Facets of Software Development

Computer Science & Programming, Engineering & Management

Liber Amicorum:

Professor Jaco W. de Bakker

Dines Bjørner
Department of Computer Science
Technical University of Denmark
DK-2800 Lyngby
Denmark

March 9, 1989

Abstract

The Mathematics Centrum — now the Center for Mathematics and Informatics (MC/CWI) — has contributed significantly to the computation sciences. The involvement of the CWI in large scale information technology projects appears to destine the CWI to also contribute to computation engineering.

In this note allow me to speculate on a context in which our many, individual contributions to the computation sciences may fit into practical life.

Contents

1 Introduction

Three more-or-less un-co-ordinated directions of advanced engineering development and scientific research are taking place within computation today: (a) very large scale, advanced systems and software development environments (SDEs) are being "researched", built and experimented with — by the european information technology industry; (b) the European Economic Community (EEC) has, within the ordinary ESPRIT programme, involved a large number of computer scientists in a number of rather goal-oriented, ie. applied research projects; and (c) "good old computer science" — as pursued by individual scientists — is still adding stone-upon-stone of interesting results to a "mountain" of paper.

In relation to point (a), this note reflects upon (i) the professions and professionals who are going to use these environments, (ii) the project and product qualities these environments are to support, and (iii) the methods, principles and techniques according to which the professionals use the environments.

This note attempts to taxonomise the disciplines and professions of the field of software development. We do so by enumerating salient features of 'Management', 'Software Engineering', 'Programming', and 'Computer Science'. The harmonious co-operation of professionals of these fields is necessary in the successful development of successful software products. Such harmonious co-operation must be guided by a method, with its techniques and tools, clearly perceived by all professionals.

The note is organised as follows: in section 2 we define four software development professions and the tasks of four groups of corresponding professionals. In section 3 we outline the kind of project and product qualities that these professions and professionals must all strive to achieve. And, finally, in section 4 we briefly mention some issues of methodological nature — such which should bind the professionals together, and into whose moulds our engineering, programming and scientific research should deliver practically useful results.

2 Four Axes of Software Development

2.1 Definition of Disciplines

There is the subject field, and there are its practitioners. In this subsection (2.1) we deal with the former, while in the next subsection (2.2) we deal with the latter. The

last subsection (2.3) of this section then expands on the field and its practitioners.

2.1.1 Computation Sciences and Engineering

Definition 1 *The computation sciences deal with what objects (data and processes) can exist inside computers and how they are constructed.*

Characterisation 1 *The computation sciences consist of computer science, computing science and hardware+software "science"*[1].

The above characterisation, although presently left mostly undefined, is perhaps dogmatic, but it will work well for us; and whatever short-comings it might have, will not shine through too much.

Definition 2 *Computation engineering consists of computer (or hardware) development and software development.*

So we divide our world into science and engineering. In science we try to understand; in engineering we succeed in practice.

2.1.2 Computer Science

Definition 3 *Computer science is the mathematical study of programs.*

In computer science we investigate theories of programs: of **what** programs are, of classes of program schemes, of what can be computed, of computational complexity (how difficult it is to compute), of computational models, and of the mathematical tools and techniques necessary to express and further develop that study.

Examples of sub-studies within computer science are: automata theory, formal languages, program schemata, complexity theory, mathematics of computation (incl. recursive function theory and proof theory studies), computational geometry, theory of denotational semantics (Scott domains, metric speaces, power domains, etc.), theory of algebraic semantics, computational category theory, etc.

2.1.3 Computing Science

Definition 4 *Computing science is the mathematical study of programming.*

In computing science we study **how** to construct, or develop, programs. Another word for computing science is programming methodology.

Examples of sub-studies within computing science are: information systems analysis, (conceptual) data modelling, requirements definition, functionality definition (ie. architectural specification), program transformation (incl. stepwise program refinement) techniques, inductive and deductive program development, program correctness verification and proof, machine coding techniques, functional programming, logic programming, imperative programming, parallel programming, compiling techniques, database techniques, etc.

[1] We put double quotes around 'science' since we do think that — whatever it is — it is presently not a science.

2.1.4 Systems "Science"

Definition 5 *Systems science is the pragmatic, empirical study of hardware/software systems and their development.*

In systems "science" we study the practical aspects of development: modularisation — for purposes of re- and co-operative use, version control and configuration, variant journaling, test case generation and validation, requirements & design decision tracking, development management monitoring & control, resource estimation, process (ie. development management) models, as well as the pragmatics of tooling for these practical aspects.

2.1.5 Hardware and Software Development

Definition 6 *Hardware and software development consists of, or is the practice of management, engineering, programming and theory checking.*

These are then the four axes of software[2] development: its management, its engineering, its programming, and its theory checking.

o o o

Note 1 *The above definitions are just that. You may not agree with them. You may rather wish to prefer already established "definitions" of the named fields. Be that as it may. If you think you disagree, then replace, for example, 'computer science' and 'computing science' with x and y. Now you have less grounds for disagreement! So: please do not get hung up on our definitions. It is always useful to attempt to clarify definitions — if for nothing else, then as an analytic exercise that sharpens our insight.*

2.2 Characterisation of Developer Rôles

From now on we limit our attention to software.

The previous subsection postulated a quartet of sub-disciplines, and hence of potentially distinct developer rôles in software development. We shall now further characterise these.

2.2.1 Management and Managers

Characterisation 2 *The development manager deals with resources.*

Characterisation 3 *The development manager plans overall development, estimate resource requirements, budgets these, establishes financing for these, allocate and schedule resources (people, machine (and other tools), time), and monitors and controls resources.*

[2] — and, correspondingly, hardware

Characterisation 4 *The development manager builds the organisations required to carry out development.*

Characterisation 5 *Management builds on the laws of sociology (econometrics, management science, etc.).*

Characterisation 6 *Managers treat resources as social objects: humans, money, time.*

2.2.2 Engineering and Engineers

Characterisation 7 *The software engineer accomodates programs and programming to existing hard and soft technologies.*

Characterisation 8 *The software engineer deals with such issues as fitting development tools to existing computer systems, constructing versions of software, configuring products from versions, provide for journaling of variant developments, build test cases, validate designs and implementations, etc.*

Characterisation 9 *The software engineer builds tools.*

Characterisation 10 *Software engineering builds on laws of software "science" and the natural sciences.*

The laws of the natural science enter our concern in connection with physical (electrical and other) limitations of ("the always current") technology.

Characterisation 11 *The software engineer treats software as physical objects.*

Software is subject to execution, and execution behaviours are examined (say for purposes of testing and validation), related documents are "walked through", and in general: quality assurance is something which treats documents and code syntactically.

Note 2 *Our definition of 'Software Engineering', and its implications, appears to be different from that of 'Software Engineering' as generally formulated. We stick to our definition. We cannot make much sense out of any other definition. Still we are not entirely happy with the situation altogether.*

Note 3 *European computation sciences seem to have focused very much on contributions to computer and computing science, whereas US computation sciences appears to have contributed to not quite overlapping facets of computer science and to software engineering.*

2.2.3 Programming and Programmers

Characterisation 12 *The programmer defines requirements, designs the architectural components, specifies functionalities, transforms specifications into code, and argues correctness of these steps.*

Characterisation 13 *The programmer constructs theories by adhering to rules of programming methodology.*

Characterisation 14 *The programmer builds on laws of computer and computing sciences.*

Characterisation 15 *Programmers treat programs and programming as formal, mathematical objects.*

Programmers reason about the formal, mathematical objects denoted by what is written down in formal requirements, specification, design and code documents, and the relations between such documents.

2.2.4 Theory Checking and Computation Scientists

Programs express theories. Programming is "executing meta-programs". New software applications usually go beyond state-of-the-art understanding.

Characterisation 16 *The resident computation scientist checks whether the programmer expresses proper theories.*

Requirements definitions, functionality specification, and program transformation do not always, all the time, stay within established theories. Usually, however, what the programmer intends is perfectly permissible, only the theories appear to not cater for the special cases, and the resident scientists must check for compliance. This checking may involve applied research into areas of computer science. This research produces new computer science results.

Characterisation 17 *The resident computation scientist builds new computation models and theories.*

Characterisation 18 *Theory checking builds on the laws of mathematics.*

2.3 Programs and Software

We have taken for granted definitions of 'programs' and 'software'. We now seemingly repair that omission.

Definition 7 *By a computer 'program' is meant a set of 'instructions' capable, when incorporated in a 'machine-readable' form, of causing a 'machine' having 'information-processing' capabilities to indicate, perform or achieve a particular function, task or result.*

A program is a single, indivisible structure which prescribes an algorithm over some data structure. When speaking of a program we speak of its "internal" qualities.

You observe that we have defined one concept, program, in terms of at least four other (undefined) terms! For the time being we rely on your previous knowledge of what a program is — and otherwise beg your indulgence.

Definition 8 *By 'program description' is meant a 'complete' 'procedural' presentation in verbal, schematic or other form, in sufficient detail to determine a set of instructions constituting a corresponding computer program.*

Definition 9 *By 'supporting material' is meant any material, other than a computer program or a program description, created for aiding the understanding or application of a computer program, for example problem descriptions and user instructions.*

Definition 10 *By computer 'software' is meant any or several of the items referred to in the previous three definitions.*

A software, or programming system is a collection of programs for the specific purpose of solving an externally stated problem. When speaking of software we speak of its "external" qualities.

The above four definitions (with their appended qualifications) were brought so that you understand the difference between program and software, program system and software system.

2.4 The Four Axes

So the four axes of software development (and its professions) are: management (managers), software engineering (software engineers), programming (programmers), and theory checking (computation scientists).

Oftentimes one and the same person plays all four rôles, sometimes unaware, sometimes frustratingly conscientious of the overlap.

3 Software Quality Measures

SO WHY ARE WE DOING ALL THIS SCIENCE "STUFF"? WHY DO WE HAVE ALL THESE THEORIES, AND WHAT'S THE GOOD OF IT?

We do it, and have it, for three reasons: (i) because we want to understand what we are doing, (ii) because we want to make sure that we are doing it right, and (iii) because it is fun knowing and doing it right!

But what do we mean when we say 'right'; how do we "measure" that which is right, and how much right it is? We do it by subdividing what has to be "measured" into separate, hopefully quantifiable "issues", or qualities. We divide these first into product and project qualities, then each of these fields into sub-fields, etc.

3.1 Software Product Qualities

Here we are interested in describing qualities of software products. We list 8 quality criteria. Except for the last, we claim them independent of one another. Thus we can speak of for example 'correctness' independent of 'reliability', etc.

1: Fitness for Purpose :

Definition 11 *Software is fit for its purpose if it meets the following kinds of more or less independent expectancies:*

- *each concept of a "real (or perceived) world" is mapped one-to-one into functions and facilities of the software,*
- *the software addresses ('solves', computes) the customers problem,*
- *it is reasonably easy to train the users of the software,*
- *the software can be used commensurately easily (operativeness),*
- *the software is ergonomically adequate (physically user friendly), and*
- *the software is adequately concise, where relevant, in its observable behaviour.*

Some people prefer to use the overall, encompassing characterisation: *user friendly human computer or man-machine interface (HCI, resp. MMI)* for a subset of the above facets. We prefer to be precise so that we can "measure". We also find that the first facet "drives" most remaining: isomorphism between external world concepts and software functions and facilities favourably determines most remaining facets.

2: Correctness :

Definition 12 *Software is correct if it meets its functional specification, ie. if and only if the realisation can be proven correct wrt. the specification.*

3: Reliability :

Definition 13 *Software is reliable if it clearly prescribes the rejection of invalid input data.*

Typically a compiler rejects syntactically incorrect programs.

A functional specification defines behaviour of a system wrt. acceptable input, but just specify **undefined** or **error** for unacceptable input. A reliable system would safeguard the system against such undefined, erroneous input — while a fault tolerant system, see next, would go further.

4: Fault Tolerance :

Definition 14 *Software is fault tolerant if it clearly prescribes "repairs" to erroneous input, and/or to spurious changes in internal data values.*

Typically a compilers' error-correcting parser 'recovers' from 'minor' errors in input programs.

5: Security/Integrity :

Definition 15 *A software system is secure if an un-authorised user, while anyway using the system (i) is not able to ascertain* **what** *the system is doing, (ii) is not able to find out* **how** *it is doing it, (iii) is not able to prevent the system from operating, and (iv) does not know whether he knows!*

Software integrity must be designed directly into the product, already from its identification, and certainly in its function specification.

Software is maintainable if it is extensible and repairable.

Software extension involves "adding" new functionalities to the software.

Repairability involves three rather distinct issues: software may need **perfective** maintenance in order to improve its performance, **adaptive** maintenance in order to fit it to a changing environment, and **corrective** maintenance in order to repair 'bugs'. In all instances 'maintenance' implies changing the software.

When maintenance of a building for example requires finding the exact location of pipes embedded in walls before drilling into these, then engineers' blue prints are inspected before drilling, and when walls are planned to be torn down, then engineering calculations are (re)done to see whether crucial support goes below set standards.

So it is with software: its maintenance may require inspection of any number of development steps, from requirements, via specifications and abstract designs to concrete designs to locate points of 'maintenance'.

Somehow that for which maintenance is enacted defines a measure of complexity against which the effort (time, cost, etc.) of carrying through the maintenance can be compared. Hence:

6: Maintainability :

Definition 16 *Software is said to be maintainable if the complexity of the 'thing' for which maintenance is enacted stands in some reasonable relationship to the effort of carrying through the maintenance.*

Operationally speaking we can say that if the cost of maintenance is directly proportional to that (otherwise undefined) measure of complexity, then the software is maintainable.

Assume a succession of changes: from the first point in the chain of development documents — where we see that a maintenance change need be expressed (requirements definition - function specification - abstract design - · · · · - concrete design) —

to where the change eventually finds its change in code. If this propagation can be simply ("linearly") contained (and does not, for example "mushroom" all over the code), then the software is maintainable. The preceding is, unfortunately, not a very good characterisation. A better definition would require a longer discussion.

7: Portability :

Definition 17 *Software is portable if it can be "moved easily" from one computing system to another.*

The above "easiness" has not been defined. Hence the definition is meaningless. The point is this: when we define such things as maintainability and portability, then we first need erect a whole set of auxiliary, and properly defined notions — as was implied in the definition and characterisation of maintainability (viz.: complexity, homomorphic, linear, measure, propagation, etc.). For portability we need a similar set of auxiliary concepts — basically amounting to the same as for maintainability.

Porting software is a special kind of adaptive maintenance of software.

A compiler is portable if it can be moved from one operating or machine system to another — and this kind of portability is called re-hosting. If the compiler can be easily changed so as to generate code for another than the originally intended target machine, then the compiler is said to be re-targetable. So compiler portability implies the sum of two things: re-hostability and re-targetability.

8: Robustness :

Definition 18 *Software is robust if its maintenance does not impair any of the qualities mentioned in this section — incl. robustness!*

3.2 Software Development Project Qualities

Here we are interested in qualities of the process by which software is developed.

1. Planning :

Definition 19 *A project can be planned if a method can a-priori be identified, a method which in deterministic steps of development concisely prescribes how to develop the product.*

2. Estimation :

Definition 20 *A projects' consumption of resources (time, manpower, machine and other development tools) can be estimated if for each sub-task that the method (any method) requires, the effort to carry through this task can a-priori be established.*

3. **Resourcing** :

   Definition 21 *A project is resourcable if expected and available resources can be "nicely" mapped onto required resources.*

   The crucial term here is 'nice'. We did'nt define it! If the resources required during a project does not vary dramatically, but can be built up, kept steady, and subsequently tapers off, in an orderly fashion, then the project is resourcable. Resourcability thus depends on the nature of the artifact being built — but knowledge about resourcability can be ascertained once initial planning and estimation has taken place, ie. early!

4. **Economic** :

   Definition 22 *A project is economic, firstly, if the estimated required resources are within expectations, and, secondly, if the actual resources consumed (ie. accounted) stays within budget!*

5. **Trustworthy** :

   Definition 23 *A project is trustworthy if (i) the development staff and the customer at all times believes that management is in full control, (ii) if the developers believe that their method will bring them through the project, and (iii) if the customer believes he will get the quality software at the time he expects.*

6. **Enjoyable** :

   Definition 24 *A project is enjoyable if the development staff has intellectual fun, and is being further educated in pursuing it.*

4 Methods and Principles

4.1 Methods

Definition 25 *A method is a set of procedures for selecting and applying, according to a number of principles, a number of techniques and tools in order to efficiently achieve the construction of a certain, efficient artifact.*

So a method is an orderly arrangement, a procedure or process for attaining an object; a systematic procedure, technique, or mode of inquiry employed by, or proper to a particular discipline; a systematic plan followed in presenting or constructing material; a body of skills and techniques; and a discipline that deals with principles and techniques.

Definition 26 *Methodology is the study of (knowledge about) methods — not just one, but a family, and not just an individualised knowledge, but also a comparative knowledge.*

So a methodology is a body of methods, rules and postulates employed by a discipline; or the analysis of the principles of inquiry in a particular field.

The JSP (Jackson Systems Programming) is a method for constructing a class of software applicable in the context especially of sequential file processing.

The JSD (Jackson Systems Design) is a method for constructing a class of software applicable in a context which extends JSP to parallel (or concurrent) process oriented transaction processing, including what would otherwise be classified as database processing.

Other, claimed, methods are: VDM, SADT, Yordon, etc. Some are formal, some are ad hoc; some cater for a narrow spectrum of either the software life cycle or the application field, or both.

4.2 Principles

Definition 27 *A principle is a comprehensive and fundamental law, doctrine, or assumption; the laws, or facts of nature or mathematics, underlying the working of an artificial device; or a rule, or code, of conduct.*

Let us, however, already here try to identify some such principles — in the form of itemized lists[3] which we do not presently comment upon:

General Principles :

- Analysis of existing artifacts and theories
- Combination of design and analysis
- Decomposability and reducibility
- Abstraction
- Limits of scope and scale
- Divide and conquer
- Impossibility of full capture
- ...

Principles of Philosophy :

- Prevention is better than cure
- From systematic via rigorous to formal approach
- Prove properties rather than test for satisfaction
- ...

Concrete Principles :

- Iterative nature of development

[3]These lists are drawn, directly from: *Software Engineering Education*, eds.: Norman E. Gibbs and Richard E. Fairley, Springer-Verlag, 1987, pp 370-371.

- Representational and operational abstraction
- Denotational vs. computational semantics
- Applicative vs. imperative function definition
- Hierarchical vs. configurational development and presentation
- Discharge of proof obligations
- ...

Principles of Methodology :

- Reduction principle: the whole = the sum of the parts
- Discrete nature of software development: case analysis, induction, abstraction
- Embedded nature of software: impact of context, environment, and enclosing system
- ...

4.3 Techniques and Tools

The computer and computing sciences work bottom-up: provide simple, sometimes elegant techniques and tools that hopefully fit into some method, and hopefully can be used according to some principles.

The computation sciences appear to have a very long way to go before they seem ready to maturely approach the problem of synthesizing methods and enunciating clear principles.

Meanwhile many ad hoc tools will be developed, and hundreds of thousands of hours will be spent on techniques without principles, and on methods with no theories.

5 Conclusion

We have set the stage for what we believe are relevant issues in software development. We have not solved any problems of technical nature. But we have resolved what it is we should be looking for when methods, principles, techniques, and tools are claimed useful in software development.

They have to help guarantee product and project qualities, and they have to fit into (reflect, or mirror) a world consisting of managers, engineers, programmers and theory checkers.

The note can be construed as advocating more unity and clear direction in the fields of computation sciences in support of engineering.