A Brief Top-Down and Bottom-Up Philosophy on Software Evolution

Markus Pizka[*], Andreas Bauer
Institut für Informatik, Technische Universität München
Boltzmannstr. 3, D-85748 Garching b. München, Germany
{pizka|baueran}@in.tum.de

Abstract

The decision on whether to proceed top-down or bottom-up during software development has a strong and underestimated impact on the quality of the final product including its later evolvability. Various examples for both strategies taken from such different domains as operating systems and computer games provide evidence that bottom-up developed systems are more suitable for future evolution. The reasons for this range from the increased compositionality of bottom-up developed artefacts at the technical level up to a greater independence from certain requirements which constitute the most transient part of a software system. Besides those advantages concerning evolvability, the negative effects of bottom-up orientation can not be ignored. Furthermore, proceeding bottom-up contradicts most conventional development processes. We regard this as a clear indication for the need of new development processes to improve the construction of evolvable software.

1. Processes Influence Evolvability

Studies of software project success and failure, like Standish Group's "Chaos" reports [23] state that project management aspects, such as executive management support, proper planning, and a well-defined process do have a much stronger impact on the quality of the final product than any particular technique, such as preferring Java over C as the implementation language or structuring by means of PL/1 modules versus more fashionable EJB components.

While the correlation of the development process with software quality in general is well-known and extensively studied in the software process improvement field, see e. g. Cleanroom, GQM [2], and CMM [17], there exists only limited knowledge about the impact of the development process on the evolvability of the software product. Although we do not believe that the influence of technical aspects on software evolution is similarly negligible, we do claim that the development process indeed has a strong and largely underestimated influence on the evolvability of the software systems it produces.

[*]Part of this work was sponsored by the German Federal Ministry for Education and Research (BMBF) as part of the project VSEK (www.software-kompetenz.de).

The emphasis is on evolvability rather than on evolution. While different process model like the sequential waterfall versus more evolutionary models [4] clearly differ in their ability to support evolution, we will not elaborate on performing evolution, but explain that the process employed during development of the initial release has a lasting impact on the subsequent evolvability of the resulting system.

The decision on whether to proceed top-down (TD) or bottom-up (BU) seems to predetermine the expectancy of life of a software system. Examples taken from such diverse fields as operating systems and computer games point out that BU constructed systems are much better prepared for evolution than TD created ones. BU orientation seems to deliver artefacts of increased compositionality and with greater independence from the ever changing requirements. This gives rise to the consideration that BU orientation might be a better principle for the construction of e-type systems [13] than the conventional TD refinement. Of course, this supposition is provocative as it puts the established TD oriented way of thinking in question and therefore needs to be substantiated. This paper discusses the differences between BU and TD development concerning software evolvability and provides a variety of examples illustrating the effects of the two strategies on software evolution.

2. Other Work on Processes and Evolution

There is no other work directly comparing the BU versus TD development with respect to software evolvability — at least not known to the authors. Clearly, there is a lot of work on how software evolves and how to pro-actively perform and steer the process of software evolution. Some of these processes, such as Change Management and Configuration Management are already broadly accepted and used in current practice. More innovative processes also take program comprehension [26], assessment [24], or software reuse [7] into account. All of these processes provide valuable guidelines on how to perform a certain maintenance task or (parts of) an evolution step occurring in a system. In contrast to this, this paper does not focus on the "how to" side, but on the principle "impact on" evolution which results from and is influenced by following either the BU or the TD strategy.

Lehman's work has brought valuable insights on the principles of software evolution [12, 13] describing it as a multi-dimensional, multi-level, and multi-agent feedback process.

Lehman's extensive studies show that the process of evolution, generally, is independent from certain technical aspects, such as the choice for a specific programming language, but instead driven from an ever changing environment and assumptions made during system modelling. We broadly agree on this view and have provided further evidence from Open Source software in [3]. Nevertheless, different processes and techniques do have an impact on the ability, speed, and costs of software evolution. And answers to the question about how to increase the evolvability of software systems are of high practical relevancy.

Raijlich's staged software life cycle [19] also details Lehman's considerations on how a system evolves by distinguishing certain stages of evolution from its initiation to phase-out. Again, this model describes the stages a software system runs through, but not the factors increasing or decreasing the length of the various stages which is subject to this paper.

And of course, there is a proliferation of work on techniques for evolution such as on architecture [14], metrics [9], and slicing [25]. While some of these works on techniques and tools have already gone quite far the consideration of processes seems to lag behind.

3. Bottom-Up and Top-Down Development

Though BU and TD are commonly used terms, their interpretation in different contexts varies. We will therefore clarify our notion of BU and TD.

3.1. Top-Down and Bottom-Up Principles

As depicted in Fig. 1 software establishes a correspondence between requirements and a given technical reality. The requirements are at the "top" and the technical basis at the "bottom". Both sides are not fixed but more or less deliberately chosen. Since we are usually not willing to go down to elementary physics and up to the meaning of life, a software project usually defines or selects a certain technical basis as a starting point to implement hopefully realistic business requirements. While the technical basis of an embedded system, for instance, is close to control unit hardware and protocols the basis of an information system will rather be an API of a database system and a programming language. Similarly, requirements can either be demands from users, or the properties of an abstract concept, such as a state automaton. According to this, the distance between top and bottom can be as short as a single atomic transformation step, or span numerous intermediate levels.

Software is the extensive and complicated cluster of decisions, concepts, documents, and algorithms providing a mapping between the predetermined requirements and the available technique. This mapping can be viewed from either side: software maps the requirements onto the technical basis, but also the technical basis onto the requirements.

Although both directions are valid, they imply a *very different* way of thinking! The mapping usually does not occur at once but is constructed over hundreds of intermediate steps over a long period of time. This construction may start with the requirements and refine them stepwise down to an implementation in which case we call it **top-down development**. **Bottom-up development**, in contrast, originates from the technical basis and tries to reach up to the requirements by building higher level services and components on what is already implemented. Though the outcome is always a correspondence between top and bottom the established path will differ significantly according to the direction taken, because the individual transition steps are guided by different perspectives! Most operating systems (OS) are an example for BU construction. Hard disks with sectors and heads have been abstracted to blocks and further on to universal byte structured files as well as directory trees. TD constructed OS ended up in very different storage concepts, such as a persistent object store [5], because they are usually guided by user-level concepts. In contrast to this, information systems, such as for banking, are mostly constructed TD, because fulfilling requirements and ease of use are of paramount importance. An exaggeratedly BU driven banking system would consist of various tiny applications that the user has to combine in the right order to achieve the desired effect — just like searching for a document with Unix shell commands `grep`, `find`, `xargs`, and " | ".

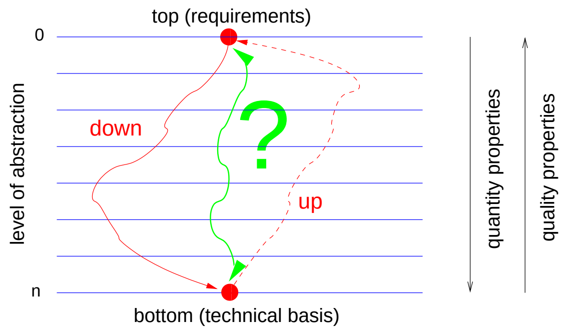


Figure 1. Going up or down?

3.2. Pros and Cons

Trivially, a TD constructed system is more likely to match its requirements, because this is what the construction starts with. However, TD development may well miss technical reality. Even if the initial set of requirements is realistic, intermediate decisions may cause an aberration from the optimal mapping as shown in Fig. 1: once the divergence between the constructed path and the available infrastructure becomes evident, it either has to be corrected with complicated and inefficient workarounds at the lower technical levels, or some kind of expensive backtracking must be applied to revoke previous decisions.

Another well-known problem of TD development is its "big-bang nature", i. e. there is no running system until the end of development. This entails high risks due to late validation by the customer and a long time to system. Albeit, the vast majority of software projects adheres to the TD principle, simply because most common process models determine a strict TD order of events.

BU development overcomes these problems. It quickly provides a rudimentary running system, proceeds rapidly and delivers highly efficient systems that — at the end — might fail to match the requirements. Again, this deviation can be compensated with workarounds or backtracking. But, BU has an important additional option. Unlike the technical basis, requirements can be modified to some extent. In practice, BU constructed systems are usually not significantly changed if the requirements are not perfectly met but the user is forced to adapt to the system. While this may sound irritating, it is common practice in numerous domains, such as system level software but also for enterprise resource planning systems like SAP R/3. In a recent IT manager workshop[1] all attendees agreed that they rather adapted the organisation than change their standard software.

3.3. Philosophical Implications

One can easily further generalise these considerations by abstracting from software. As a matter of fact this helps to understand the role of TD and BU thinking on software evolution.

Top represents long-term wishes and dreams. The bottom side is constituted by actually existing options. Someone whose pattern of thought is strongly bound to the top side usually fails to deliver a result which is the typical trouble of the perfectionist. In contrast, the tinkerer, who is strongly focusing on the bottom produces a lot of useless things.

The key to long term success is to keep both perspectives in balance. It is no coincidence that successful, large-scale organisations perform both strategic and operational management with similar priority. Strategic questions correspond to long term wishes (top) and operational issues represent short term options (bottom). By integrating both sides, these organisations are able to quickly produce useful results (products) by continuously adapting to the changing demands on a solid operational foundation.

This long term success, i. e. preserving value by continuous adaptation, is exactly what software evolution is aiming for, too. We claim, that the principle to balance BU and TD orientation, with an emphasise on BU, could be just as beneficial for software systems as it is for organisations. Currently, most development processes are TD centric and hamper evolvability.

4. Impact on Software Evolvability

Besides the differences of TD versus BU on software quality in general, the decision on whether to proceed upwards or downwards affects two properties of the resulting product that are crucial for successful long-term evolution: first is the compositionality of the developed structure and its elements, second is the stability of the concepts derived. We argue that consequent BU orientation helps to produce components of reduced size and increased *orthogonality* resulting in higher compositionality. Furthermore, a bottom-up developed architecture is less coined by the requirements of users which are by far the most volatile part of a software system. Trivially, systems with decreased exposure to variance need to be changed less frequently which slows the process of decay [6]. Increased compositionality, additionally, allows for less intrusive, i. e. destructive, and therefore more flexible changes. Both aspects contribute to increased evolvability.

[1]German CIO quarterly meeting, Munich, Germany, March 2004

4.1. Compositionality

BU software engineering contributes to the separation of concerns [16]. Unlike TD approaches which mostly try to satisfy user requirements, BU approaches force the developer to think in terms of *orthogonal* and *scalable* components and subcomponents, simply because the later requirements are a priori unknown and the constructed system must be flexible enough to satisfy various requirements.

New functionality in BU designed systems is merely a recombination of already existing subsystems or components. In other words, BU engineering tends to alter the requirements to match the system, rather than modifying the system to reflect new demands.

The concerns which are separated at the technical level are those that enable orthogonal scalability of features and functionality, while TD usually separates requirements, use cases, and scenarios. Evolvability, of course, is mostly influenced by the former, giving reason as to why at least BU aspects in system design are crucial.

4.2. Increased Stability

As discussed in sec. 3.3, requirements represent wishes and dreams which are usually highly volatile. In fact, no other part of a software system changes more frequently than the business processes underlying the requirements [10]. Hardware and other low-level technical details are a lot more “stable” than requirements. Studies on software maintenance show that about 55% of all change requests are new or changed requirements (perfective) while only 25% are technical changes (adaptive) [18].

To achieve longevity of a system it is in general useful to operate as far away from volatility as possible, because frequent changes entail decay [6] reducing the expectancy of life of the system. The decisions made during BU construction are indeed hardly influenced by changing requirements but based on the more stable technical platform.

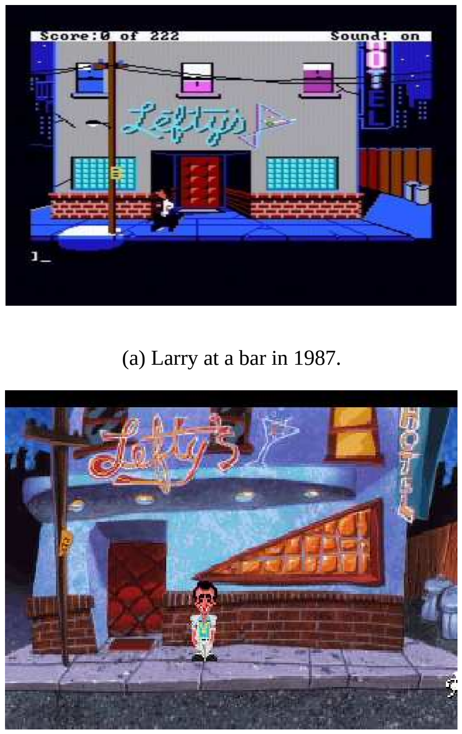
Contrary, systems leaning towards the more variable requirements side, as it happens during TD construction, encounter greater difficulties to keep up with the constant change at the top side. Changes of the requirements often affect extensive parts of the system because initial decisions — the most important ones — are already steered by volatile top-level properties, such as business functions of different departments of an organisation.

5. Examples of BU and TD Construction

In the following, we present different examples of BU versus TD construction ranging from adventure games (see sec. 5.1) to a large-scale weather satellite imaging and analysis processor (see sec. 5.3).

5.1. Adventure Games

Interactive adventures are complex software systems. Between 1975–1980, these games were text-based written in languages like FORTRAN, or LISP running on large mainframes and controlled via terminals usually residing in a campus' students lab. Later ports of the games often came down to being only a few kBytes in size.



(a) Larry at a bar in 1987.

(b) Larry at the same bar, four years later.

Figure 2. Leisure Suit Larry now and then.

While many of the companies who created and sold the early text-based adventures went bankrupt, their products managed to evolve and almost shape an academic field (see [15]). Interestingly, these systems emerged BU with an *invariant core* that is independent from assumptions regarding user acceptance or operability. The community furtheron established components that facilitate creating similar applications by emphasising portability and compatibility (bottom side). Meeting the top requirements asserted by market pressure was largely disregarded and led to commercial failure but also to high-quality software that is still studied by many, today (e. g. Inform [1], Z-virtual-machine, etc.).

Nowadays, games of that genre tend to fill up half a hard disk's capacity with multimedia sound effects and movies that are necessary in order to address the customers demands on a highly competitive and global market.

Screenshots of the considerably successful game "Leisure Suit Larry" are depicted in Fig. 2: firstly as it was originally released in 1987 and secondly, a new version with updated graphics and a comfortable mouse-driven user interface from 1991. The costly re-issue of the same game within only four years after its initial release became a necessity because the overall concept of "Larry" was tightly bound to the top-level user interface, which was changing rapidly.

5.2. Systems Programming

Systems programming is an interesting example in a sense that there exist numerous successful architectures developed both in a BU as well as TD fashion. However, while BU developed systems like the nowadays MacOS X, *BSD, Linux, or Solaris all thrive for continuing success, most of their TD counterparts are either in the process of being phased out, or have already ceased to exist.

MacOS X, formerly NextStep, was Apple's change towards a new platform consisting of new generation Motorola PowerPC CPUs and a Unix system kernel. It is interesting to notice, however, that an earlier and hugely expensive TD attempt of Apple Computer to design an OS on their own, failed and drove the company almost towards irrelevance during that period. The requirements for the system were to be absolutely compatible with their previous non-preemptive OS while, at the same time, offering users an entirely new system with better multitasking and multimedia capabilities. The project was called "Rhapsody" and never got to a stage where it would even come close to the requirements asserted TD from managers trying to gain market share.

In many similar cases where huge vendors tried to create an OS from scratch in a TD manner, either the cost and time needed were underestimated, or the technical reality was not considered adequately. For example, Spring, an early 1990s object oriented OS developed at Sun, never really got to the point where it could compete with the traditional Unix derivate Solaris. And when Intel first published its 64-bit plans, the companies IBM, SCO (then Caldera), amongst other firms failed to finalise project Monterey aiming to be the first commercial software platform for the upcoming processors. IBM has moved on to using a BU evolved Linux, and SCO entered a legal battle against the former partner IBM for that. Systems like Monterey, Sun Spring, OS/2, and Rhapsody were well funded, but either never released, or have disappeared in the meantime while the old BU Unix derivates are the only ones left besides – as well BU driven – Microsoft products.

5.3. McIDAS

When the Man computer Interactive Data System (McIDAS) celebrated its 25th year of existence, the American Meteorological Society published a article [11], explaining the BU based evolution of this immensely complex system which is used to analyse and process data, collected by weather satellites. Since 1973, McIDAS has undergone four major architectural and also design changes, moving the entire processing system consisting of many different components and applications from a mainframe environment over to a distributed Unix architecture on which it still serves its national and international clients today. Interestingly enough, the article describing those architectural changes concludes:

"1) In an environment of change and evolution, the bottom-up design approach produces a system more satisfactory to users than a top-down approach. [...]

2) Requirements are not our final goal, user satisfaction is."

These two conclusions already sum up the fundamental area of conflict as discussed in sec. 3.2: requirements vs. user satisfaction. According to the researchers at Space Science and Engineering Center at the University of Wisconsin, Madison, who are mainly responsible for McIDAS, a BU approach may often be unable to address *all* the requirements, while a TD approach is usually finalised on the expense of flexibility, portability, and evolvability of the *entire* system.

5.4. UNIX

"Those who do not understand Unix are condemned to reinvent it, poorly." — Henry Spencer

Doug McIlroy, the principal inventor of UNIX *Pipes*, summarised the popular OS design principle as follows: "This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface. [21]"

Of course, this does not only hold for text processing shell commands, but it rather reflects the overall — and still practically successful — BU approach that Ken Thompson, the original designer of the UNIX OS [20], originally chose. The focus is on technical aspects such as text streams and the interfaces between programs, but not on a set of requirements that have to be fulfilled.

The fact that most Unix compatible (Linux, BSD, etc.) OS still continue to evolve under this maxim is a clear indication that the almost invariant properties of the physical platform demand far less frequent changes than high-level requirements asserted in a TD manner. As a matter of fact, many of the changes occurring "below" an (Unix) OS core are transparently wrapped by compilers. In other words, an improved pipelining architecture in a modern microprocessor hardly affects the way programmers maintain file systems, network stacks, or device drivers; the bottom-level tends to be rather invariant compared to TD requirements.

In the Unix world, users traditionally adapt to the system's properties, and not vice versa. Instead of imposing a new solution on the system changing, or at least, influencing its overall properties, existing orthogonal building boxes (pipes, filters, scripts, daemons, sockets, etc.) are carefully composed by the user to address the new requirements. In consequence, the changing requirements have a reduced scope and do not affect the evolution of co-existing programs, or processes — clearly a contrast to a TD system design where the set of initial requirements predetermines implementation details.

5.5. Open Source

The Open Source movement, which was largely sparked by the GNU project, is yet another example of successful application of the BU approach. The original goal of GNU was to create a complete, but entirely free OS according to Unix design principles. Interestingly enough, since 1984 the GNU project has failed to meet its TD asserted goal by being unable to deliver a stable OS kernel. However, GNU as a whole was successful, since the building boxes necessary to accomplish their goals, indirectly, gave birth to other highly successful, large-scale and long-living software projects such as Linux, Mozilla, KDE, Gnome, and others.

Instead of building a holistic solution, the GNU project first focussed BU on universal tools that would help build the fundamental components of a Unix system: an extensible text editor (Emacs), a compiler (GCC), an automated build system (make, autoconf, automake), command line interpreters (e. g. bash), and so on. Nowadays, most of these systems have already become integral part of various (and even commercial) Unix operating systems, such as Sun OS or Mac OS X, for instance. Linux and many other highly successful Free Software projects would not exist, if it wasn't for GNU's BU approach on building large-scale systems. In that sense, the Open Source movement has taken the approach of BU software development one step further than Unix.

One of the most successful and comprehensive offsprings from GNU's global efforts is definitely the GNU Compiler Collection (GCC) which, nowadays, supports more platforms and programming languages than any other compiler, i. e. dozens of languages, software and hardware configurations. It is a formidable example taken from the Open Source world of a BU constructed single software system that evolved from a simple C-compiler for 32-bit platforms that address 8-bit bytes and have several general purpose registers [22] to an industrial-strength cross-compiler. Our own experiences working on GCC are summarised in [3].

6. Consequences

Coming to realise the various differences between TD and BU and favouring the BU approach for evolvability has consequences on the way software should be developed as well as for users of software systems.

6.1. Development Processes

TD asserted requirements affect no other aspect of software engineering as much as the development process. Most of the traditional and widely used processes like variations of the waterfall, but also object oriented methods like RUP assume

TD driven development starting with the requirements and refining them down to a technical implementation disregarding other possibilities. Recently, agile methods and Extreme Programming, in particular, have somewhat changed that, but as the majority of systems is not yet created according to these virtues.

Maintenance processes, however, are already often BU driven as the comprehension and analysis of the existing system is an explicit and early step in the process [8]. Given that the biggest share in development costs of a long-living software system is spent maintaining and helping the system to evolve, it seems inappropriate to disregard such aspects in the majority of development processes used today.

We argue that development processes which disregard BU approaches in total are inappropriate to tackle maintenance as well as evolutionary aspects of software systems which constitute the major investment in software products. BU motivated aspects must be regarded rather early in the development process if the system is to evolve successfully in the long run.

6.2. Who Needs to Adapt?

Very often, new requirements introduced to a software system result in modifications to it. Depending on whether the system was created using a more BU, or a TD oriented approach, the changes may affect large parts of the system, independent whether it is object oriented, based on components, etc.

On the other hand, requirements could be reflected without changing the system, but changing the way users operate it, for instance. In fact, many requirements can be addressed without subsequent changes, simply by rearranging the functionality already provided by a system. UNIX, again, provides a perfect example for this, where many of its major applications are nothing but a combination of filters using pipes, and possibly, additional shell commands.

In BU designed systems it should be possible to satisfy many of the users' requirements by giving users the freedom to personalise what is already implemented. TD systems often do not allow for such freedom. Any kind of changes, however, demands a certain creativity and an environment supporting them: this, often, is a lot more sensible on the user side than on the technical side of a system because users are usually less constrained than physical reality.

7. Conclusions

This paper compared BU versus TD software engineering from a software evolution perspective. While BU developed systems have well-known drawbacks, such as imperfectly matching user requirements, we argue that building systems from the technical basis upwards has significant advantages concerning software evolvability. One reason for this is the greater distance of the BU approach from the ever changing requirements providing for more stability. The second reason is higher compositionality because BU decision making forces to think in orthogonal and scalable components. Hence, BU developed systems do not have to be changed as frequently as their TD oriented counterparts and if changes are required, they are easier to accomplish. Consequently, BU developed systems have a higher expectancy of life and evolve healthier.

References

[1] Inform — a design system for interactive fiction. http://www.inform-fiction.org/.

[2] V. R. Basili et al. Goal question metric paradigm. In *Encyclopedia of Software Engineering*. John Wiley & Sons, 1994.

[3] A. Bauer and M. Pizka. The Contribution of Free Software to Software Evolution. In *IWPSE*, Helsinki, Sept. 2003. IEEE CS.

[4] K. Beck et al. Manifesto for agile software development. www, 2001. http://agilemanifesto.org/.

[5] Comandos Consortium. *The Comandos Distributed Application Platform*. Comandos Consortium, 1992.

[6] S. G. Eick et al. Does code decay? Technical Report 81, National Institute of Statistical Sciences, Mar. 1998.

[7] M. Griss et al. Systematic software reuse (panel): Objects and frameworks are not enough. In *ACM SIGSOFT Symposium on Software Reusability*, pages 17–20, 1995.

[8] C. S. Hartzman and C. F. Austin. Maintenance productivity. In *Conf. on Collaborative research: Software Engineering*, Toronto, Ca, 1993. IBM Press.

[9] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.

[10] W. Kadir and P. Loucopoulos. Relating evolving business rules to software design. In *SERP*, Las Vegas, June 2003.

[11] M. A. Lazzara et al. The man computer interactive data access system: 25 years of interactive processing. *Bulletin of the American Meteorological Society*, 1999.

[12] M. Lehman. The programming process. Technical Report RC2722, IBM Research, Yorktown Heights, NY, Sept. 1969.

[13] M. Lehman. Software evolution threat and challenge. Professorial and Jubilee Lecture, Oct. 2003. 9th international Stevens Award, hosted by ICSM 2003.

[14] N. Medvidovic, R. Taylor, and D. Rosenblum. An architecture-based approach to software evolution, 1998.

[15] N. Montfort. Toward a theory of interactive fiction. In E. Short, editor, *if Theory*, to appear 2003.

[16] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. of the ACM*, 15(12), Dec. 1972.

[17] M. C. Paulk et al. Capability Maturity Model for software 1.1. Technical Report SEI-93-TR-024, SEI, Pittsburgh, Feb. 1993.

[18] T. M. Pigoski. *Practical Software Maintenance*. Wiley Computer Publishing, 1996.

[19] V. T. Rajlich and K. H. Bennett. A staged model for the software life cycle. *IEEE Computer*, pages 2–8, July 2000.

[20] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Comm. Assoc. Comp. Mach.*, 17(7):365–375, July 1974.

[21] P. H. Salus. *A Quarter Century of UNIX*. Addison Wesley, Boston, 1994.

[22] R. M. Stallman. GNU Compiler Collection Internals. http://gcc.gnu.org/onlinedocs/gccint/, 2004.

[23] Standish Group International. Chaos, 1995.

[24] STSC. Software Reengineering Assessment Handbook. Technical report, STSC, U.S. Department of Defense, Mar. 1997.

[25] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[26] A. von Mayrhauser and A. M. Vans. Comprehension processes during large scale maintenance. In *ICSE*. IEEE CS Press, 1994.