another group's document and by the Quality Assurance group itself. There should be a problem report issued for each problem quality assurance (or anyone else) finds in reviewing the documents. All problem reports should be submitted to quality assurance for coordination (to prevent multiple reports from being issued on the same problem) and then issued formally by the quality assurance group.

Many students find writing difficult. Therefore, in addition to its responsibility for document consistency, quality assurance should note awkward phrasing and help to put the documents into simple language.

For the most part it is unreasonable to expect quality assurance to produce documentation or programming standards that go beyond the document outlines included in this book. Students do, however, sometimes enjoy defining document formatting conventions such as the use of alphabetic and roman numerals for section numbers, indentation guidelines, and page-heading style guides.

3. *Presentation Committee.* Each group makes periodic review presentations of its draft documents to the rest of the class. These presentations should be conducted with relative formality and records should kept of any action items that result from these reviews. The presentation committee, which runs the reviews, should make sure that it stays on the subject and gets bogged down neither in resolving problems nor in name calling or other nonproductive activities. It is also responsible for pointing out the issues raised at the review sessions for recording by the project historian and must set standards for the presentations, provide a moderator, and in general encourage the groups to take the matter seriously.

4. *Technical Committee.* The technical committee makes recommendations to the management group about technical topics that are not within the purview of individual groups. It is responsible for analyzing conflicts that develop between groups and for providing the objective, technical information on which resolutions may be based. As an example, it may be that a number of systems could conceivably serve

as host to a project. After listening to arguments for each of the possible systems this committee would report to management on the issue. As another example, different groups may have different ideas about the scope of a project. Although this is primarily a requirements matter, other groups may have good reasons for wanting to define scope somewhat differently. It may, for example, be possible to include in the initial version of the system features that the requirements group would otherwise have left as future enhancements. Again after listening to all sides the technical committee must produce a report that defines the problem. If it so desires it may also make recommendations.

These committees should schedule meetings approximately once a week. If there is no work be be done at a meeting it can be adjourned quickly, but it is important that meetings be scheduled regularly so that the committees can develop an understanding of their responsibilities.

THE PLACE OF THIS BOOK IN THE SOFTWARE ENGINEERING LITERATURE

Much as been written recently on the subjects covered or touched on by this book. The general field is commonly known as *software engineering*. Because it involves the application of technology to the development and maintenance of software, it naturally divides itself into two subfields: software engineering technology and software engineering management. This book is about software engineering technology; it is not about software engineering management. As such it contrasts with many of the other books in the field.

In addition to the technology/management dimension, books in this field may be measured on a formality scale. At the informal end they are primarily anecdotal; at the other end they are formal and theoretical. This book is in the middle; it presents material rigorously but does not sacrifice intuitive understanding. It is intended to be accessible to practitioners in the field and to senior level computer science undergraduate students.

The following annotated bibliography discusses selected works. They were included if they are widely known or of significant interest. With this contour map of other publications current books may be located. References in the annotations are to other works in this list.

Representative Books

Bersoff, E.H., V.D. Henderson, and S.G. Siegel, *Software Configuration Management*, Prentice-Hall. Englewood Cliffs, New Jersey, 1980. Software product assurance through configuration management. A thorough, management oriented, presentation.

Booch, G., *Software Engineering with Ada*, Benjamin-Cummings, Menlo Park, California, 1983. The "object-oriented" approach to software design based on ideas

in Abbott (1983). Relevant to the design of software architecture and algorithms. Presented on a level accessible to most programmers.

Brooks, F.P., *The Mythical Man Month*, Addison-Wesley, Reading, Massachusetts, 1975. Good advice about managing software development.

Jackson, M. *Principles of Program Design*, Academic, New York, 1975. A semiformal approach to program design that maps the syntactic structure of a program's input into a structure for an algorithm to process that input. Similar to Warnier (1974).

Jensen, R.C. and C. C. Tonies, Eds., *Software Engineering*, Prentice-Hall, Englewood Cliffs, New Jersey, 1979. A relatively informal collection of articles. Primarily management oriented. Topics include management issues, algorithm design (structured programming), testing, security, and legal aspects.

Jones, C.B., *Software Development: A Rigorous Approach*, Prentice-Hall, Englewood Cliffs, New Jersey, 1980. An attempt to make formal approaches to algorithm specification and verification accessible to practitioners. Focuses on abstraction and formal specification and verification. Too difficult for most programmers.

Metzger, P.W., *Managing a Programming Project*, Prentice-Hall, Englewood Cliffs, New Jersey, 2nd ed. 1981. A step-by-step guide through the development cycle. Greatest emphasis is on managing the programming and testing activities.

Myers, G., *Composite Structured Design*, Van Nostrand, New York, 1978. A relatively informal data flow approach to program design. Similar to Yourdon (1979).

Pressman, R.S., *Software Engineering, A Practitioner's Approach*, McGraw-Hill, New York, 1982. A survey that covers most topics in the standard life cycle. Relatively informal. Includes separate chapters on each of the traditional structured design approaches: stepwise refinement, cohesion versus coupling of subprograms (called "modules") (Stevens, 1974), and dataflow (bubble charts) (Myers, 1978; Yourdon and Constantine, 1979), and data structure (Jackson, 1975; Warnier, 1974).

Procedings: Specifications of Reliable Software, IEEE Catalog No. 79CH1401-9C, 1979. A comprehensive collection of original and review papers on methods of formal specification and verification.

Tausworthe, R.C., *Standardized Development of Computer Software*, Prentice Hall, Englewood Cliffs, New Jersey, 1979, two volumes. Discussions and outlines of standardized documents developed for NASA at JPL to cover the entire development cycle. Medium formality. Covers management and technical aspects. A good presentation of the state of the art in industry in 1979.

Warnier, J.D., *Logical Construction of Programs*, Van Nostrand, New York, 1974. A semiformal approach to program design that maps the syntactic structure of a program's input into a structure for an algorithm to process that input. Similar to Jackson (1975).

Yourdon, E. and L. Constantine, *Structured Design*, Prentice-Hall, Englewood Cliffs, New Jersey, 1979. A relatively informal dataflow approach to program design. Similar to Myers (1979).

Zelkowitz, M.V., A.C. Shaw, and J.D. Gannon, *Principles of Software Engineering and Design*, Prentice-Hall, Englewood Cliffs, New Jersey, 1979. A somewhat disjointed collection of articles that covers aspects of the development cycle. Focuses primarily on programming techniques, that is, algorithm implementation.

Representative Papers

Abbott, R.J., "Program Design By Informal English Descriptions," *Communication of the ACM*, November 1983. An approach to algorithm design based on the idea of identifying common nouns with abstract data types. Forms the basis of Booch (1983).

Alford, M.W., "A Requirements Engineering Methodology for Realtime Processing Requirements," *IEEE Transactions on Software Engineering*, **1**(3):60-69, 1977. Describes SREM, a tool for high-level system design by functional decomposition. Intended for requirements analysis. Competes with PSL/PSA.

Blazer, R. and N. Goldman, "Principles of Good Software Specification and Their Implications for a Specification Language," *Proceedings: Specifications of Reliable Software*, IEEE Catalog No. 79CH1401-9C, 1979. Discussion of requirements for languages for system specification. See also Goldman (1980).

Goldman, N. and D.S. Wile, *A Database Foundation for Process Specification*, USC-ISI Report RR-80-84, 1980. A language for system specification derived from the requirements in Balzer (1979). A promising approach that uses techniques and notations from relational databases and formal specification to build a model of the system specified.

Guttag, J.V., E. Horowitz, and D.R. Musser, "Abstract Data Types and Software Validation," *Communications of the ACM* **21**(12):1048–1064, 1978. The formal algebraic approach to abstract data types. Relevant to the design of software architecture and algorithms.

Heninger, K.L., J.W. Kallander, J.E. Shore, and D.L. Parnas, *Software Requirements for the A-7E Aircraft*, NRL Memorandum Report 3876, Naval Research Laboratories, Washington, D.C., 1980. An informal but rigorous specification of an existing software system. Demonstrates that software can be well documented without excessive formalism.

Luckham, D.C. and W. Polak, "A Practical Method of Documenting and Verifying Ada Programs with Packages," *Proceedings of the ACM-SIGPLAN Symposium on the Ada Programming Language*, 1980, pp. 113–122. An attempt to apply the algebraic method of abstract specification (Guttag, 1978) to Ada packages.

Parnas, D.L., "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, **15**(12):1053–1058, 1972. One of the original works that defined software architecture as something other than algorithm design.

Stay, J.F., "HIPO and Integrated Program Design," *IBM Systems Journal*, 15(2):143–154, 1976. An early approach to stepwise refinement and dataflow design.

Stevens, W., G. Myers, and L. Constantine, "Structured Design," *IBM System Journal* 13(2):1974. A discussion of criteria for decomposing programs into subprograms. Introduces the notions of *coupling* and *cohesion*.

Teichrow, D. and E. Hershey, "PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Transactions of Software Engineering* 3(1):41–48, 1977. A tool for high-level system dataflow design. Intended for requirements analysis. Competes with SREM. Lately has evolved into an automated document production system capable of integrating text with automatically generated reports and diagrams.

ACKNOWLEDGMENTS

I should like to acknowledge the work of all those who preceded me in this field. I hope that I have done justice to their conditions. In addition, I should like to acknowledge Jane Radatz for her enthusiastic support during the early work of writing this book and for her technical assistance, Ric Cowan for his aid, especially during the final period, and the Aerospace Corporation and California State University, Northridge, for their support.

RUSSELL J. ABBOTT

Chatsworth, California
August 1985

CONTENTS