A Controlled Experiment on the Impact of Software Structure on Maintainability

H. DIETER ROMBACH

Abstract—**This paper describes a study on the impact of software structure on maintainability aspects such as comprehensibility, locality, modifiability, and reusability in a distributed system environment. The study was part of a project at the University of Kaiserslautern, West Germany, to design and implement LADY, a *LA*nguage for *D*istributed s*Y*stems. The study addressed the impact of software structure from two perspectives. The language designer's perspective was to evaluate the general impact of the set of structural concepts chosen for LADY on the maintainability of software systems implemented in LADY. The language user's perspective was to derive structural criteria (metrics), measurable from LADY systems, that allow the explanation or prediction of the software maintenance behavior. A controlled maintenance experiment was conducted involving twelve medium-size distributed software systems; six of these systems were implemented in LADY, the other six systems in an extended version of sequential Pascal. The benefits of the structural LADY concepts were judged based on a comparison of the average maintenance behavior of the LADY systems and the Pascal systems; the maintenance metrics were derived by analyzing the interdependence between structure and maintenance behavior of each individual LADY system.**

Index Terms—**Complexity metrics, comprehensibility, controlled experiments, distributed systems, language comparisons, locality, maintainability, modifiability, reusability, software structure.**

I. Introduction

THE growing complexity of software requirements on one hand and the rapid advances and dropping cost of microelectronics on the other hand make it more and more attractive to use distributed systems. But this opportunity cannot be used effectively unless appropriate development and maintenance methodologies are available. Today, in most cases, development and maintenance in distributed system environments are supported by techniques and tools originally intended for sequential problem solution and centralized system architectures. No consensus has been reached within the software community as to what an appropriate software development and maintenance environment for distributed systems should look like.

Manuscript received August 30, 1985; revised June 23, 1986. This work was supported in part by the Ministry of Research and Technology of the Federal Republic of Germany as part of the DISTOS project at the University of Kaiserslautern; preparation of the final version of this paper was supported in part by NASA, Goddard Space Flight Center, under Grant NSG-5123. Computer support was provided in part by the Computer Science Center at the University of Maryland.

The author was with the Department of Computer Science, University of Kaiserslautern, D-6750 Kaiserslautern, West Germany. He is now with the Department of Computer Science, University of Maryland, College Park, MD 20742.

IEEE Log Number 8610900.

This paper emphasizes two important methodological aspects: 1) what are appropriate concepts to structure software for distributed systems and how should these concepts be represented as language features in order to support maintainability of the resulting software systems, and 2) what structural criteria (metrics), measurable from a software system, can be used to explain or predict its maintenance behavior. The maintenance aspects of interest in this paper are comprehensibility (how hard is it to understand a given software system), locality (how many software units are affected by each maintenance cause), modifiability (how hard is it to change a given software unit), and reusability (how effectively can already existing documentation and experience be used during maintenance).

The study presented in this paper was part of the DISTOS project (*DIST*ributed *O*perating *S*ystem), started in 1980 at the Department of Computer Science of the University of Kaiserslautern, West Germany. The main part of this project was the design and implementation of LADY, a *LA*nguage for *D*istributed s*Y*stems [18], [20].

The objectives of the presented study were to analyze and answer whether the LADY language designers met their maintenance oriented goals with the chosen concepts and related LADY features, and to determine how developers can make effective use of these LADY concepts and features in order to develop maintainable software by measuring structural data that allow the explanation or prediction of maintenance behavior. The approach chosen to answer all those questions was to evaluate data gained from a controlled maintenance experiment with 12 medium-size software systems. Controlled experiments are justified as a first step for verifying certain hypotheses. Such an experiment allows concentrating on certain software factors of interest, in this case structural factors, by controlling others. On the one hand, results derived from a controlled experiment cannot be transferred into a different project environment; such results always have to be verified in a second step in this different project environment. On the other hand, some phenomena might never be detected in a real environment because their effect might be overwhelmed by the effect of some other factor.

There are a number of related projects investigating the appropriate structural concepts for distributed systems (e.g., [1], [14], [16], and [21]). Besides all other differences between these projects and the DISTOS project, DISTOS is the first project (to the author's knowledge)

0098-5589/87/0300-0344$01.00 © 1987 IEEE

that presents quantitative analysis results showing to what degree the original goals were met by the chosen language approach.

The following sections detail the software structure model chosen for LADY, the software maintenance model used, the specific analysis goals and questions for the study, a detailed description of the experimental approach, and analysis results.

II. Software Structure Model

One traditional model of structuring software for distributed systems is to extend structural models existing in sequential languages, such as Pascal, by including 1) an additional module type for representing parallelism (process) and 2) mechanisms for passing messages among processes.

The LADY model of structuring software for distributed systems is very different. This model allows the explicit representation of distribution aspects and different intensities of coupling among software units. One hypothesis is that this model allows a more natural representation of software for distributed systems, which in turn is expected to improve quality aspects such as maintainability and reusability.

In this context, the most interesting structural concepts of the LADY model are:

1) Three Different Refinement Levels to Describe Distributed Software Systems:

a) At the system level, a distributed system is described as a set of distribution units (teams) communicating via message passing. This communication mechanism is required because two teams, as units of distribution, might be located at different nodes of a distributed system without shared memory.

b) At the team level, each team is structured as a set of processes communicating via shared memory (monitors). This communication mechanism is possible and appropriate because processes of one team will always be located on one single node of a distributed system.

Specification levels a) and b) are intended to support locality and modifiability by providing appropriate means for modularization, and comprehensibilty by allowing the development and maintenance personnel to understand the system structure in two consecutive steps.

c) At the module level, each module type (process, monitor, class, or procedure) is described by a Pascal-like algorithm.

2) Two Different Communication Concepts: Potentially distributed units (teams) communicate via messages; units that can never be distributed (processes within one team) communicate via common memory. The availability of both communication concepts allows a natural representation of different types of cohesion among processes.

3) Separation of Specification and Implementation: This concept supports comprehensibility and reusability by separating the use of a unit from its implementation.

4) Highly Parameterized Types of Structural Units: This concept supports the extensive reuse of software [12].

TABLE I
Structural Language Concepts

Structural Concepts	Implementation Languages	
	LADY	C-TIP
Number of system description levels	3	2
System level	a system is described as a set of teams interconnected via channels between input/output ports (SPECIFICATION)	a system is described as a set of processes interconnected via channels between input/output ports (SPECIFICATION)
Team level	a team (the unit of distribution) is described as a set of processes interconnected via monitors (SPECIFICATION)	---------
Module level	Each module (process, monitor, class, and procedure) is described by a sequential Pascal algorithm	Each module (process, class, and procedure) is described by a sequential Pascal algorithm
Communication concept	2 different concepts: - Exchange of messages (between teams) - Common memory (within teams)	1 single concept: - Exchange of messages
Separation of specification and implementation	YES (for teams and modules)	NO
Strong typing	YES (supported by library system)	YES (not supported by library system)
Parameterization	YES (supported by language, except for number of ports and number of structural units)	YES (not supported by language)

In Table I the structural concepts of LADY and C-TIP are compared.

III. Software Maintenance Model

Software maintenance is defined as the performance of those activities required to keep a software system operational and responsive after it is accepted and placed into production [10]. Maintenance activities can be divided into three categories: *corrective*, *adaptive*, and *perfective*. Whereas *corrective* maintenance refers to changes usually triggered by a failure of the software detected during operation, *adaptive* and *perfective* maintenance refer to changes due to external changes. Adaptive maintenance is initiated by changes of the operational environment; perfective maintenance is initiated by changes of the requirements.

One important criterion for judging the quality of a method or tool used during maintenance is its effectiveness in either detecting failures, isolating all related faults, or preventing errors while correcting the identified faults. The underlying defect model [15], [24] recognizes the existence of three types of software defects: 1) Errors are defects in the human mind, trying to understand given information, to construct and document solutions, or to apply methods and tools, 2) Faults are the concrete manifestations of errors within the software (probably different types of documents), and 3) Failures are departures of a

software system from its requirements (or intended use respectively) detected during operation. The purpose of the study presented in this paper is to evaluate the impact of a method to structure software for distributed systems and the related language tool (LADY) on all three categories of maintenance activities. The evaluation process is based upon the following software maintenance model:

1) Detection of Needs to Change Software: In the case of adaptive and perfective maintenance this step is trivial; in the case of corrective maintenance software failures have to be detected. In most practical cases only system failures can be detected and it is not trivial to determine whether a system failure is due to a software failure, a nonsoftware failure (e.g., hardware), or whether it is not a failure at all. Erroneously reported failures may be due to inappropriate documentation or differences between the specified and user-expected function of a software system; as a consequence, perfective maintenance activities can be triggered.

2) Isolation of Related Faults: The isolation of *all* software faults having caused a particular failure or that exist because of changed environment or requirements is crucial for the success of a maintenance task. Being aware of the fact that different faults can trigger identical failures, the isolation of *all* faults related to a certain failure can definitely be considered to be a difficult maintenance step.

3) Correction of Faults: The correction of isolated faults should follow the same pattern as development activities: a) *designing possible changes*, b) *implementing one selected change design*, and c) *validating the changed software*. In general there exist different options to correct isolated faults. Different alternatives should be designed; their impact on other parts of the system should be analyzed. Finally a decision concerning the "best" design has to be made. The degree of validation is determined by the changes made.

Only those maintenance aspects are included in this model that are important with respect to the evaluation goal of this study. Therefore, this model, as opposed to other models [10], emphasizes the technical aspects of software maintenance rather than the management aspects such as "scheduling of maintenance tasks" or "releasing changed software."

IV. Analysis Goals and Questions

Quantitative studies require a mechanism for determining what data is to be collected, why it is to be collected, and how the collected data is to be interpreted. The mechanism proposed in [6] calls for 1) determining the analysis goals of interest and 2) refining each goal into a set of questions for the purpose of quantifying the goals of interest. The questions define the specific set of data to be collected.

The two analysis goals of this study are:

1) Determine differences in the maintenance behavior of systems implemented in either one of two languages (LADY and C-TIP) that are different with respect to the incorporated structural concepts and features (language designer's goal).

2) Determine measurable structural criteria (complexity metrics) that allow the explanation or prediction of the maintenance behavior of a LADY system (language user's goal).

A. Analysis Questions Related to Goal 1)

Goal 1) is refined into five analysis questions, to be answered by comparing the average maintenance behavior of the LADY systems and the C-TIP systems:

1) Maintainability: Is the average effort in staff-hours per maintenance task different?

2) Comprehensibility: Is the average isolation effort (effort to decide what to change) in staff-hours per maintenance task, or the average amount of rework (all effort spent for changing already existing documents such as requirements, designs, code, or test plans) per system unit as a percent of all effort spent per unit throughout the lifecycle different?

3) Locality: Is the average number of changed units per maintenance task, or the average maximum portion of the change effort spent in one single unit per maintenance task different?

4) Modifiability: Is the average correction effort in staff-hours per maintenance task and unit different?

5) Reusability: Is the average amount of reused documentation as a percent of all documentation per maintenance task different?

The maintenance behavior is not only evaluated with respect to maintainability, but also with respect to individual steps of the maintenance model. Failure detection is not expected to be impacted by software structure; therefore it is not evaluated in the context of this study. In contrast, the isolation step is evaluated in terms of comprehensibility and locality, and the correction step in terms of modifiability. In addition, the impact of LADY on reusability is analyzed; reuse of software can be expected to improve the quality and productivity of maintenance substantially.

B. Analysis Questions Related to Goal 2)

Goal 2) is refined into analysis questions similar to goal 1) but will be answered by analyzing the impact of the structure of each individual LADY system on its actual measured maintenance behavior:

1) Maintainability: What is the impact of the individual software structure on the average effort in staff-hours per maintenance task?

2) Comprehensibility: What is the impact of the individual software structure on the average isolation effort (effort to decide what to change) in staff-hours, and on the amount of rework as a percent of all effort per maintenance task?

3) Locality: What is the impact of the individual software structure on the average number of changed units per maintenance task and on the average maximum portion of

the change effort spent in one single unit per maintenance task?

4) Modifiability: What is the impact of the individual software structure on the average correction effort in staff-hours per maintenance task and unit?

5) Reusability: What is the impact of the individual software structure on the average percentage of reused documentation per maintenance task?

While questions related to goal 1) address whether there is a difference using either one of the two languages with respect to maintainability, questions related to goal 2) are intended to define structural criteria for good maintenance behavior of LADY systems. This paper emphasizes the answers to questions related to goal 1); only a brief overview is given on possible answers to questions related to goal 2).

V. Experimental Approach

The hypothesis in this paper is that software structure impacts software maintenance. A blocked subject–project study [5] was conducted to verify this hypothesis. This type of study examines the analysis goals and questions across a set of students (performing maintenance tasks) and a set of software systems. The originality of this study comes from the fact that real maintenance experiments were conducted. On the contrary, most of the published results concerning software maintenance behavior are based upon change data collected during development. The underlying but unverified assumption in these papers is that the change behavior during development and maintenance is identical.

A. Maintained Software Systems

Data on the maintenance behavior of 12 medium-size software systems were collected. These systems were developed prior to this study and can be divided into four classes with respect to their requirements and the used implementation language:

1) LADY TSS: Three systems implementing requirements of a time sharing system in LADY.

2) C-TIP TSS: Three systems implementing the identical requirements of a time sharing system in C-TIP.

3) LADY PCS: Three systems implementing requirements of a process control system in LADY.

4) C-TIP PCS: Three systems implementing the identical requirements of a process control system in C-TIP.

The functions of the time sharing system are similar to those of the SOLO operating system [7]. The process control system controls a hierarchically organized system to distribute parcels by ZIP-code.

A brief overview of the systems of each class in terms of size, structure, and development effort is contained in Table II.

TABLE II
Maintained Software Systems
(mean values per system class)

Characteristics	System Classes			
	TSS		PCS	
	LADY	C-TIP	LADY	C-TIP
Length (KLoC)	15.2	10.7	2.5	1.5
Development effort (staff-hours) before testing	257.9	301.3	70.6	87.6
No. of teams (*) (types/objects)	7/16	---	5/9	---
No. of modules (types/objects)	18/42	20/61	12/38	9/30
No. of processes (types/objects)	11/27	11/28	9/27	6/12
No. of monitors (*) (types/objects)	5/11	---	4/12	---

(*) only for LADY systems

B. Experiment

For each of the 12 systems characterized in Table II a series of 50 maintenance tasks was designed: 25 corrective, 10 adaptive, and 15 perfective maintenance tasks. The corrective maintenance tasks were triggered by system failure specifications, the adaptive maintenance tasks by change specifications of the operational environment, and the perfective maintenance tasks by change specifications of the requirements, respectively. Altogether, 600 maintenance tasks were conducted; the maintenance effort per task varied from 15 minutes to more than 1 day.

Whereas the design of the changes of environment and requirements was no problem, the design of the faults to cause the 25 failures was complicated. The goal was to conduct identical maintenance tasks for all twelve software systems. This goal could not be met because of different requirements, peripheral device environments, and different software structure. The compromise was to conduct identical patterns of experiments for all twelve systems. The environment and requirement changes were completely identical for those systems implementing identical requirements (see previous subsection about the maintained software systems). The ideal set of 25 faults (only faults not detectable by compiler) would be representative for faults to be expected during real maintenance for each system, identical across all systems implementing identical requirements, and causing identical failures in all systems. Neither of these requirements can be fulfilled. There exists no accepted theory to predict the number or type of faults that will be detected during use of a system based on development data, nor is it possible to seed systems of different structure with identical faults, and even identical faults in different systems do not necessarily result in identical failures observed during use.

Therefore, the set of 25 failures was different for each system. But for each system the set of faults fulfilled three important criteria:

1) The distribution of faults was identical to the average one detected during development of these systems with respect to some fault classification scheme.

2) The faults affected identical system functions in each system implementing identical requirements, and the faults covered all structural system units at each level.

3) The fault classification scheme used in this study distinguished between data faults, control faults, and computation faults, either affecting only one single unit or affecting more than one unit [24].

C. Maintenance Personnel

The maintenance tasks were carried out by six graduate level research assistants. All of the students gained their knowledge about the maintenance environment (described in the following subsection) from a six month practical software engineering project and classes at the University of Kaiserslautern. Each of the students maintained one time sharing system and one process control system, both implemented in either LADY or C-TIP.

The validity of the results from comparing the maintenance behavior of LADY and C-TIP depends on equally qualified maintenance personnel. The selection process was based on a variety of criteria. The students were ranked according to educational performance (course grades), experience (in industrial projects), and their relative programming talent. The results from the practical project were particularly useful in estimating the relative programming talent of a set of candidate students for this experiment; the relative programming talent was judged based on the time spent for completing the course assignment and the quality of the final product according to some acceptance test procedure. The students ranked one, four, and five were assigned to maintain C-TIP systems, the students ranked two, three, and six were assigned to maintain LADY systems. There was a random assignment of students in each language class to time sharing systems and process control systems; the only restriction was that a student who was involved in the prior development of a particular system (two out of six students) was not allowed to maintain this same system. The selection process was not and could not be completely objective and is therefore subject to criticism.

Another problem which might result in falsifying the results of controlled experiments is the Hawthorne effect [8]. When human beings become aware that certain aspects of their behavior is being monitored, their behavior will change. There is no simple solution to this problem. However, it can be assumed that this effect becomes insignificant if human beings are monitored over a long period of time. This was the case with this controlled experiment. The students involved were used to be monitored over a period of one or two years prior to the experiment; as part of several prior assignments, including the practical course mentioned above, they had to fill out forms identical to those used during the controlled experiment.

Despite this training course, it can be assumed that students were still more familiar with C-TIP than they were with LADY because Pascal is the language used for education at the University of Kaiserslautern. Later in this paper, the analysis results will be interpreted taking this fact into consideration. Each student had two weeks time to become familiar with the systems to be maintained by reading all the available documents and learning to use the systems prior to the experiment.

D. Experimental Environment

The maintenance environments for the controlled experiments were almost identical for both language classes [22]. The target system for all maintained software systems was a network of six Texas Instruments 990/5 microcomputers. Texas Instruments 990/10 minicomputers were used for all maintenance activities. The tool environment can be divided into three parts:

1) A requirements and design documentation tool [26], which can be used to input, validate, change, and document requirements and design decisions. The chosen type of design representation is based on ideas of DeRemer and Kron about module interconnection languages [9]. A refined language version was used in these experiments [23], [26].

2) A separate set of implementation tools for each language, LADY [17] and C-TIP, each consisting of a compiler and an integrated library system to maintain all existing team and module type implementations.

3) A functional test tool, which supports the design of test cases (based on information from the design tool) by selecting test data, creating test beds, running tests, and documenting test results.

The technique used for isolating faults was reading documents. The techniques used for validating the correctness of changes were testing and reading.

E. Data Collection and Validation

Data were collected both to characterize the software structure and to characterize the maintenance behavior.

According to the previously listed analysis goals and questions, the following data had to be collected in order to characterize the *maintenance behavior:*

1) Number of modules changed per maintenance task.

2) Effort (in staff-hours) to isolate what to change per maintenance task.

3) Effort (in staff-hours) to implement changes per maintenance task and unit.

4) Portion of already existing documentation reused per maintenance task.

These data were collected for each maintenance task by forms [22], that are similar to those developed in the Software Engineering Laboratory at NASA/University of Maryland [27]. Validation of the collected data was carried out by the author by meeting with each of the students after each experiment. A sound data validation procedure [2] is crucial for the significance of analysis results.

Data to characterize the *software structure* of each maintained system were collected from each design and implementation document. They were collected from the version available at the end of development and updated after major maintenance changes. These data are easier to validate than maintenance data because they are based on written information in various documents. The collected data from implementation documents can be divided into

two major classes: 1) data related to the *external complexity* of each structural unit, and 2) data related to the *internal complexity* of each structural unit. External complexity is based on an information flow model similar to the one presented by Henry and Kafura [13]; it was refined and extended for use in this specific environment [22], [23].

1) *External complexity* data collected from each unit are:

a) Number of input ports, number of output ports (for processes or teams), number of functional entries (for classes or monitors), and number of parameters per port or entry, *if observing a single unit.*

b) Number of other units explicitly connected to the observed unit via a channel to/from this unit or implicitly connected to the observed unit by exchanging information via other units or using shared assumptions,[1] *if observing a unit as integrated into the whole system.*

2) *Internal complexity* data collected from each unit are:

a) Length of modules: the number of lines of implementation documents excluding pure comment lines. Length of teams: the number of its modules.

b) Structure of modules: the number of nonsequential control flow constructs. Structure of teams: the number of communication channels between its modules.

c) Intensity of the environment embedding of a unit: the number of interface accesses occurring within the unit implementation documents. These interface accesses are either send/receive operations to activate communication channels (in the case of teams or processes), or procedural calls of other units (in the case of monitors, classes, and procedures).

The corresponding structural design data were collected from function oriented designs [23].

F. Data Evaluation

All questions related to goal 1), i.e., whether there is a difference between LADY systems and C-TIP systems with respect to maintainability, were answered by comparing the average maintenance data across the four different classes of maintained software systems. The statistical test used in this part of the study was the nonparametric Mann–Whitney U-test; it allows to determine the significance of maintenance differences between two different classes of systems. All differences reported in the following section are of significance level 0.05 or better; in the case of comparing two populations of three systems each the best possible significance level is 0.05. In addition, statistical results need always be supported by evidence of causal relationships; in this study we want to be sure that the observed differences with respect to maintenance are really caused by structural differences and not differences among the maintenance personnel or the influence of other factors. Conducting a sound controlled experiment with constant environmental and personnel factors suggests that the only variable factor, *structure*, is responsible for observed differences.

All questions related to goal 2), i.e., which structural criteria (metrics) allow the explanation or prediction of the maintenance behavior, were answered by determining the correlation between different aspects of maintenance behavior (see section on goals and questions) and different aspects of software structure for all software units. The statistics used in this part of the study were the nonparametric Spearman correlation coefficient and related significance test [28]. The different aspects of software structure under investigation were all individual internal complexity aspects, all external complexity aspects which may or may not include implicit relationships among units (as listed in the previous subsection), and all possible (multiplicative) combinations of external and internal complexity aspects. These analyses finally resulted in identifying meaningful metrics for this particular environment.

VI. Quantitative Comparison of Two Language Concepts

The data collected from the controlled experiment as well as analysis results are presented according to the analysis questions related to goal 1). All presented differences between mean values are significant at the 0.05 level or better.

1) *Maintainability* is characterized by the average effort in staff-hours per system and maintenance task.

The differences in Table III between the average correction effort of LADY and C-TIP systems are not significant for any type of maintenance cause (failures, environment changes, and requirement changes). In contrast, LADY systems require significantly ($p < 0.05$)[2] less isolation effort than C-TIP systems for each class of maintenance causes (32.9 percent less for failures, 16.7 percent less for environment changes, and 28.9 percent less for requirements changes). The ability to understand system complexity in three steps and the availability of separate specification and implementation documents for each unit in LADY are assumed to be mainly responsible for the detected difference. This conclusion is supported by the data in Table IV.

This table contains a subset of the data in Table III (only for failures) partitioned into the class of more complex time sharing sytems and the class of less complex process control systems.

[1]Shared assumptions are all types of dependencies among software units which are not explicitly represented as interface data or global data. A frequent example is an implicitly shared assumption concerning buffer size.

[2]The symbol p will be used to stand for significance level according to the Mann–Whitney U-test.

TABLE III
MAINTAINABILITY
(mean values per language and maintenance cause class)

Data	Language and Maintenance Cause Classes					
	Failures		Environm. Changes		Requirem. Changes	
	LADY	C-TIP	LADY	C-TIP	LADY	C-TIP
Effort (in staff-hours) per system and maintenance task for						
- isolation + correction	1.59	2.04	13.62	15.93	8.27	11.24
- isolation	0.98	1.46	10.56	12.68	6.37	8.96
- correction	0.61	0.58	3.06	3.25	1.90	2.28

TABLE IV
MAINTAINABILITY
(mean values per system class)
(only for failures)

Data	System Classes			
	TSS		PCS	
	LADY	C-TIP	LADY	C-TIP
Effort (in staff-hours) per system and maintenance task for				
- isolation + correction	2.10	2.80	1.09	1.29
- isolation	1.33	2.09	0.63	0.84
- correction	0.77	0.71	0.45	0.45

TABLE V
COMPREHENSIBILITY
(mean values per system class)

Data	System Classes			
	TSS		PCS	
	LADY	C-TIP	LADY	C-TIP
Rework per system (in percent) of				
- all development effort	57.83	44.36	56.91	40.59
- only development effort related to requirements, system design, and team design documents	50.29	76.83	58.40	58.52

TABLE VI
LOCALITY
(mean values per system class)

Data	System Classes			
	TSS		PCS	
	LADY	C-TIP	LADY	C-TIP
Number of changed units per system and maintenance task				
- teams and modules	2.71	2.47	2.08	2.46
- only modules	1.92	2.47	1.62	2.46
Highest effort spent in one single unit per system and maintenance task (in percent of all effort per system and maintenance task)	85	74	70	69

The advantage of using LADY with respect to isolation effort is only significant ($p < 0.05$) for time sharing systems. The conclusion from this observation is that the three-level description provided by LADY is obviously an advantage for complex systems, but not an advantage or even a disadvantage in the case of very small systems. If the systems are so small that each team only consists of one single process, the redundant team level description is, of course, not helpful.

2) *Comprehensibility* is determined by the average effort necessary to understand what needs to be changed. This portion of the maintenance effort (isolation effort) is contained in Tables III and IV. Therefore the conclusions are identical to those for maintainability.

Another characterization of comprehensibility is the amount of rework done; the more a problem is misunderstood, the more rework can be expected. Besides the effort spent on changes during development, all maintenance effort (at least in the case of failures) is considered rework. Because rework is only of interest in relation to nonrework, it can only be evaluated properly based on data from the prior development of the software systems under investigation. Valid development data were available for all development phases excluding test phases. The effort data collected during testing proved to be less valid due to the nature of testing and debugging.

The overall rework is significantly ($p < 0.05$) higher for LADY systems than for C-TIP systems (57.83 versus 44.36 percent). Because the main differences between LADY and C-TIP deal with interunit structure, it seems worthwhile to look into the portion of rework spent in requirements, system and team design documents before unit test. For this rework portion a significant ($p < 0.05$) difference for time sharing systems can be observed: 50.29 percent rework for LADY systems, but 76.83 percent for C-TIP systems. This result is supported by the subjective impression of the author that students developing LADY systems understood the overall system better before starting with detailed module design than did students developing C-TIP systems. The rework data from test phases are not as reliable, but in general the rework portion over the entire development process is about 20 to 25 percent higher (in absolute numbers) than the rework percentages excluding testing, as contained in Table V.

3) *Locality* is characterized in this study by the average number of changed modules per system and maintenance task, and the average maximum portion of the change effort concentrated in one single unit per maintenance task. As a consequence, locality of a system is good if 1) only a small number of units is affected by each maintenance task, or 2) when more than one unit is affected, a high percentage of effort is concentrated in one single unit.

The numbers of affected units per maintenance task in Table VI can give a wrong impression because in the case of LADY systems specification units (teams) are counted in addition to algorithmic units (modules). Therefore, only the average number of modules should be compared.

The average number of affected modules is significantly ($p < 0.05$) lower in the case of LADY systems (e.g., 1.92 versus 2.47 for time sharing systems). Beyond it, if more than one module is affected, significantly ($p <$

TABLE VII
MODIFIABILITY
(mean values per system class)

Data	System Classes			
	TSS		PCS	
	LADY	C-TIP	LADY	C-TIP
Effort (in staff-hours) spent per system, module, and maintenance task caused by a failure for				
- isolation + correction	1.09	1.13	0.67	0.52
- only correction	0.40	0.28	0.28	0.18

0.05) more effort is concentrated in one single module on average (e.g., 85 versus 74 percent for time sharing systems). It seems that modules in LADY are better used to support principles such as data abstraction or information hiding.

Another interesting aspect is that during the development of LADY systems (without manipulated change causes), it seldom was more than one team changed. Teams proved to be especially suited for encapsulating functional units of systems. This result is even more remarkable because students had much more experience in applying the structural concept integrated in C-TIP than the one integrated in LADY.

4) *Modifiability* is characterized by the average correction effort per unit and maintenance task. The corresponding numbers in the case of failures in Table VII are determined easily by dividing either the overall effort or only the correction effort (see Table IV) by the product of number of failures (25) and average number of changed modules per failure (see Table VI).

As expected, the average effort in staff-hours to correct a single module seems to be relatively independent of the chosen implementation language. Whether the correction process was conducted well can not be decided based on the experimental design; the number of faults introduced during correction or not found during isolation was not large enough to derive statistically significant results. Nevertheless, the results with respect to comprehensibility in terms of rework have indicated that, at least during development, the amount of rework is significantly lower in the case of LADY systems. Based on these results from development it is hypothesized (without verification) that the structural concepts of LADY make it easier to understand a complex software system and make it less likely to miss existing faults or to introduce new faults during maintenance.

5) *Reusability* is characterized by that portion of system documents that does not need to be developed because it was developed previously. Most of the differences in Table III in the case of changes of environment can be explained by the LADY language concept of "device teams," which allows easy integration of new or different devices into a system. The differences in Table III in the case of requirements changes can be partly explained by LADY's concept of parameterized unit types. There was substantial reuse of team and module types; new module objects could often be integrated without any modifications.

Reuse of team types is more complicated. The current LADY version allows no change of the number of module objects and the port interface by parameters. Therefore, even if a team type could be completely reused with respect to its function, each environment change that requires a port interface change results in a new team type. Due to this restricted parameterization concept the reusability of teams is limited.

Overall, about 60 percent of all new integrated modules in the case of changes of requirements are reused unchanged. Another 20 percent required only port interface changes. In contrast, about 45 percent of the new C-TIP modules are reused (unchanged or slightly changed) for requirements changes. In the case of failures, it is hard to describe the level of reusability. In this study only reuse of implementations (by creating new objects of parameterized types) was measured. If the term reuse is thought to include the reuse of design documents or even the reuse of experience, then, naturally, the level would be higher.

VII. Complexity Metrics for Maintenance

Only highlights of the results of the analysis questions of goal 2) are presented in this section; the presentation is restricted to results concerning LADY systems. For a more detailed discussion of all questions about the impact of software structure (as measured from implementation or design documents) on maintenance behavior (in the case of LADY and C-TIP as implementation languages) the reader is referred to other papers [22], [23].

To characterize briefly the impact of the structure of LADY systems on their maintenance behavior, the suitability of three classes of complexity metrics is discussed to explain or predict the different maintenance aspects. The three classes are: internal (or code oriented) complexity metrics (length, structure, and intensity of embedding), external (or system structure oriented) complexity metrics (information flow between units with or without implicit flows), and hybrid complexity measures (combinations of metrics of the other two classes). First, the best metric is presented for each maintenance aspect. It is noted that the combined complexity metrics are much better than either internal or external complexity metrics. In addition, it is discussed whether the exclusion of nonexplicit measurable information flow has disadvantages. It should be mentioned that all of the following results are validated by determining the nonparametric Spearman rank correlation coefficient. Therefore, these metrics are only valid on an ordinal scale. All correlations presented are significant at the 0.001 level or better.

1) Maintainability: The average effort (in staff-hours) per unit and maintenance task is best explained or predicted by those combined complexity metrics that measure the external complexity by information flow including implicit flows and measure the internal complexity by either length or structure (average r: 0.9; $p < 0.001$)[3].

Combined complexity metrics without implicit flows (average r: 0.85; $p < 0.001$) are still sufficiently good. Even the internal complexity metrics length and structure (average r: 0.7; $p < 0.001$) and external complexity metrics (average r: 0.75; $p < 0.001$) are useful to explain or predict maintainability.

2) Comprehensibility: The average isolation effort (in staff-hours) per maintenance task is best explained or predicted by those combined complexity metrics which measure the external complexity by information flow including implicit flows and measure the internal complexity by either length or structure (average r: 0.85; $p < 0.001$). Combined complexity metrics without implicit flows (average r: 0.74; $p < 0.001$) are still sufficiently good. Even the internal complexity metrics length and structure (average r: 0.66; $p < 0.001$) and external complexity metrics (average r: 0.8; $p < 0.001$) are useful to explain or predict comprehensibility. The results are similar to those for maintainability, except that the influence of implicit information and the dominance of external complexity compared to internal complexity is more obvious.

3) Locality: The average number of changed units per maintenance task, and the average maximum portion of the change effort spent in one unit per maintenance task are best explained or predicted by those combined complexity metrics which measure the external complexity by information flow including implicit flows and measure the internal complexity by the intensity of environment embedding (average r: 0.92; $p < 0.001$). Again, combined complexity metrics without implicit flows (average r: 0.83; $p < 0.001$) are still sufficiently good. Locality is the only maintenance aspect where internal complexity metrics such as length and structure show no correlation. Instead, the newly defined internal complexity metric characterizing the intensity of the environment embedding (measured, e.g., for processes, by the number of send and receive calls) shows surprisingly high correlation (average r: 0.78; $p < 0.001$). The external complexity metrics (average r: 0.8; $p < 0.001$) are still very useful to explain or predict locality.

4) Modifiability: The average effort (in staff-hours) per maintenance task and module is best explained or predicted by those combined complexity metrics which measure the external complexity by information flow including implicit flows and measure the internal complexity by either length or structure (average r: 0.83; $p < 0.001$). Combined complexity metrics without implicit flows (average r: 0.81; $p < 0.001$) are still sufficiently good. The internal complexity metrics length and structure (average r: 0.8; $p < 0.001$) are especially useful to explain or predict modifiability. The external complexity metrics are very unreliable to predict modifiability (average r: 0.62; $p < 0.001$). The differences, when compared to all other maintenance aspects, are that modifiability is less influenced by implicit information flows and that internal complexity aspects are more dominant than external complexity aspects.

5) Reusability: No metric to explain or predict reusability could be validated by the available data. The subject impression of the author is that reusability is influenced much more by good specifications separated from the implementation, extensive use of parameterization, and the availability of tool supported document administration rather than by structural aspects. The first two criteria are especially supported by LADY, and this might explain the better reusability results in Section VI. However, this assumed impact could not be quantified.

The important results can be summarized as follows:

1) The impact of structure on software behavior could be quantified for all defined maintenance aspects except for reusability.

2) The best results were obtained from combined complexity metrics. The external complexity included implicitly measurable information flows; the internal complexity was characterized by length or structure, except for locality where the intensity of unit embedding was most sensitive.

3) The combined complexity metrics that exclude implicitly measurable information flows still showed sufficiently good results. This fact allows the use of completely automatable metrics.

4) Often, very simple metrics such as the internal complexity metrics did very well in explaining or predicting maintenance behavior. This was especially true in the case of modifiability.

5) The fact that external complexity metrics were sufficiently good, particularly for comprehensibility and locality, encouraged the author to use these metrics to explain or predict maintenance behavior based upon software structure measured from design documents [23].

To explain or predict maintenance aspects reliably, a metric vector has to be observed including at least the presented types and a metric measuring the balance between average system, team, and module level complexity. Systems which were very different in the way complexity was distributed between these three levels showed different maintenance behaviors. Although it is not yet known what constitutes a good distribution, it is evident that the distribution of structural complexity across different description levels influences maintenance (especially comprehensibility).

The presented metrics can be readily used in projects by establishing lower and upper bounds (baselines) for the complexity of units in a specific environment. If the value for a given unit exceeds the given boundaries, it should be examined to determine whether this unusual complexity value is justified or not. As a consequence, either the corresponding unit has to be changed, or if the unit is not going to be changed, at least it should be watched closely in the future. In order to make effective use of these com-

[3]The symbols r and p will be used to stand for Spearman correlation coefficient and significance level, respectively.

plexity metrics in software projects they have to be computed automatically.

VIII. Conclusions

The newly developed implementation language for distributed systems with its characteristic structural concepts, LADY, proved to be better suited to develop maintainable software than the trivial extension of sequential Pascal (Table I). Especially for complex systems, the language design decisions to describe a software system on three different refinement levels and to separate specification and implementation of each unit had a positive impact on maintainability. As the locality results [see Section VI-3)] show, the unit types for distribution (teams) and separate compilation (modules) are well suited to structure software according to principles such as information hiding and data abstraction. The strong typing combined with high parameterization of units seems to improve reusability of units. Based on the fact that the students who carried out the controlled experiments are more experienced in using Pascal than LADY, one can speculate that the LADY-specific results will become even better as experience using the LADY concepts increases (learning curve).

For all maintenance aspects except reusability the results could be stated quantitatively in terms of complexity metrics. The fact that most of the maintenance aspects can be explained and predicted sufficiently well by the use of only explicitly measurable information flows between units makes it possible to automate these metrics. This is a prerequisite for using metrics in practical project environments and, thereby, obtaining additional validation results.

Before the results can be used in different environments they have to be revalidated in these environments. Results with respect to goal 1) can be used for setting up criteria for selecting an appropriate implementation language for a distributed software project or for setting up an evaluation framework for language development groups. Results with respect to goal 2) can be used as management tools during development and maintenance. After having established baselines concerning the relation between software structure and maintenance behavior for a given environment, such metrics can be used as tools indicating abnormally high or low structural complexity. Such indications should lead to a careful analysis of whether the complexity is justified or whether better solutions exist. The fact that the metrics can be automated and applied in very early design phases make them a very powerful management tool.

Further research in this field will be concentrating on the validation of these results in new projects and the development of metrics measuring the actual degree of fulfillment of desirable structural principles, such as information hiding, within a given language. Research of the latter type is being conducted in the case of Ada® at the University of Maryland [3], [4], and [11].

At the University of Kaiserslautern a follow-up project, the INCAS project (INCremental Architecture for distributed Systems), was started in 1983; this project aims at the development of a comprehensive methodology for the design of locally distributed systems [19], [25]. As part of the INCAS project, an improved version of LADY was developed. Some of the improvements were stimulated by the results presented in this paper.

Acknowledgment

I would like to express my thanks to Prof. J. Nehmer, Department of Computer Science of the University of Kaiserslautern, West Germany, for his initial encouragement and many subsequent helpful comments; M. Clev, E. Jergens, U. Marx, H. Neumann, A. Schwartz, R. Wagner, and V. Wilke, graduate students at the University of Kaiserslautern for their careful experimental work and data collection that this paper is based on; NASA, Goddard Space Flight Center, for its support of the preparation of the final version of this paper; and C. Loggia Ramsey and J. Ramsey, University of Maryland, for their constructive comments on a previous version of this paper published at the Maintenance Conference in Washington, October 1985. Finally I want to thank the Guest Editor, N. Schneidewind, and the referees for their thorough criticism and for numerous suggestions.

References

[1] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe, "The Eden system: A technical review," Dep. Comput. Sci., Univ. Washington, Seattle, Tech. Rep. 83-10-05, Oct. 1983.

[2] V. R. Basili, "Data collection, validation, and analysis," in *Tutorial on Models and Metrics for Software Management and Engineering*, IEEE Catalog No. EHO-167-7, 1981, pp. 310–313.

[3] V. R. Basili and E. E. Katz, "Metrics of interest in an Ada environment," *IEEE Comput. Soc. Workshop Software Eng. Technol. Transfer*, 1985.

[4] V. R. Basili, E. E. Katz, N. M. Panlilio-Yap, C. Loggia Ramsey, and S. Chang, "Characterization of an Ada software development," *Computer*, vol. 18, no. 9, pp. 53–65, Sept. 1985.

[5] V. R. Basili, R. W. Selby, Jr., and D. H. Hutchens, "Experimentation in software engineering," Dep. Comput. Sci., Univ. Maryland, College Park, Tech. Rep. TR-1575, Nov. 1985; also *IEEE Trans. Software Eng.*, vol. SE-12, pp. 733–743, July 1986.

[6] V. R. Basili and D. M. Weiss, "A methodology for collecting valid software engineering data," *IEEE Trans. Software Eng.*, vol. SE-10, pp. 728–738, Nov. 1984.

[7] P. Brinch Hansen, "The SOLO operating system," *Software—Practice and Experience*, vol. 6, no. 2, pp. 141–200, Apr.–June 1976.

[8] J. Brown, *The Social Psychology of Industry*. Baltimore, MD: Penguin Books, 1954.

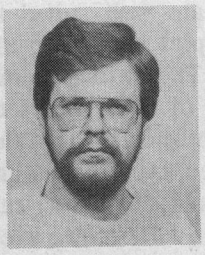
[9] F. DeRemer and H. H. Kron, "Programming-in-the-large versus programming-in-the-small," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 80–86, June 1976.

[10] Federal Information Processing Standards, *Guideline on Software Maintenance*, U.S. Dep. Commerce/National Bureau of Standards, Standard FIPS PUB 106, June 1984.

[11] J. D. Gannon, E. E. Katz, and V. R. Basili, "Metrics for Ada packages: An initial study," *Commun. ACM*, vol. 29, no. 7, July 1986.

®Ada is a registered trademark of the U.S. Department of Defense (Ada Joint Program Office).

[12] J. A. Goguen, "Parameterized programming," *IEEE Trans. Software Eng.*, vol. SE-10, pp. 528–543, Sept. 1984.
[13] S. Henry and D. Kafura, "Software structure metrics based on information flow," *IEEE Trans. Software Eng.*, vol. SE-7, pp. 510–518, Sept. 1981.
[14] R. C. Holt, *Concurrent Euclid, the Unix System, and Tunis*. Reading, MA: Addison-Wesley, 1983.
[15] *Standard Glossary of Software Engineering Terminology*, IEEE, New York, IEEE Standard 729-1983, 1983.
[16] B. Liskov, "On linguistic support for distributed programs," *IEEE Trans. Software Eng.*, vol. SE-8, pp. 148–159, May 1982.
[17] R. Massar, "LADY—A language for distributed operating systems: Design and implementation," Ph.D. dissertation, Dep. Comput. Sci., Univ. Kaiserslautern, West Germany, July 1984.
[18] J. Nehmer, R. Masser, W.-F. Racke, H. D. Rombach, and R. Schrapel, "DISTOS: A method for constructing distributed operating systems," Dep. Comput. Sci., Univ. Kaiserslautern, West Germany, DISTOS Project Rep., Apr. 1982.
[19] J. Nehmer, C. Beilken, D. Haban, R. Massar, F. Mattern, H. D. Rombach, F.-J. Stamen, B. Weitz, and D. Wybranietz, "The multicomputer project INCAS—Objectives and basic concepts," SFB 124, Univ. Kaiserslautern, West Germany, Tech. Rep. 11/85, 1985.
[20] J. Nehmer and D. Wybranietz, "LADY—A language for the design of distributed operating systems," in *Proc. ACM SIGOPS Workshop Operat. Syst. in Comput. Networks*, Ruschlikon, Switzerland, Jan. 1985.
[21] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel, "LOCUS: A network transparent, high reliability distributed system," in *Proc. 8th Symp. Operat. Syst. Principles*, Dec. 1981, pp. 169–177.
[22] H. D. Rombach, "Quantitative evaluation of software quality characteristics based on system structure," Ph.D. dissertation, Dep. Comput. Sci., Univ. Kaiserslautern, West Germany, June 1984.
[23] ——, "Software design metrics for maintenance," in *Proc. 9th Annu. Software Eng. Workshop*, NASA, Goddard Space Flight Center, Greenbelt, MD, Nov. 1984.
[24] V. R. Basili and H. D. Rombach, "Tailoring the software process to project goals and environments," Dep. Comput. Sci., Univ. Maryland, College Park, Tech. Rep. TR-1728, Nov. 1986.
[25] H. D. Rombach, "The multicomputer project INCAS—Objectives, basic concepts, and experience," in *Proc. Pacific Comput. Commun. Symp.*, Seoul, Korea, Oct. 1985.
[26] H. D. Rombach and K. Wegener, "Experiences with a MIL design tool," in *Proc. 8th Conf. Programming Languages and Program Development*, Zurich, Switzerland, Mar. 1984.
[27] SEL, "Software engineering laboratory (SEL): Data base organization and user's guide," NASA, Goddard Space Flight Center, Greenbelt, MD, Rep. SEL-81-102, July 1982.
[28] S. Siegel, *Nonparametric Statistics for the Behavioral Sciences*. New York: McGraw-Hill, 1955.



H. Dieter Rombach was born in West Germany. He received the B.S. degree in mathematics from the University of Karlsruhe, West Germany, in 1975, the M.S. degree in mathematics and computer science from the University of Karlsruhe in 1978, and the Ph.D. degree in computer science from the University of Kaiserslautern, West Germany, in 1984.

From 1978 to 1979 he was a Research Staff Member in the Institute for Technical Data Processing, Nuclear Research Center, Karlsruhe. From 1979 to 1984 he was a faculty member with the Department of Computer Science at the University of Kaiserslautern. He is currently an Assistant Professor of Computer Science at the University of Maryland, College Park. His research interests include software methodologies, measurement of the software process and its products, and distributed systems.

Dr. Rombach is a member of the IEEE Computer Society, the Association for Computing Machinery, and the German Computer Society (GI).