Software Engineering: Multi-person Development of Multi-version Programs

David Lorge Parnas

Middle Road Software
Ottawa, Ontario Canada

Abstract. A brief description of the early days of "Software Engineering" precedes a discussion of the various meanings that have been proposed for that term. The differences between Software Engineering and Programming are described by discussing the tasks beyond programming that are expected to be performed by Software Engineers. Legislators and Educators are challenged to take steps to make Software Development a profession by establishing entrance standards for professionals and quality standards for products.

Keywords: Software Development, Software Engineering, Professional Education.

1 Introduction

In the 1960s, a few Computer Scientists began to use the phrase "Software Engineering" without a clear definition [11, 5]. They were using the term as an expression of hope rather than as a description of an established field. They expressed the hope that, in the near future, people who developed software would be as prepared for practising their profession just as Professional Engineers were prepared for theirs. It was hoped that they would be able to construct their products with the discipline and professionalism that are supposed to characterize Engineering.

- Most of the people who were constructing software at that time were not well-prepared for their work by their education. Many had been educated as scientists or mathematicians; their education had been focussed on the creation and organization of knowledge. Building software is not the creation of knowledge; it is the creation of products. Those who were Engineers had not been taught about software.
- Software was not being produced using the fairly formal process used in construction projects and other areas of Engineering. The precise documentation and mathematical analysis that was part of the work of Engineers in the traditional disciplines (Civil, Mechanical, Electrical,...) was also missing. Programming was practiced as an intuitive process; some called it an art.

Few of those who introduced the use of the term, "Software Engineering", knew much about the engineering profession or engineering education, (which is quite different from education for scientists and mathematicians) but they thought that developing software was more like engineering than science.

C.B. Jones and J.L. Lloyd (Eds.): Festschrift Randell, LNCS 6875, pp. 413–427, 2011.

The failure to provide a concrete definition of the subject has resulted in confusion that persists to this day. There are many interpretations of the term being used today. Among them:

- programming,
- designing programming languages,
- construction of data base and other business systems, and
- managing large, complex, technical projects,
- developing software for pay,
- Professional Engineers whose specialty is constructing software-intensive systems.

The most accurate title for the people that are now called "Software Engineers" would be "Paid Software Developers". They are professionals in the sense that they are paid for what they do, which is develop software but they have no credentials that are relevant to software development, the vast majority of the ones who are Professional Engineers gained that credential in another discipline and then migrated into software development without significant education about software design.

For each of the traditional engineering disciplines, there is agreement on a core body of knowledge, which comprises the skills and knowledge that all of those licensed to practice in that discipline must have. No such body of knowledge has been identified for software developers. There are a number of proposals but they do not have the broad acceptance that the core body of knowledge for disciplines like Civil Engineering has received.

In this paper, I will use the term, "Software Engineer" to mean Paid Software Developer. Only when we have reached agreement on the core body of knowledge that is required, and there is a licensing scheme that insures that licensees have that knowledge, can we use the term, "Professional Software Engineer".

2 The Search for a Definition of Software Engineering

When the term "Software Engineering" was introduced, many asked a simple question."How is "Software Engineering different from programming"? Some of those who asked this question were simply skeptical and wondered if those who used this phrase were simply inventing a new term to attract more attention and funding. Others were asking it rhetorically, to suggest that there was no such field.

Many unsatisfactory descriptions of the difference were proffered. Among them:

- Software Engineers develop system software (e.g. compilers and operating systems).
- Software Engineers are conscious of cost.
- Software Engineers are conscious of the need to deliver on schedule

None of these responses was satisfactory.

- Those who had initiated the use of the phrase were not thinking only of programmers who wrote system software.
- All programmers must be conscious of costs.
- Most of us work under time pressure.

These are not the factors that distinguish a Software Engineer from other programmers. Two answers struck me as insightful and, complete.

2.1 Brian Randell's Answer

The best answer to the questions posed above was provided by Brian Randell when he described Software Engineering as "The multi-person development of multi-version programs". This pithy phrase implies everything that differentiates software engineering from other programming.

The focus of computer education has long been on algorithm design. When people teach programming, it is assumed that a program will be written by a single person. Moreover, they often assume that the ultimate user of the program will be the programmer. No time was spent on the possible need for several versions of the program and the consequent need to "design for change". It was assumed that since software was not hardware, a program could not be hard to change. Nothing is taught about the extra work needed if your program will be used by others or must be part of a multi-person project.

2.2 Fred Brooks' Answer

A few years later, there was another insightful answer. Fred Brooks' classic 1974 book, "The Mythical Man-Month" [4] discusses the same issue from a different perspective. In that book, a diagram identifies 2 dimensions which I will call "productizing" and "integration". The diagram shows that by integrating a program with others, one moves from a program to what Brooks calls "a programming system". By testing, documenting, and preparing a program for use and maintenance by other people, one converts a program to "a programming product". Brooks' formulation too makes it clear that there is much more required of a Software Engineer than programming skill. It should go without saying that a Software Engineer must be able to program, but must be able to do many other things.

2.3 The Remainder of This Paper

The remainder of this paper will expand on the Randell and Brooks observations. It first discusses the many types of people who will be involved with a programming systems product, and then the reasons for the production of many versions of the product. Finally, it describes the tasks beyond programming that must be carried out by professional software developers. [26]

3 Multi-person Development of Software

If we accept Brian Randell's characterization of software, a software product will always be important to more than one person; there will be at least one developer and at least one user[1]. In many cases, there will be more than one developer. Success requires careful and extensive communication between those people.

[1] I leave it to philosophers to decide whether a product that was intended to sell, but fails to attract any users should be considered software.

3.1 Communication between Users and Developers

Good communication between users and developers is essential for the success of a project.

- Before software design begins, it is essential that the needs and preferences of the users be communicated to the developers.
- After the development is complete, the user-visible properties of the product must be communicated to the users so that they can make effective use of the software.

Volker Gruhn [6] has noted that the single best determiner of a project's success is good communication between developers and users. The programmers must understand the users and the users must understand the product.

3.2 Multi-developer Projects

Studies of Global or Distributed System Development [2] make it clear that communication between the developers is also critical to success. When several developers cooperate to produce a software product, it is important that the responsibilities of each be clearly specified. Otherwise, there will be things that are done more than once while other things are neglected. Conventionally, a product is divided into programming assignments, which I call modules, and each work assignment is the responsibility of a group of developers or an individual developer. Each developer must know both what he or she is required to do and what can be expected of the others. All developers require a clear description of the characteristics of the support system (hardware and software).

4 Multi-version Software Products

Successful software products evolve but the older versions don't always disappear. Sometimes, an evolution step produces a complete improvement with the new product replacing the old. Even in such cases, some customers will decide to keep using the old version. Sometimes an evolution step produces a new version intended to run on a new platform, serve a new class of users, or provide a different service. The older versions remain in the product line. A prototypical example is the Adobe Acrobat line of products to read, produce, and modify .pdf files [1]. There are free, standard, and "professional" versions. The current version is "10" but older versions are still in widespread use. Versions exist for PC, UNIX, and Apple platforms. Versions also exist for different regional languages.

There are many approaches to producing multi-version software products (software product lines).

- New products are developed "from scratch".
- New products are developed from scratch but may reuse some old code. Some of the reused code may be modified.
- New versions are produced by modifying an existing version.
- Abstract programs may be developed for a family, then refined for a particular family member [17].

- Products can consist of modules, some of which have several interchangeable implementations [14, 15, 19].
- Some versions may be a subset or extension of others [18, 16].

A product line may be constructed by a combination of these techniques. Below, the reader will find a summary of the advantages and disadvantages of each method, followed by a more detailed discussion of what is required to use each of the techniques.

4.1 Developing Products Independently

If an organization is to develop a product line as a set of separate projects, the failure or delay of one project will not prevent other products from being brought to market. However, the result may be a "line" that seems to have little or nothing in common. As time passes, a provider may find that they are burdened by having to maintain so many separately developed programs. This burden is not usually shared by new competitors in the market; this gives newcomers a cost advantage.

4.2 Reuse

Software reuse is like motherhood, everyone is in favour of it, but it seems to be very hard to do. Unless the software was carefully designed and documented with future applications in mind, it may be harder to modify and reuse a program than to simply rewrite it. The reused code may be less than ideal for its new application. It can be difficult to find reusable code and, when it is found, the programs are often poorly documented and hard to understand.

4.3 Sequential Modification

If each new version is produced by modifying an older version, little time need be "wasted" on planning for change and each version is a concrete, executable, program. Time-to-market for the first few versions is usually reasonable, but after many revisions, the time to make revisions often increases. It has been observed that complexity grows with each modification and the cost of maintenance increases as the software ages. [22]

Creating a new product in a software product line by modifying an earlier product, especially a product that was not designed for ease of modification, requires developers to have both a broad overview and detailed knowledge of the older product. Because it is natural to try to minimize the changes, the result is often a program that contains artifacts from the older products that are not ideal for the new product but would be difficult to modify or remove. For example, some data structures may retain data elements that are not needed in the new version. It is common to find redundant information and information being handled in inconsistent ways. If this process is repeated many times, the product loses what Brooks has called conceptual integrity[2] [4]. This process has been observed to result in high "maintenance" costs because,

[2] I leave it to philosophers to decide whether a product that was intended to sell, but fails to attract any users should be considered software.

when a change is needed in all the products, each version must be changed separately. Maintainers have observed that it is more difficult to maintain a set of almost-alike programs than to maintain a set of completely different ones. [22]

4.4 Multi-version Interchangeable Components

With good modularity [14, 15 19], sharing is increased and maintenance costs can be reduced. Time must be spent in designing and documenting both the module structure and the module interfaces in order to keep the interfaces reusable across the product line.

If two components are to be interchangeable, they must offer the same interface to all the programs that use them. This requires detailed, communication between the developers that will implement the various versions. It also requires precise communication between the teams that will implement the components and the teams that will write programs that use those components. [13, 30]

4.5 Contractable and Extensible Systems

If a program has been designed for extension and contraction [18], sharing across versions is improved. Upward compatibility of applications is also improved. However, time must be spent in designing (and documenting) module structure and interfaces in order to make this kind of change possible.

Making systems extensible and contractable, requires restricting the "uses" relation between programs to be loop free and (consequently) define a hierarchy. [16] This often requires splitting functions that might have been implemented in one program so that they will be performed by several distinct programs. This allows the individual programs to be on different levels in the "uses" hierarchy [18]. This type of design requires careful communication about the interfaces and which programs are permitted to use each program. If programmers do not adhere to specified restrictions on the "uses" relation, it is possible to end up with a system in which nothing works until everything works. That will also make creating some subsets and extensions more difficult because many programs will have to be revised. Failure to communicate about these issues often leads to the development of duplicate or almost-alike services.

4.6 Program Family Approach

The program family approach requires that significant time be invested in analyzing the problem to make predictions about possible changes; that "up front" investment can make maintenance of a large product line much easier.

Early work on program families [17] proposed that designers develop a family tree. The root of the tree, an abstract program, represents all the programs in the family. The "child" of a node represents a subfamily of the family represented by a node. Only the leaves on the tree can be concrete (complete and executable) programs. No concrete program should have descendants.

More recent work has revealed that the tree model is too restrictive. It forces an order on decisions that are not dependant on each other. However, the principle of identifying the commonalities before describing variable aspects is valid and important.

5 Communicating Design Information about Software

Almost all software development organizations recognize the importance of good communication between developers and users and between developers. Many managers believe that communication can be improved by "people methods" such as, getting people into the same room (or have them attend the same net meetings), "stand up" meetings, social occasions, etc. Others suggest that communication is improved by having two people work on each module (pair programming). Unfortunately, the problem is not that simple! No matter how often people meet, and how friendly and well-intentioned they may be, their discussion will not cover all the essential details. Further, it is likely that the ambiguity and lack of structure in natural language communication will lead people to believe that they have reached agreement when they actually have significant differences in opinion or understanding. Finally, if the agreement is not written down (documented) details are likely to be forgotten or remembered differently by some of the parties.

The best way to get people to communicate about the many detailed decisions that must be understood is to have them cooperate in the production (and reviewing) of highly structured design documents. If the documents are structured in such a way that they can be checked for completeness, the writers and reviewers are less likely to skip any topics. If the language used in the documents is mathematics rather than natural language, the possibility of undetected disagreements is greatly reduced. It is essential to train all developers to write and read mathematical documents. [29]

6 Software Engineering Tasks Other Than Programming

At the heart of the semantic question, "Is Software Engineering different from programming?", lies a practical question about the education of people who want to become Software Engineers, viz., "What do they need to learn?". [26] There should be no doubt that Software Engineers must be good programmers and that they must understand programming languages as well as other tools of the trade. It doesn't seem to matter which tools they use during their basic education; during their career they will have to learn many others. What does matter is that they understand the basic concepts of programming well enough that they will be able to learn new languages, evaluate languages, choose languages to use, and to do the same with other tools. However, beyond programming and these related issues, there are many things that Software Engineers must know how to do. These can all be attributed to the multi-person, multi-version nature of the activity.

6.1 Requirements Elicitation and Documentation

The most basic obligation of Professional Engineers is to ensure that their products are fit for use. To do that, the Engineer must know what is required of the product. The requirements are not limited to the wishes expressed by the customer; in addition to customer requests, the requirements include:

- functions of which the customer might not (yet) be aware,
- the properties of the environment in which the product must be able to function, and
- requirements implied by the obligation of licensed Engineers to protect the public.

Determining the requirements frequently requires the participation of professionals other than Engineers. In some cases, the eventual developers might not be involved in determining what the requirements are. To fulfill their obligations, (and reduce their liability) Engineers must insist on having a document that clearly states the requirements and has been approved by all relevant parties. Such a document might be compiled by user representatives, the developers, or a third party. Before starting to develop the product, the developing Engineers must confirm that this document is complete, consistent, and unambiguous. [7, 28]

It is often stated that Engineers must deal with conflicting requirements. This is not correct; there may be conflicting wishes but these must be resolved so that the binding document is unambiguous and consistent.

Very often, there will be errors in the original document. Further, during the development period, understanding may deepen and the situation may change with the result that the requirements change. The Engineer should insist that any requirements changes are reflected in a revised requirements document that has been reviewed and approved by all concerned parties.

The fact that a requirements document may be incorrect, or soon become outdated, is often used to argue that spending time on requirements documentation is a waste of time. In fact, it is a reason to do a better job of determining requirements and to prepare both the software and the document so that the most likely changes will be localized. A requirements document can characterize the set of most likely changes. [28].

A realistic requirements document must reflect an understanding of the technologies that are available. This determines both what can be built and the information that must be obtained and documented before the design is complete. If those who will do the development are not involved in writing the document, they must review the document to confirm that it is realistic before accepting an assignment to build a system that satisfies the stated requirements[3].

A good requirements document must:

- leave no user visible decisions to the programmer,
- contain a complete description of possible environmental conditions, and
- be an unambiguous description of acceptable behaviour.

6.2 Module Structure Design and Documentation

It is now widely acknowledged that any software too large to be written in a day by a single programmer should be “modular”. By this is meant that the program should comprise several modules, each of which can be written, tested, changed or replaced without precipitating changes in the others. It was shown in the early 70s that the task is very different from programming. Programming is concerned with

[3] I have known Engineers who accepted a requirements document that they knew to be unrealistic because they would profit financially. This was a violation of their professional obligations.

determining the sequence of events needed to carry out a data transformation. If one "uses" this sequencing plan to divide a program into modules, one gets a very poor design [14, 15 19].

6.3 Module Interface Design

The success of an effort to develop a modular system depends on the design of the interfaces between the modules. To design interfaces properly, one must begin with a careful (hence nonstandard) definition of the term.

> Given two communicating software components, ***A*** and ***B***, ***B***'s *interface* to ***A*** is the weakest assumption about ***B*** that would allow you to prove that ***A*** is correct.

This definition recognizes that component A may have a different interface to each program that interacts with it and that interfaces are not symmetric.

If the assumptions are too weak, ***A*** will not be able to use ***B*** efficiently. If the assumptions are unnecessarily strong, ***B*** will be dependent on unimportant (and changeable) facts about ***A***. Designing interfaces is an essential part of Software Engineering that is not programming. It is a very different task from designing algorithms and data structures [3].

6.4 Designing for Extension and Contraction

The "uses hierarchy" first defined in [16] must be designed as discussed and illustrated in [18].

6.5 Module Interface Documentation

Because we are discussing multi-person development projects, it is essential that the interfaces between modules be documented. Without module interface documentation that is precise, accurate, and complete:

- The interface will not get the reviews required to assure high quality design.
- Those who write programs that use the module will not have the information that they need to use it properly.
- Those who maintain the software will not have the information that they need to revise the module in accordance with its specified interfaces.

If a module is used by many other modules, we usually document an interface that implies all of the assumptions that any of them are allowed to make. Some of of the using programs will not exploit the full documented interface. They have a weaker interface.

In rare cases we may also document restricted interfaces, i.e. assumptions that we do not want all using programs to exploit but are required for specific tasks such as hardware maintenance.

Documentation of module interfaces is very different from implementing them in a procedural language. It is a skill that most programmers do no possess and most CS programmes do not teach [27].

6.6 Program Function Documentation

Although the need to document programs has long been recognized, there is usually no clear distinction between internal and external documentation. The external documentation is intended for those who will use the program and describes the externally visible effects. The internal documentation is intended for those who will review or revise the program and describes the programs structure and the role of each of the structures and components within the program.

Following Mills [10], the external documentation is called program function documentation. The term, "function" is used in both its informal and its mathematical meaning. Informally, this documentation describes the effect of executing a program on externally accessible data elements, i.e the function of the program. Mathematically, the function documentation for a terminating program is a mapping from starting state to stopping state (s). When non-deterministic behaviour is possible, one must use a relational model such as the one in [20, 24]. The function of non-terminating programs can be described by a mapping from event histories to output stream values [9].

6.7 Module Internal Design Documentation

In a successful software development project, module designs will need to be reviewed by people other than the designer and the module's code will almost certainly be maintained by people who were not involved in the original programming. For these reasons, the "design" (the data structure, the abstraction function (mapping from data structure values to external views of the module state) and the program functions of the main programs) will need to be documented. This is not at all like writing the procedural programs needed to implement that design.

6.8 Quality Assurance

Today's software products are often immense and complex. Some have the chaotic property that a small change to one code-segment can be inconsistent with another piece of code elsewhere in the system and cause hard-to-find bugs. Effective inspection of such programs is difficult and testing will require much skill and knowledge. These activities require quite a different skill set from programming [23, 21].

7 A Profession Called "Software Engineering"?

The organizers of the first Software Engineering conferences wanted to establish Software Development as a profession like Mechanical Engineering or Civil Engineering. We should be asking whether they have succeeded or not. It is now more than four decades since the introduction of the term "Software Engineering" and Brian Randell's brilliant characterization of its essential properties. How far have we come?

7.1 Requirements Documentation Is Imprecise, Incomplete, Often Useless, and Sometimes Harmful

In spite of years of haranguing about the importance of "Requirements Engineering", a look at today's practice reveals that practitioners are not writing precise requirements documents. Most are in the "wish list" format, a format that does not tell the programmer's what the behaviour of their product should be. These documents do not support those who must test or inspect the product. They often sit on shelves collecting dust. [28]

7.2 Module Structure Is Still Badly Designed

Although that the principle that each module should hide (abstract from) a changeable aspect of the design has been well-known and widely accepted for four decades, examination of software structures reveals that the principle is still not consistently applied. The idea of "object orientation" another view of the same principle has led to a class of languages that supports this kind of design but, in many systems, little thought appears to have been given to exactly what should be hidden. Many objects hide nothing.

7.3 Module Interfaces Are Ad Hoc

Providing module interfaces in the form of a set of externally accessible programs, first proposed in the 70s [12, 13, 14, 79] is now used in many applications such as method libraries and APIs. However, the inclusion of somewhat arbitrary data structures (including XML structures) as part of the interface is still with us and still causing problems.

The interfaces usually appear to be the result of an intuitive process in which programs to satisfy obvious needs are provided initially and ad hoc changes are made later. There is no sign of the use of explicit, conscious, design criteria. There is no sign of conscious attention to separation of concerns or information hiding.

7.4 Documentation Is Not Viewed as a Serious Task

In other disciplines, documentation is viewed as a design medium, with each phase of the design recording its decisions in a set of well-organized, reviewable, reference documents. In software development, a document is something written after the product is finished. Descriptive documents are usually a stream-of consciousness exercise in writing from memory. There may be a specified outline but the headings are vague and often ignored.

Often the documentation is considered too unimportant for the valuable programmers (who are needed for further development) and left to technical writers who are not themselves designers and not very familiar with the design.

The quality of the documentation is often so bad that maintainers will not even look at it. They prefer to study the code even though doing so is difficult and errors are common.

7.5 Low Expectations of the Product

Developers and customers alike have learned to expect errors in early releases of software and to a long sequence of updates and new versions. When an error is pointed out to developers, the response is often something like, "Yeah, that's software; you can never get it right." Some studies have revealed that when reliability does improve, it is because users have learned to avoid the problems, not because the problems have been corrected.

7.6 Low Entrance Standards for the Profession

One of the signs that a profession has matured is the establishment of clear requirements for professionals. Professions such as Medicine, Engineering, Law, and Haircutting, require that practitioners have completed approved training programs and/or pass a test. One can get jobs with the title, "Software Engineer" without any such credentials. Even those who have academic credentials, may not have full qualifications because there is no recognized core body of knowledge that must be taught in Computer Science or Software Engineering programmes.

8 A Challenge for Lawmakers

We have legislation that is intended to make aircraft, cars, elevators, highways, bridges, buildings, medicine, medical devices, and many other products safe for use. Through this legislation we have codes that products must meet. However, the legislation has not kept up with the times. Today, we depend heavily on software; either software is in the things we use or the designers used software to develop it. In spite of this, software products have a disclaimer where we should see a warranty. There is no effective licensing process for the people who produce, test, and maintain the software. There is no agency that certifies that software is safe for use. It is time that our laws caught up with our technology.

9 A Challenge for Educators

Although the people who introduced the term "Software Engineering" wanted to see software development develop into a profession like the traditional Engineering disciplines, they have not looked seriously at the differences between Engineering education and education in Science and Mathematics. The concept of Professional Education is critical to developing a mature profession. [25] It is based on three ideas:

- There is a "core body of knowledge" required of all who practice the profession. The "core body" must be fundamental (lasting) knowledge, not merely current technology.
- Science and Mathematics are taught with the stress on how to use them.
- Students are taught how to use their knowledge of fundamental science to learn, understand, and use the latest technologies.

There are few programmes that prepare software professionals in this way. We have not yet agreed on the core body of knowledge, and many instructors try to provide broad surveys rather than teaching how to do the job properly.

10 Conclusions

Brian Randell clearly identified the key difference between Computer Science, a research field, and Software Engineering, a Profession. Unfortunately, in the (more than) four decades that have passed since the term "Software Engineering" was coined, we have seen little progress in the professionalism and discipline of software developers. Incredible hardware advances and the work of many clever (but not necessarily disciplined) programmers have created an impression that software has changed. Reading code often reveals that the differences are shallow. Using the code, you often discover that it is easy to fool it or cause it to get into an undesirable state. The knowledge about how to design software that has been gained in that period, is not being taught or used.

References

1. http://www.adobe.com/products/acrobat.html
2. Ågerfalk, P.J., Fitzgerald, B.: Special Issue: Flexible and Distributed Software Processes: Old Petunias in New Bowls? Comm. ACM 49(10), 26–34 (2006)
3. Britton, K.H., Parker, R.A., Parnas, D.L.: A Procedure for Designing Abstract Interfaces for Device Interface Modules. In: Proceedings of the 5th International Conference on Software Engineering, pp. 195–204 (March 1981); reprinted as Chapter 15 in [8]
4. Brooks Jr., F.P.: The Mythical Man-Month: Essays on Software Engineering, 2nd edn. Addison Wesley, Reading (1995)
5. Buxton, J.N., Randell, B. (eds.): Software Engineering Techniques: Report on a Conference sponsored by the NATO Science Committee, Rome, Italy, October 27-31, 1969, Brussels, Scientific Affairs Division, NATO, p. 164 (April 1970)
6. Gruhn, V., Schäfer, C.: Systemic Approaches for Software Engineering. In: Fujita, H. (ed.) Frontiers in Artificial Intelligence and Applications. New Trends in Software Methodologies, Tools and Techniques, vol. 217, pp. 85–95. IOS Press, B. V., Amsterdam (2010) ISBN 978-1-60750-629-4-85 (print)
7. Heninger, K.L.: Specifying Software Requirements for Complex Systems: New Techniques and their Application. IEEE Transactions Software Engineering SE-6, 2–13 (1980); reprinted as chapter 6 in 8
8. Hoffman, D.M., Weiss, D.M. (eds.): Software Fundamentals: Collected Papers by David L. Parnas, 664 pgs. Addison-Wesley, Reading (2001) ISBN 0-201-70369-6
9. Liu, Z.Y., Parnas, D.L., Trancón y Widemann, B.: Documenting and Verifying Systems Assembled from Components. Front. Comput. Sci. China 4(2), 151–161 (2010) ISSN1673-7350 (Print) 1673-7466 (Online), doi:10.1007/s11704-010- 0026-2
10. Mills, H.D.: The New Math of Computer Programming. Comm. ACM 18(1), 43–48 (1975)
11. Naur, P., Randell, B. (eds.): Software Engineering: Report on a Conference sponsored by the NATO Science Committee, Garmisch, Germany, October 7-11, 1968, Brussels, Scientific Affairs Division, NATO, p. 231 (January 1969)

12. Parnas, D.L.: Information Distributions Aspects of Design Methodology. In: Proceedings of IFIP Congress 1971, Booklet TA-3, pp. 26–30 (1971)
13. Parnas, D.L.: A Technique for Software Module Specification with Examples. Communications of the ACM 15(5), 330-336 (1972); republished in Yourdon, E.N. (ed.): Writings of the Revolution, pp. 5–18. Yourdon Press, New York (1982); also in Gehani, N., McGettrick, A.D. (eds.): Software Specification Techniques, pp. 75–88. AT&T Bell Telephone Laboratories, Addison Welsey (1985) (QA 76.7 S6437)
14. Parnas, D.L.: On the Criteria to be Used in Decomposing Systems into Modules. Comm. ACM 15(12), 1053–1058 (1972); republished in Yourdon, E.N. (ed.): Classics in Software Engineering, pp. 141–150. Yourdon Press (1979); republished in Laplante, P. (ed.): Great Papers in Computer Science, pp. 433–441. West Publishing Co, Minneapolis/St. Paul (1996), reprinted as chapter 7 in 8, reprinted in Broy, M., Denert, E. (eds.): Software Pioneers: Contributions to Software Engineering, pp. 481–498. Springer, Heidelberg (2002) ISBN 3-540-43081-4
15. Parnas, D.L.: Some Conclusions from an Experiment in Software Engineering Techniques. In: Proceedings of the 1972 FJCC, vol. 41, part I, pp. 325–330 (1972)
16. Parnas, D.L.: On a 'Buzzword': Hierarchical Structure. In: IFIP Congress 1974, pp. 336–339. North Holland Publishing Company, Amsterdam (1974); reprinted as chapter 8 in [8], reprinted in Broy, M., Denert, E. (eds.) Software Pioneers: Contributions to Software Engineering, pp. 501–513. Springer, Heidelberg (2002) ISBN 3-540-43081-4
17. Parnas, D.L.: On the Design and Development of Program Families. IEEE Transactions on Software Engineering SE-2(1), 1–9 (1976); reprinted as chapter 10 in [8]
18. Parnas, D.L.: Designing Software for Ease of Extension and Contraction. IEEE Transactions on Software Engineering, 128–138 (1979); also in Proceedings of the Third International Conference on Software Engineering, pp. 264–277 (May 1978); reprinted as chapter 14 in [8]
19. Parnas, D.L., Clements, P.C., Weiss, D.M.: The Modular Structure of Complex Systems. IEEE Transactions on Software Engineering SE-11(3), 259–266 (1985); also published in Proceedings of 7th International Conference on Software Engineering, pp. 408–417 (March 1984); reprinted in IEEE Tutorial: Object-Oriented Computing, vol. 2, Implementations edited by Peterson, G.E., pp. 162–169. IEEE Computer Society Press, Los Alamitos (1987), reprinted as chapter 16 in [8]
20. Parnas, D.L., Wadge, W.: Less Restrictive Constructs For Structured Programs, Technical Report No. 86-186, Queen's, Kingston, Ontario, 16 pgs (September 1986); reprinted as chapter 2 in [8]
21. Parnas, D.L., Asmis, G.J.K., Madey, J.: Assessment of Safety-Critical Software in Nuclear Power Plants. Nuclear Safety 32(2), 189–198 (1991)
22. Parnas, D.L.: Software Aging. In: Proceedings of the 16th International Conference on Software Engineering (ICSE), Sorento Italy, pp. 279–287. IEEE Press, Los Alamitos (1994); reprinted as chapter 29 in [8]
23. Parnas, D.L.: Inspection of Safety Critical Software using Function Tables. In: Proceedings of IFIP World Congress 1994, vol. III, pp. 270–277 (August 1994); reprinted as chapter 19 in [8]
24. Parnas, D.L., Madey, J., Iglewski, M.: Precise Documentation of Well-Structured Programs. IEEE Transactions on Software Engineering 20(12), 948–976 (1994)
25. Parnas, D.L.: Software Engineering Programmes are not Computer Science Programmes. Annals of Software Engineering 6, 19–37 (1998); reprinted (by request) in IEEE Software, 19–30 (November/December 1999)

26. Parnas, D.L.: Structured programming: A minor part of software engineering. Madey, J., Fiadeiro, J.L., Tarlecki, A. (guest eds.) Information Processing Letters 88(1-2), 53–58 (2003)
27. Parnas, D.L.: Component Interface Documentation - What do we Need and Why do we Need it. In: Fujita, H., Mejri, M. (eds.) New Trends In Software Methodologies, Tools And Techniques, pp. 3–21. ISO Press (2006) ISBN 978-1-58603-673-7
28. Parnas, D.L.: From Requirements to Architecture. In: Fujita, H. (ed.) Frontiers in Artificial Intelligence and Applications. New Trends in Software Methodologies, Tools and Techniques, vol. 217, pp. 3–36. IOS Press, B.V., Amsterdam (2010) ISBN 978-1-60750-628-7 (print), ISBN 978-1-60750-629-4 (online)
29. Parnas, D.L.: Precise Documentation: The Key to Better Software. In: Nanz, S. (ed.): The Future of Software Engineering, pp. 125–148. Springer, Heidelberg, doi:10.1007/978-3-642-15187-3_8, ISBN 978-3-642-15186-6 (Print) 978-3-642-15187-3 (Online)
30. Quinn, C., Vilkomir, S.A., Parnas, D.L., Kostic, S.: Specification of Software Component Requirements Using the Trace Function Method. In: Proceeding of the International Conference on Software Engineering Advances (ICSEA 2006), Tahiti, French Polynesia (October 29-November 1, 2006)