N89-16327

Integrating Automated Structured Analysis and Design with Ada Programming Support Environments

Alan Hecht and Andy Simmons
Cadre Technologies Inc.
222 Richmond St.
Providence, R.I. 02903
(401) 351-5950

Abstract

Ada Programming Support Environments (APSE) include many powerful tools that address the implementation of Ada code. These tools do not address the entire software development process. Structured analysis is a methodology that addresses the creation of complete and accurate system specifications. Structured design takes a specification and derives a plan to decompose the system sub-components, and provides heuristics to optimize the software design to minimize errors and maintenance. It can also promote the creation of reusable modules. Studies have shown that most software errors result from poor system specifications, and that these errors also become more expensive to fix as the development process continues. Structured analysis and design help to uncover errors in the early stages of development. APSE tools help insure that the code produced is correct, and aid in finding obscure coding errors. However, they do not have the capability to detect errors in specifications or to detect poor designs.

This paper will describe how an automated system for structured analysis and design, **teamwork®**, can be integrated with an APSE to support software systems development from specification through implementation. These tools complement each other to help developers improve quality and productivity, as well as to reduce development and maintenance costs. Complete system documentation and reusable code also result from the use of these tools. Integrating an APSE with automated tools for structured analysis and design provide capabilities and advantages beyond those realized with any of these systems used by themselves.

Introduction

Developing quality software on time and within budget has proven to be a difficult task. Statistics gathered by the government and private industry have shown that software development projects are difficult to control [Boehm 81]. This results in software systems that can be extremely expensive with less than adequate performance.

These problems have fostered several solutions. The U.S. Department of Defense performed an analysis of its software applications, concentrating on problems inherent with coding and implementation. This analysis resulted in the development of Ada [DoD 81]. Other people were addressing problems associated with software requirements. The results of this effort has resulted in the development of several software development methodologies based on the concept of a software lifecycle [DeMarco 78, Page-Jones 80, for example].

The DOD identified a problem specific to the *implementation* of embedded systems. There were a number of languages in use and there was potential that this number would continue to grow. The lack of a standard implementation language resulted in money being spent on new compilers (which were not significantly better), training and maintenance. The development of the Ada programming language was seen as an answer to this problem. In addition, the solution would include a programmer's environment, or toolkit, called the "APSE."

APSE

The Ada Programming Support Environment (APSE) was proposed to augment the Ada language [DoD 80, Stennig 81]. It includes tools such as the compiler, language sensitive editor, and debugger. These tools are designed with knowledge about the structure of Ada and are focused on the implementation phase of software development. The APSE presents a uniform development environment to aid Ada programmers.

APSEs help solve the problems of implementing embedded systems that were recognized by the DOD. A reduction in software development costs can be realized as a result of making the implementation phase more efficient. However, the problem still remains that APSEs do not thoroughly address the other phases of software development.

Software Development Lifecycle

Recent work has focused on gathering statistics from case studies of projects [Ramamoorthy 84]. At least half of the projects had problems which originated in the requirements or functional specification (see Figure 1). To help put this in perspective, we can view the software development process as divided into five (sometimes overlapping) phases: analysis, design, implementation, test and verification, and maintenance.

The analysis phase is concerned with understanding *what* a system is supposed to do. The result is supposed to be an *implementation independent* description or abstract view of the system to be developed. The product of analysis is a requirements specification (sometimes called a functional specification) that describes the system function and important constraints.

The design phase addresses *how* the system is to be implemented. It is concerned with the physical aspects of the system. The optimal structure of the various software modules and how they interface is determined. Ideally, the design information should be complete enough to reduce the implementation effort to little more than a translation to a target programming language.

The implementation phase is concerned with producing executable code. Knowledge of both the design and the target environment is incorporated to produce the final system software. All the physical aspects of the system are addressed during implementation.

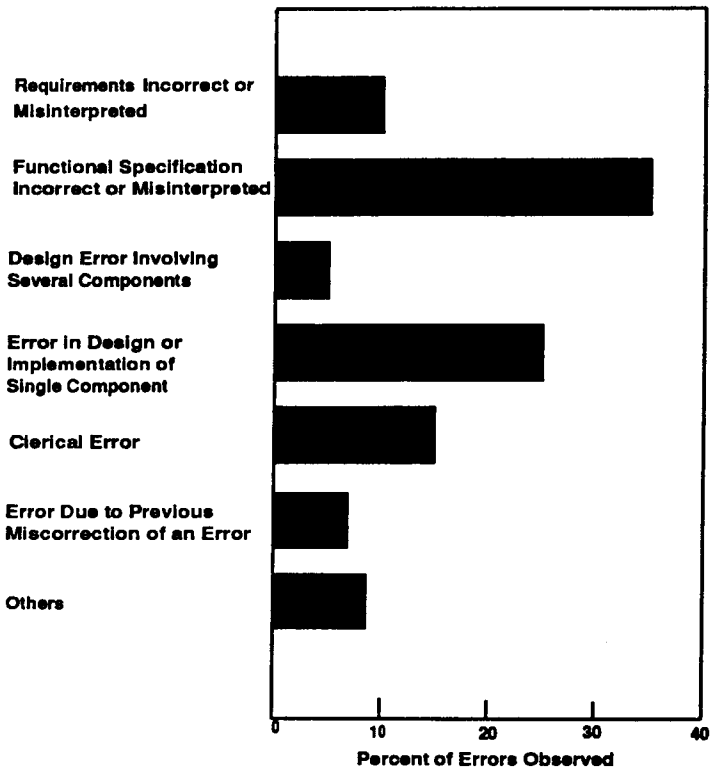

Figure 1: Sources of Errors[1].

Information from the previous three phases is used in the testing and verification phase. Test plans can be derived from specifications and designs [Boehm 84]. The testing phase verifies that the software conforms to the specification and that the code is correct. The best that test and verification techniques can do is prove that a program is consistent with its specification. They cannot prove that a program meets the user's desires [Wulf 80]. This means that extra care must be taken during analysis to insure that the specification is a complete and correct reflection of what the user really wants. This can be accomplished through methods that support checks for consistency and clearly communicate system requirements. Teamwork/SA supports one such method, and it will be discussed later in this paper.

Bug fixes and adaptations which result from experience with the software are activities of the maintenance phase. At this point the software is being used -- the ultimate test. Users will come across errors or suggestions as they gain experience with the software. Maintenance procedures must handle the orderly evolution of the code. They must insure that changes will not have deleterious effects on the system.

A study by [Boehm 84] showed that errors detected later in the development life cycle cost more to fix than errors detected during analysis (See Figure 2). Figure 1, discussed previously, showed that the majority of errors in a software project can be traced to requirements and specification problems. These facts illustrate the value of spending more time at the beginning of a project, performing analysis. This can be difficult for programmers and users to accept as both may be anxious to see code being produced [Ramamoorthy 84]. These ideas have only recently become well understood and brought into practice.

[1]Adapted from [Ramamoorthy 84]

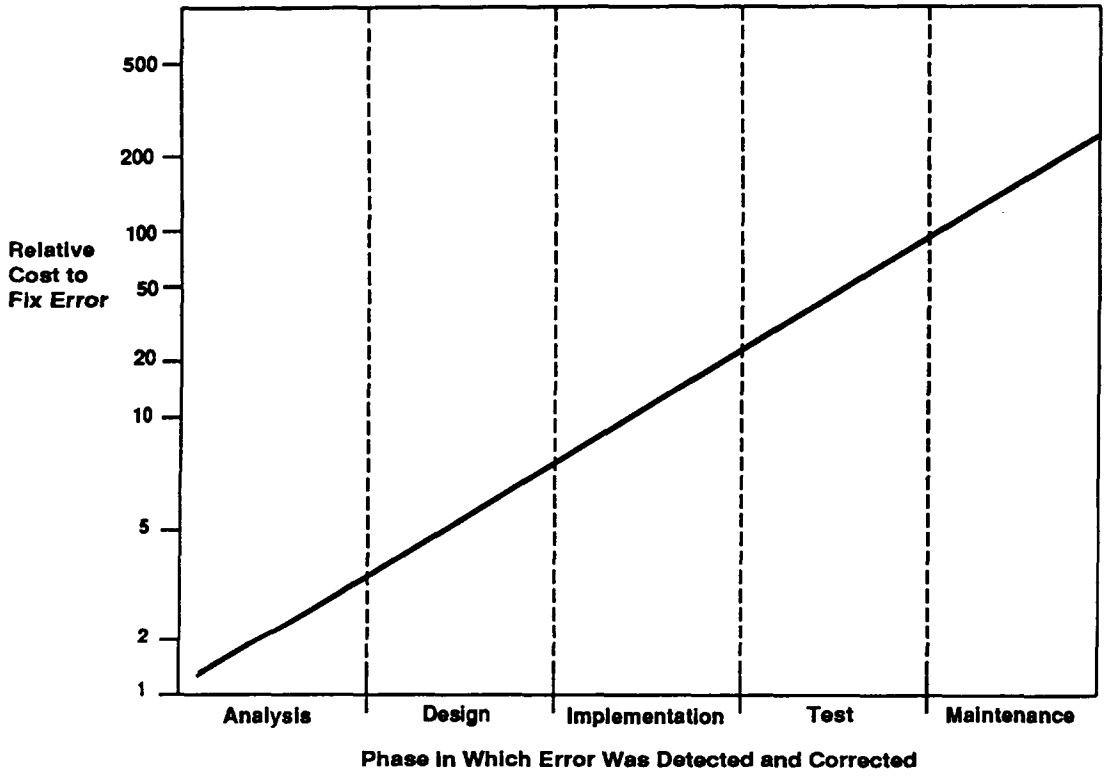

Figure 2: *Cost of Error Versus When it is Detected*[2].

Many approaches and methodologies utilize the concept of the software life cycle. In particular, structured analysis (which refers to several methods [Gane 79, DeMarco 78, Ross 77]) addresses the beginning phase of requirements analysis.

Structured Analysis

Structured analysis views a system from the perspective of the data flowing through it. The function of the system is described by processes that transform the data flows. Structured analysis takes advantage of information hiding through successive decomposition (or top down) analysis. This allows attention to be focused on pertinent details and avoids confusion from looking at irrelevant details. As the level of detail increases, the breadth of information is reduced. The result of structured analysis is a set of related graphical diagrams, process descriptions, and data definitions. They describe the transformations that need to take place and the data required to meet a system's functional requirements.

De Marco's approach [DeMarco 78] consists of the following objects: *data flow diagrams, process specifications,* and a *data dictionary* (See Figure 3).

Data flow diagrams (DFDs) are directed graphs. The arcs represent data, and the nodes (circles or bubbles) represent processes that transform the data. A process can be further decomposed to a more detailed DFD which shows the subprocesses and data flows within it. The subprocesses can in turn be decomposed further with another

[2]Adapted from [Boehm 84]

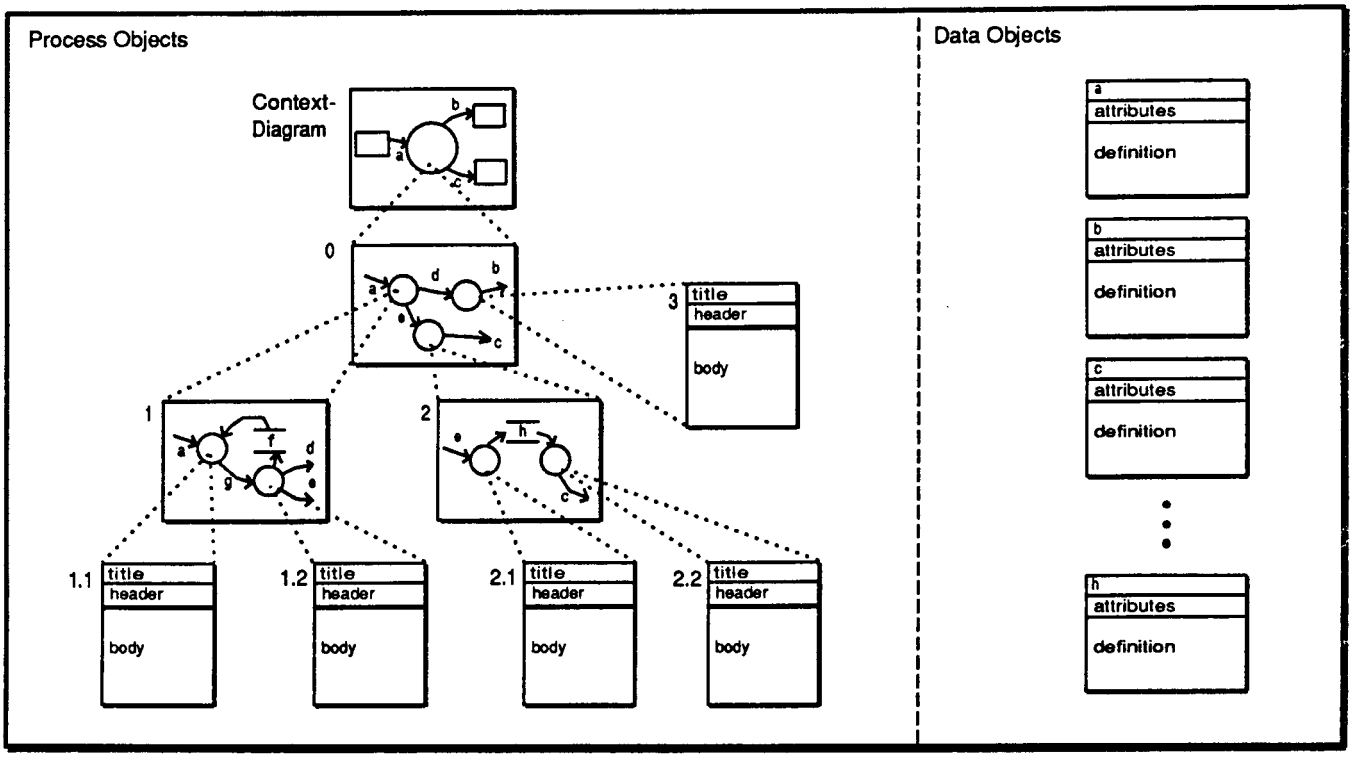

Figure 3: Analysis Model Objects

set of DFDs until their functions can be easily understood. Functional primitives are processes which do not need to be decomposed further. Functional primitives are described by a process specification (or mini-spec). The process specification can consist of pseudo-code, flowcharts, or structured English. The DFDs model the structure of the system as a network of interconnected processes composed of functional primitives.

The data dictionary is a set of entries (definitions) of data flows, data elements, files, and data bases. The data dictionary entries are partitioned in a top-down manner. They can be referenced in other data dictionary entries and in data flow diagrams.

Military standard 2167 [MilStd2167 85] requires that systems be specified in a top down manner using a structured approach similar to that described above. The high level of process and data abstraction inherent in structured analysis is compatible with the objectives of the Ada language. Where it is desirable to take an object-oriented approach to design [Booch 86, Cox 84], structured analysis helps to define classes and data hierarchies or data structure. For procedural approaches, structured analysis works well with structured design.

Structured Design

Structured design addresses the synthesis of a module hierarchy [Page-Jones 80]. The principles of cohesion and coupling are applied to derive a optimal module structure and interfaces. Cohesion is concerned with the grouping of functionally related processes into a particular module. Coupling addresses the flow of information, or parameters, passed between modules. Optimal coupling reduces the interfaces of modules, and the resulting complexity of the software.

Page-Jones' approach [Page-Jones 80] consists of the following objects: *structure charts, module specifications* and a *data dictionary.*

The structure chart shows the module hierarchy or calling sequence relationship of modules. There is a module specification for each module shown on the structure chart. The module specifications can be composed of pseudo-code or a program design language. The data dictionary is like that of structured analysis.

At this stage in the software development lifecycle, after analysis and design have been performed, it is possible to automatically generate data type declarations [Belkhouche 86], and procedure or subroutine templates.

Automating Structured Analysis and Design

Hardware CAD/CAM systems have contributed to the development a systems with higher levels of complexity, performance and reliability, at costs previously unattainable through purely manual design efforts. This is sparking interest in automating the software development process.

Teamwork is a set of automated tools for systems analysis and design. They can support many simultaneous users working on the same project or even many projects. They take advantage of features provided by the latest workstation technology, offering complete support of the DeMarco structured analysis techniques and the Page-Jones structured design techniques. Graphical diagrams are created using syntax-directed editors that incorporate model building rules. Its interactive graphics package supports a high resolution bit-mapped display, mouse and keyboard. Modern user interface techniques are used, including a multi-window display and context specific pop-up and pull-down menus.

Multiple, simultaneous views of a specification or a design can be displayed by **teamwork/SA** (See Figure 4). It has simple commands for traversing through the various parts of a model. Model objects may be entered in any order. The graphics editors allow diagrams to be easily produced and edited. Diagrams as well as components of diagrams are automatically numbered and indexed. These features eliminate many manual, time consuming tasks.

Project information is retained in a project library, through which individuals can simultaneously share model information and computer resources. Team members linked over the network can access the same information for review. Multiple versions of model objects are retained in the library. Team members can independently renumber and repartition diagrams, which allows exploration of different approaches to describe a system.

Teamwork's consistency checker detects specification errors within and between data flow diagrams, data dictionary entries, and process specifications, and design errors within and between structure charts and module specifications. Typical errors and inconsistencies include DFD balancing errors (data flows from one diagram that do not match data flows to a related diagram) and undefined data dictionary entries. The consistency checker uses the semantics and rules of structured analysis and structured design. Checking is performed "on-demand", which allows the analyst and designer to work top-down, bottom-up, or any other way. It encourages the exploration of partial models that may be (during the intermediate stages of building the model) incomplete or incorrect. The speed and depth of checking in **teamwork/SA** helps produce consistent and correct specifications, which can be used with the tools provided in an APSE.

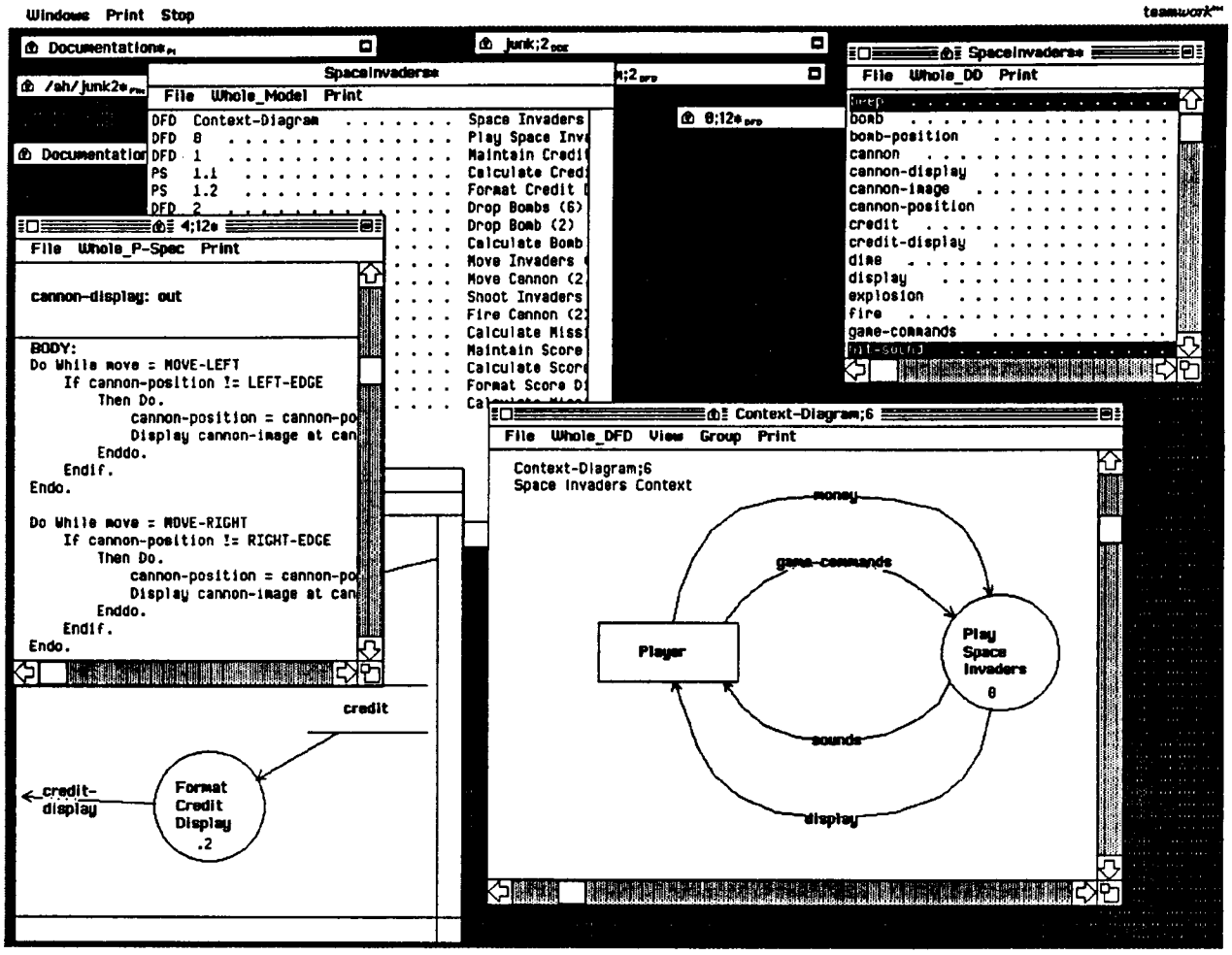

Figure 4: teamwork/SA Desktop

Integration of Teamwork with APSE

Teamwork was designed to allow the information it captures to be utilized for many purposes. These include packaged specifications, project status reports, configuration management, system documentation, and test plans. The information is captured as the specification and design are created. As described above, teamwork helps to insure consistency of the information as a system progresses through these phases. The relationships between the various representations of processes, data, and modules are recorded in the project library. This information may be selectively retrieved and reformatted with post processors which can be developed for a variety of software development tasks, such as the following:

- Producing data type declarations and procedure templates specific to the syntax of any language, especially Ada.
- Generating test plans.
- Generate formatted requirement documents, such as MIL-STD 2167.

In addition, by combining an APSE with teamwork, the complete lifecycle documentation can be consistently maintained, from requirements to code listings. If any change is made to any piece of a project, that change can be reflected in the corresponding parts of the project.

Conclusion

APSEs help reduce some of the problems associated with software development, especially during the implementation phase. Automated analysis and design environments address the problems associated with poor specifications and software system structure. Either tool by itself is better than totally manual development. The combination of all these tools can provide automated support for the entire software development lifecycle, insuring consistency and reducing errors and developments costs.

References

[Belkhouche 86] Belkhouche, B., and J.E. Urban.
Direct Implementation of Abstract Data Types from Abstract Specifications.
IEEE Transactions on Software Engineering :649-661, May, 1986.

[Boehm 81] Boehm, Barry W.
Software Engineering Economics.
Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[Boehm 84] Boehm, Barry W.
Verifying and Validating Software Requirements and Design Specifications.
Software , January, 1984.

[Booch 86] Booch, G.
Object-Oriented Development.
IEEE Transactions on Software Engineering :211-221, February, 1986.

[Cox 84] Cox, Brad J. .
Message/Object Programming: An Evolutionary Change in Programming Technology.
Software :50-61, January, 1984.

[DeMarco 78] DeMarco, Tom.
Structured Analysis and System Specification.
Yourdon Press, New York, 1978.

[DoD 80] US Dept. of Defense.
Requirements for Ada Programming Support Environments - Stoneman.
February, 1980

[DoD 81] US Dept. of Defense.
Reference Manual for the Ada Programming Language - Proposed Standard Document.
July, 1981

[Gane 79] Gane, Chris and Trish Sarson.
Structured Systems Analysis: Tools and Techniques.
Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1979.

[MilStd2167 85]
Military Standard - Defense System Software Development DOD-STD-2167.
June, 1985

[Page-Jones 80] Page-Jones, M.
The Practical Guide to Structured Systems Design.
Yourdon Press, New York, 1980.

[Ramamoorthy 84]
Ramamoorthy, C.V., et. al.
Software Engineering: Problems and Perspectives.
Computer :191-209, October, 1984.

[Ross 77] Ross, D. and R.E. Schoman Jr.
Structured Analysis for Requirements Definition.
IEEE Transactions on Software Engineering SE-3(1), January, 1977.

[Stennig 81] V. Stennig et. al.
The Ada Environment: A Perspective.
Computer :26-36, June, 1981.

[Wulf 80] Wulf, W.A.
Trends in the Design and Implementation of Programming Languages.
Computer :14-23, June, 1980.