# Software Engineering Approaches Before the Notion

Hans Dieter Hellige

Research Center artec
University of Bremen
Enrique-Schmidt-Str. 7
D-28334 Bremen
Hellige@artec.uni-bremen.de

*Paper presented at the conference*
*„Pioneering Software in the 1960s*
*in Germany, The Netherlands, and Belgium"*
*Centrum Wiskunde & Informatica (CWI) in Amsterdam*
*2.-4. November 2006*

Organizers: Gerard Alberts (UvA Amsterdam), Paul Klint (CWI, Amsterdam), Wilfried Brauer (TU München), Arjan van Dijk (NGI), Ulf Hashagen (MZWTG/Deutsches Museum), Hans Dieter Hellige (Universität Bremen), Jacques VandenBulcke (SAI/Universiteit Leuven), John Impagliazzo (IFIP)

Supporting Organizations:

- Societies: GI, NGI, SAI, IFIP Working Group 9.7 „History of Computing"
- Universities and Research Institutes: CWI, Deutsches Museum/MZWTG, TU München

## 1 Introduction

Historical descriptions of the development of software engineering and software design in general continue to be highly fixated on the NATO conferences in Garmisch and Rome. They usually adhere to the self-understanding of the organisers and participants at the two conferences, as expressed in many publications. Ever-recurrent retrospections to 1968/69 have reinforced again and again the view of the conferences being a key turning point in the history of software. There thus ensued a rather oversimplified argument: the 'software crisis' that arose in the second half of the 1960s triggered the initiative behind the conference that then resulted in the discipline called 'software engineering'. (I need only refer to the programme for the Amsterdam conference: "By the end of the 1960s some perceived the field as being in a crisis, others saw new challenges." The famous Garmisch-Partenkirchen Conference in 1968 marked the self-conscious start of a new discipline called software engineering."). It was not until more recent studies that consideration was also given to software development in the pre-Garmisch period.

This narrow focus was made possible by the exceedingly clever arguments and institutional approach taken by the organisers of the Garmisch conference, and by the efficacious propagation of the policy programme adopted for research and the profession in general. A greatly exaggerated description of the so-called software crisis was merged with the offer of clearly articulated recipes for solving said crisis. In the report that soon became famous, the differences in view that did indeed exist, and which became obvious in the follow-up conference, were then harmonised to form a single, standard message. This was finally bundled into a catchy metaphor that mutated even during the conference itself into an autosuggestive guiding model. Although this 'staging' of a paradigm shift in the field of software design was crowned with success, evaluation of how successfully the discipline has developed is still a matter of considerable controversy within the software community. Some see stepwise, spiral or dialectical progress, while others refer to a chain of failed approaches or even decades of going astray.

The subject of my paper is not the problem of arriving at some overall evaluation of software engineering – something that not even a historian of science and technology could resolve – but rather the assessment of the NATO conferences as paradigm shifts and epochal turning points in the history of software development. This latter perspective was underpinned by various participants who applied development models from the disciplines of general history and economic history. For Dijkstra, for example, the Garmisch conference denoted "the end of the middle ages" and the beginning of the "Age of Enlightment". Other participants referred to the stages in the development of industrial production to characterise the initiated transformation process in software production. It is striking in this regard that historical models were being applied, on the one hand, while abrupt actionistic steps were being taken to dissociate oneself from such models, on the other hand. McIlroy, for example, advocated that "crofters" be replaced straight away by "mass-production techniques", while F. L. Bauer noted in his report in "Datamation" on the Garmisch conference that the crux of the matter was "to switch from home-made software to manufactured software, from tinkering to engineering." Here, as in most historical outlines, there is no manufactory period, considered by Adam Smith, Charles Babbage and especially Karl Marx to be an essential logical transition from craft to industrial production.

As a critique of the still dominant view the paper outlines the approaches to 'software engineering' in the 1950s and 1960s distinguishing two differing cultures overlapping in time:

- the software manufacturing in the military and governmental software contractor sector, which was influenced by operations research and system engineering and placed the focus on a team-oriented division of labour, and

- the concept of software architecture in the corporate software sector which combined hierarchical organization principles and structural methods with an integrated design philosophy.

## 2 Software manufactories in the military contracting sector under the influence of operations research and systems engineering

The reservoir of ideas associated with software engineering dates back to the earliest days of modern computing. Indeed, one can trace comparisons between the production of programs, on the one hand, and the division of labour in manufacturing and industry, on the other, as far back as Babbage. As early as 1946/47, John v. Neumann was developing a 6-phase model of programming, in which he worked on the assumption, like Alan Turing in those days as well, that the development process inherently involved the division of labour. In 1947/48, George R. Stibbitz was already modelling program and computer procedures as a system of hierarchical levels, and expected that this would facilitate development, testing and modification processes likewise based on a division of labour. Alwin Walther, in the years between 1946 and 1952, was the first to draw immediate parallels between program production and the planning of an industrial production process. With their concept of program assembly using standard sub-routines, Wilkes, Wheeler and Gill went beyond such purely mental analogies, transposing the 'modular design philosophy' from electronics to software design in 1951. Yet even at this low level of modular program assembly, they could already see the problem of determining 'the best way to construct subroutines' and selecting the most appropriate variants.

However, engineering concepts did not achieve any genuine breakthrough in the software field until the mid-1950s, in the context of large-scale software systems developed by software contractors in the military and aviation sector. The operating and evaluation programs created between 1952 and 1961 for the SAGE airspace monitoring system signified a quantum leap in scale (1.1 million instructions in total), with more than 800 instead of the originally projected 100 programmers deployed on the project, assisted by a further 1400 personnel. The 'SAGE program contractor' was initially the MIT Lincoln Laboratory, which was succeeded by a department of the Rand Corporation from which the System Development Corporation (SDC) later ensued. In order to handle the scope and complexity of the task at hand, the managers of the project applied the repertoire of engineering and project management tools previously used in hardware development. Herbert D. Bennington, with hindsight, considered the direct transfer of methods from hardware to software design to be the crucial factor for success: 'It is easy for me to single out the one factor that I think led to our relative success: We were all engineers and had been trained to organize our efforts along engineering lines.' (Annals 5, No. 4, p. 300)

The methods imported related above all to approaches in the field of operations research and 'systems engineering', which had been developing since about 1943 in the Bell laboratories as ways of controlling complex projects in communications technology. These were accompanied by Taylorist instruments of 'industrial engineering', such as flowcharts and Gantt diagrams for tracking defined 'milestones'. The first model for approaching the development of large-scale software systems was born of this methodological mix. The well-known phase model was designed in 1955/56 by the 'head of programming' at the Lincoln Lab, John F. Jacobs. It is generally attributed to Herbert D. Benington, because it was he who first presented the concept at a military conference on 'programming methods' in June 1956, and who published the paper in the SAGE issue of the Annals, whereas the more detailed history of the model's creation in Jacobs' SAGE memoirs were barely heeded.

The basis for this procedural model involved the systematic modularisation of the entire program in 'large decentralized programs' for input, output, bookkeeping, control and processing, which were subdivided respectively into single blocks according to their 'data-processing functions'. The development process was then split into nine phases. The first comprised the 'general operational plan' to be jointly developed by the 'system engineers' and the users. The 'machine' and 'operational specifications' were derived from the 'requirements' specified in the plan. In those specifications, the computers, terminals and system programs were still being viewed purely as 'black boxes', i.e. as abstract functions that were not specified in any concrete detail until the following program specification phase. The 'component subprograms' subsequently produced as distributed tasks were later tested and assembled in a series of steps to form ever-larger 'subassemblies', and finally the 'main program'. The end of the process was the 'shake down', meaning the integrated testing of hardware and software in the 'operational environment'.

This model approach, based as it was on the industrial production and assembly process, was aimed above all at reducing the amount of communication between the programmers and at enabling the deployment of a large number of 'relatively inexperienced programmers': 'This considerably simplifies the design problem; after the blocks have been documented, groups of programmers can be assigned to each part with the assurance that little communication between these programmers will be necessary.' (Benington, p. 356.) This procedural model thus corresponds to a relatively inflexible top-down approach and a rigid 'waterfall' model. Benington interpreted the SAGE software development approach, also with hindsight, as a small-team or chief-progammer concept with a structured form of approach. He saw himself as a 'chief engineer who was cognizant of these activities and responsible for orchestrating their interplay. In other words as engineers, anything other than structured programming or a top-down approach would have been foreign to us.' (Benington, p. 351.)

In the course of the development process, however, the limits to and problems with the chosen process model became ever more apparent. For example, it was impossible to maintain the strict division of labour that was originally intended for the development of 'decentralized programs' in a realtime system with high interdependency of the discrete processes. In the control programs for integrating all the single components into a single system, an enormous amount of coordination work had to be carried out, with the result that head office had to intervene constantly, as the parallel graphs in the chart show. What then happened was that neither the original hardware nor the specifications of the software system proved stable. The top-down 'stages' model based on electronics design turned out to be particularly disadvantageous in the software development process. Since a modification of one 'component subprogram' gave rise to a whole chain of modifications in other subprograms, about half of the 800 programmers were purely focused on making adjustments to programs. Software completion was delayed by a year as a result, and disrupted the schedule for the entire project. After implementation, it was the software that proved to be the critical bottleneck in the SAGE system. The problems associated with the procedural model chosen led to modifications in project management even during the project itself. The previous barriers to communication between the programmers were lifted, because this was the only way to keep the scale of changes within limits. For Benington, the crucial factor enabling this unprecedented and pioneering software project to be completed after all was that the 'disciplined approach' otherwise prevalent in engineering was not applied one-to-one to production of the program. It was realised 'that computer programming and the computer programmer were 'different'. They could not work and would not prosper under the rigid climate of engineering management.' (Benington, p. 51.)

The software development methods applied in the SAGE project, which were subsequently elaborated and refined by the System Development Corporation, acquired the status of models to emulate for the following large-scale information systems for the military, the aviation sector and for communications technology. These, too, attempted to tackle the much-bemoaned increase in the complexity of online, realtime and time-sharing systems with phase models, milestones and the splitting of giant programs into 'blocks' or 'modules' that were then distributed to different specialists. However, the approach to structuring the product and the process did not adhere quite as strictly to the industrial engineering template as had been the case in the SAGE project. The literature on software development methods also refrained from recommending strict division of labour and rigid workflow planning. In the following, I would like to demonstrate this by referring to some outstanding treatises on 'systems engineering' and program development methods.

The first 'Systems Engineering' textbook, presented in 1957 by Harry H. Goode and Robert E. Machol, and based primarily on experience gained at Bell Labs and AT&T, provided an ideal-typical 'systems design' phase model for computing and automation projects (Goode, Machol, pp. 35 ff.). And yet the authors did not interpret the 'well-defined phases' as strict chronological organisation, because in practice a phase is 'often unrecognizable until it has passed' (ibid., p. 35). Accordingly, the prototype they chose for their model was not, as in later phase models, the bar chart introduced as early as 1911 by Henry Lawrence Gantt, but a flow diagram akin to the schematic diagrams used in electronics. System components and alternative solutions were also included, so the transition from phase model to task plan was a fluid one. Like Goode and Machol before them, Tinus and Och, two system engineers from Bell Laboratories, come to the conclusion soon after, that, although ‘mileposts’ were essential for the main development stages and defined phases, as were timelines for each separate unit, if the growing complexity of automation and information systems were to be controlled, they had definite limitations in practice: ‘Generally, these steps follow in chronological order. However, many of the detailed steps actually overlap and even recur during the development program.’ (Tinus, Och, p. 8)

The software development principles articulated from 1960 onwards by methodology specialists at SDC and by project managers working on large-scale military software systems generally understood phase models as mere guidelines for program development that were not to be slavishly adhered to during the development of specific software systems. This is evident from the first two State-of-the-Art Reports, by Hosier and by Holdiman, on management techniques for the program development of realtime systems. Both reports received little attention from the early software engineering community, despite being published in key journals. Only later did it become apparent, as noted by Barry W. Boehm, ‘how many of today’s software engineering hot topics had already been understood in 1961 in Bill Hosier’s IRE article.’ (Annals 5 (1983) 4, p. 351)

In his 1961 theory of method for the ‘system genesis’ of ‘real-time digital systems’, William A. Hosier collated his years of experience in SAGE program development at the Lincoln Lab and in the ‘system engineering’ of aircraft control systems and the BMEWS anti-missile system at Sylvania. He particularly emphasised the importance of close collaboration between the various specialists involved in system development. Most importantly, all the hardware designers and programmers in the small and highly competent team for the ‘tentative design plan’ had to ‘be able to communicate which each other’, if necessary by calling in ‘one or two men of more general system experience to bridge these gaps and to help resolve differences of viewpoint.’ (p. 101) For Hosier, the program itself was first and foremost a ‘product of team effort’. For that reason, he also believed it imperative when using a task distribution approach ‘that each programmer must clearly see his role in the system and understand all the rules.’ (p. 108). However, like Frederick Brooks years later, he demanded that the first design, even in the case of very large program systems, be produced by a small team or eminent authority, because: ‘The proverb about too many cooks was never more appropriate than here.' (p. 109) In the subsequent stages of program development, on the other hand, Hosier believed that collaboration and communication were essential for avoiding the risk of parochialism: 'Parochialism is a common fault both of subdivisions of a system programming team and of the teams as a whole.' (p. 114) He therefore advocated that teams have frequent meetings with 'external consultants', at which, alternately, 'selected programmers review their progress and problems as speakers to the group.' The thought process involved in designing 'system programs' was to be visualised graphically, above all, in order for ideas to be transferred 'intact from one mind to another'. (p. 109)

Hosier used an 'over-all flow chart' showing the 'main stages of the process' as the main tool for organising and monitoring the 'real-time program development' process. Although oriented in terminology and notation towards the Taylorist instrument of flowcharting, his form of presentation bears greater similarity to the different 'stages', work packages and interrelated competencies of a network plan. Nor does he understand the flowchart as some '*paragraphing* of the whole program' (p. 110), i.e. as strict, hierarchically controlled scheduling. This is because 'the entire structure can not be 100 per cent foreseen, and must evolve through considerable experiment. But if a sound beginning is made, and if it is kept clear what is frozen and what is fluid, it will greatly lessen the conflicts and re-work that always accompany the assembly process.' (p. 110) A coordinated approach is therefore attempted more with shared 'tools and procedures for program production and testing' than with strict schedules. This means that Hosier understands program development as a process that is both targeted and evolutionary. He has no wish to hamper the creativity of programmers by imposing project management à la SAGE. Instead, he wants to steer that creativity using a collection of team-based, systems engineering instruments and 'programming techniques'. He does not address the latter in detail, because he knows programmers' psychology only too well: 'Programmers differ little from engineers, in general, in their reluctance to stop tinkering with and improving their creations. This is a laudible trait; but as delivery dates approach and time grows short, it has to be restrained.' (p. 114)

In his 1962 report on ‘Management Techniques for Real Time Computer Programming’, which appeared in the ACM Journal, Thomas A. Holdiman likewise drew lessons from his experience as project manager at General Dynamics / Electronics and at the Mitre Corporation that was spun off from the Lincoln Lab. In his treatise, which seems at first to take its cue more from Benington than from Hosier, the process of program creation is based entirely on industrial manufacturing processes. He explicitly sees the ‘program production process’ as the production of components in defined phases and as ‘program assembly’: ‘Programming the central computer in a large system amounts to custom fabricating a number of ‘think’ pieces, each created by a different craftsman and fashioned to fulfill a broad program design. The separate pieces (program components) are then united into a coherent whole [...]. The different parts of the program are required to be mutually consistent as to techniques, methods, and content to an unusually high degree.’ (Holdiman, p. 392) Holdiman’s ‘program assembly’ model is deeply rooted likewise in the engineering metaphors coined by Jacobs and Benington. His models for ‘program component design’ and the entire ‘program production process’ also bear a great similarity to their ‘waterfall’ model. At closer glance, however, it is clear that Holdiman did not think of the specific process of software development as the assembly of components with distinct division of labour, but, rather like Hosier, as a highly collaborative process. Due to the interplay of components within the overall program, programmers had to understand *those* program component processes that interacted with their own parts of the program. They also had to have some idea of the overall context. The ‘component designer’ therefore needed ‘considerable freedom for design’, in order to optimise his product: ‘Part of his *art* is to reconcile his piece of the program with those being produced by the other programmers’. (ibid., p. 393, my italics)

Holdiman thus considers the series production of programs to be uneconomical for the production of realtime programs. Instead, the various components had to be produced in ‘concurrency’, yet: ‘concurrency naturally demands a great deal of coordination among programmers working on interacting components’ (ibid., p. 394). For Holdiman, therefore, the activity of programmers is still ‘craftmanship’, in the last analysis. He sees the sequence of increasingly specific models applied in the development process less in terms of the rationalisation and mechanisation of program production, but rather in terms of gaining a better overview and making it easier to make subsequent modifications: 'The division of the programming task into a succession of models permits change, and even though it implies much reworking of the program from model to model, yet it maintains a working situation manageable from every standpoint.' (ibid., p. 402): Holdiman is therefore anticipating the ideas of 'hierarchical modelling' and 'levels of abstraction' that Dijkstra, Parnas, Zurcher and Randell envisaged in the second half of the 1960s.

The software development principles devised by Hosier and Holdiman were followed in subsequent years by many reports on experience gained, reflections on design issues and analyses of project management in the contracting sector. Even in the 1960s, a rich body of treatises was produced by those involved in SDC, the Air Force and the contractor companies, in which experience with 'large-scale programming' in the central command and control systems of the Air Force, flight surveillance and space travel was analysed. This scattering of essays, ACM and AFIPS proceedings, reports and studies was finally collated in a software development methodology published by Perry E. Rosove, head of the 'Advanced Systems Division' at SDC. His 'Developing Computer-Based Information Systems', which appeared in 1967 after four years' work, is rarely cited in historical summaries, unlike the short, oft-cited paper by Winton W. Royce, dating from 1970, even though the book contained much more fundamental analyses of procedural models and anticipated some key aspects of an evolutionary development process.

For Rosove, too, a 5-phase model formed the starting point for organising the software development process. With its direct reference to Gantt's bar chart, its form of presentation is already very similar to Barry Boehm's waterfall model. However, it differs from the latter, and from Royce's 7-phase model in that it permits iteration loops not only to the preceding step in each case, but also throughout the software life-cycle. Rosove had realised, from observing software development in the large-scale information systems operated by the Air Force, that the system requirements could not be specified exclusively in the first phase as envisaged, but that the initial requirements continually changed in the course of projects and that new requirements could arise even at late stages due to changes and advances in technology. Likewise, ‘system design’ meant for him not only the logical and temporal stage following the ‘requirement phase’, on the one hand, but also ‘a function which is carried out repetitively at different levels of a system development process’ (Rosove, pp. 18 f.).

However, the main reason for adopting a flexible approach, in Rosove’s view, was the ‘close relationship between the user and the software developer’ throughout all phases of development. He therefore advocated what he termed an ‘evolutionary approach’, in which ‘design and production iterations’ invariably overlap (ibid., p. 43 f.). According to Rosove, an ‘evolutionary development process’ that departs from any rigid phase model is also essential because, in contrast to the production of machines that could be manufactured as independent elements based on deterministic principles, the software of information systems represented a non-deterministic ‘particular configuration of independent elements’ that required much more complex feedback loops and trade-offs between alternative hardware and software designs (ibid., pp. 31 ff.). The ‘evolutionary design’ method was adopted by the US Department of Defense and recommended to all teams working on the development of data processing programs for command-and-control systems. If user requirements were to be genuinely implemented, it was essential that needs analysis was not confined to the early stages of the development process: ‘The user of the system should participate in every step of the evolution. He is a vital part of the system and if he delegates his responsibility to an agency outside his organization, there is danger that the user will depend on automated decision aids without realizing the extent to which human judgement of operational parameters has been built into such aids by an outside developer.’ (Christie, Kroger, p. 59)

The structuration concepts and management methods developed by Holdiman, Hosier and Rosove for software production appear at first glance to anticipate the software engineering approaches of the late 1960s, due to their notion of a semi-industrial production and assembly process based on components and modules. A closer look, however, reveals some characteristic differences:

- The structural analogy with interchangeable manufacture [i.e. manufacturing parts with such small tolerances that they can be fitted into any machine] is only weakly pronounced; the interdependent structure of problems and responsibilities is maintained despite efforts to modularise.
- The phase model is not aimed at a strictly 'Taylorist' division of labour, but remains embedded in a concept of concurrent software development and the 'evolutionary approach' that appears modern from today's perspective and in which the program systems evolve in close consultation with the users.
- Program production remains an 'art', a team-like design process in which the realtime programmers perform 'creative work in a highly specialized environment'.

In the decade prior to the NATO conferences of 1968/69, various methods for managing design, projects and development arose in the program and system development of large-scale software and information systems for the military. In many respects, these methods already anticipated the evolutionary approaches later advocated by Meir M. Lehmann, and the 'holistic' methods of participative and agile software development based on teamwork, developer creativity and user focus. The software design methods at that time were therefore very similar in kind to the general engineering methodologies that arose during and immediately after the Second World War, to operations research, system engineering, value analysis, the forecasting methods used by the RAND Corporation and other think tanks (Delphi method, scenario technique), and to the concurrent engineering concepts in the German and American arms industries. Although largely products of the military sector, they all had a team focus, followed the New Deal tradition of interdisciplinary teams of experts, brain trusts and think tanks of the war and post-war years, a tradition that was maintained for a long time in the large-scale software projects of the military-industrial complex.

At the end of the 1960s, however, this team-centred approach to software manufacturing was abruptly stopped due to rising software expenditure leading to greater cost constraints being imposed on the financially and institutionally privileged 'users' in the military. At the annual ACM conferences and AFIPS meetings from 1967/68 onwards, complaints were increasingly voiced about costly ‘one-of a-kind software systems’, the unreliability, burgeoning costs and obstinacy of programmers, complaints that were very reminiscent of opinions voiced at the NATO conferences. Demands were now being raised for stricter regulation and control of the ‘system life cycle’ with regard to ‘reuse’, ‘compatibility’ and ‘transferability’. (Ratynsky 1967, Ward 1969) It was in this context that the waterfall model, which did not permit any recursion, become more and more dominant from 1967 onwards. The guiding notion of a ‘software manufactory’ was thus superseded in the contracting sector by the guiding notions of industrial software production. The ‘component assembly’, the ‘orderly process of software production’ and, from 1971 at SDC, the ‘software factory’ were no longer non-binding metaphors, but strict principles of software development. Very soon, given the increasing dominance of ‘software engineering’, the design treatises of the ‘manufactory’ period were confined to oblivion, with few exceptions. A similar fate was also suffered by the second main thrust of 1960s and 1970s software design methodology, namely the software architecture concept in the corporate software sector, which I would like to discuss briefly at the end.

# 3 Software architecture as a concept for structuration and design in the corporate software sector

This second methodological approach, at the transition from software development as a craft to a semi-industrial process, was the result of efforts by major hardware and software producers in the 1960s to master the growing complexity of operation systems and major applications software for airline reservation systems and the like. Around 1960, operating systems were the largest and most complex software systems that had ever been developed, due to the increasingly complex handling of resource management in batch processing, and the control of user activity in real-time and time-sharing systems. It was also in this field that the most spectacular delays, concentrations of errors and project disasters occurred. Complaints about a 'software crisis' began to arise in this connection as early as 1961/62.

The unmastered complexity of large-scale software systems was therefore an occasion, also in the corporate sector, to look for methods providing clear structuring of program systems and development processes. These were found in strategies for hierarchical systems, modularisation and the creation of different levels of abstraction. Thus, managers of major IBM software projects, such as Robert V. Head and James Martin, were soon to view hierarchical structuring and modularisation as the crucial strategic approach for handling the complexity of large program and information systems. The conclusion drawn in 1963 by Robert V. Head from IBM's 'Systems Research Institute' from experience with major projects such as SAGE, SABRE, etc., was that the 'system complexity' of real-time systems resulting from the scale of such systems and the high degree to which programs interacted could only be managed with a highly structured program specification and a 'disciplined approach to system analysis and maintenance': 'This dictates a requirement for some scheme for artificially segmenting the programs and increases the need for well-defined program-to-program linkage.' (Head, p. 376) His demand was therefore to 'break the problem down in true programmable chunks'. Since the programming group could not simply be confronted with a 'job' involving 50 to 150 thousand machine instructions, it was essential to have specifications 'to compartmentalize this task into easily manageable units of work'. (ibid., pp. 377 f.)

In addition to 'chunks' and 'units', he also speaks of 'levels' within the overall task. In order to use as many 'inexperienced personnel' as possible, as is necessary in large-scale project, while also reducing the amount of communication and coordination between programmers, what is needed are clear distinctions between the various areas of assignment: 'While informal communication among programmers on technical problems is no doubt desirable, it can in a large system seriously impair productivity. One purpose of specifications is to organize and formalize such communication.' (Head, p. 378)

Attaching greater weight in this manner to the strict demarcation of sub-tasks, and having the project managers monitor the division of labour, signals the transition from management and team-centred approaches to software development, based on systems engineering, to procedures based on industrial engineering. These approaches were no longer satisfied with creating model-like phases, and even less with the compromise hitherto between *horizontal* modularisation of tasks and independent structures for solving problems, which failed to reduce the amount of communication to any decisive extent. Solutions were sought instead in forms of system structuring that were consistent with the distribution of tasks, i.e. in *vertical* modularisation and the break-down of complex program and information systems in ways that avoided interference between the parts. This change in methods and strategy is particularly evident in the design theories for real-time and operating systems propounded by James Martin in 1965. These were derived from his own experience as 'original designer' of BOADICEA, the British flight reservation system.

The interconnection between product or task structure, on the one hand, and the organisation of work, on the other – something that was only vaguely addressed by his colleague Robert V. Head at the 'IBM Systems Research Institute', was raised by Martin to a fundamental aspect. He showed that the interdependencies between the segments of large-scale program systems invariably leads to a form of hypercomplexity that can no longer be controlled when tasks are horizontally distributed. The crucial problem for him was the 'problem of control': 'How can the programs that make up a real-time system be developed concurrently by separate programmers when there are so many interactions between them? How can control over the interactions be maintained?' (Martin, p. 327) Martin saw the solution to this problem of hypercomplexity in interdependent structures as residing in classical engineering methods: 'It is interesting to pursue the analogy between this type of system and a complex piece of mechanical or electrical engineering. Some of the techniques that have become traditional in conventional engineering point the way here also.' (loc. cit.)

Three years before the Garmisch conference, the engineering metaphor is already fully present. All that is missing is the actual term 'software engineering'. Analogous to interchangeable manufacturing in the mechanical engineering field, or component structures in electrical engineering, Martin wanted to eliminate the interdependencies between program segments and the complex interaction between programmers by means of a 'division into sub-systems' and demarcation of units even before the development process began: 'The unit is insulated as completely as possible from the other units. The interface between the unit and the rest of the system must be very clearly defined. Every attempt will be made not to modify the interfaces between the units. This means that, before programming begins, the functions of the units must be very clearly thought out.' (ibid., p. 329) He also wanted to minimise vertical interdependencies with a stepped 'model-by-model build up' approach. The objective was to 'freeze' the functional specifications at the end of each development stage and in this way avoid recursions to previous phases in development. He ultimately envisaged a 'central control group' or 'central authority' that gains 'complete knowledge of the system under development', that communicates any modifications that may be necessary after all, and that monitors and controls the entire development process. Although teamwork was not entirely dispensed with under these conditions, Martin focused it primarily on adapting and adjusting the single components, and on generating the required group awareness of the 'importance of the separate elements meshing together cleanly' (ibid., p. 333). In this way, and according to the basic principle that the 'simplicity of individual programming mechanisms' had absolute priority over 'individual brilliance', software development ceased to be an art or a craft. The guiding principle had clearly become that of interchangeable manufacture and the engineering of components ready for assembly.

Yet by the mid-1960s, these rigorous software engineering positions were being fundamentally questioned by other notions of design in the IBM context. For Frederick Brooks, especially, who was head of hardware and software development for the /360 system, *decomposition methods* did not suffice for handling complexity in software systems. Since the issue here was to balance the heterogeneous interests of the stakeholders involved and to integrate divergent design requirements in a user- and use-compatible way to create a consistent and cost-efficient system design, software development is also based to a significant extent on *composition methods*. In his famous IFIP lecture in 1965 ('The Future of Computer Architecture'), Brooks suggested the term 'software architecture', analogous to the term 'computer architecture' that he himself had coined in 1960/61.

In using the term *software architecture*, he was not dispensing with methods of structuration, but rather adding to them – especially in the design phases in which structures are formed – using design methods that centred on the wishes of the 'users', and on the quality and consistency of design. The complexity of system architectures closely related to actual use, and of 'software objects', were not transitory but fundamental in nature. In his later criticism of software engineering, this was a point that he dwelled on in detail: 'Our complexities are arbitrary, because they are fruits of many independent minds acting independently. [...] This complexity is compounded by the necessity to conform to an external environment that is arbitrary, unadaptable, and ever changing.' (Brooks 1987, p. 6) This means, as Brooks has maintained, that software development, system design and system architecture do not adhere to the paradigms of mathematics, or of physics. Overcoming complexity by means of abstract modelling is likely to fail, as is the attempt to attribute heterogeneity and diversity to some 'fundamental unified theory'.

The software architecture concept found many supporters in the software community between the mid-1960s and mid-1970s. I have collected many examples from that period in which the design-based notion of software architecture is applied, not only in IBM circles, but also among many operating systems specialists in the USA and especially in England. It was not until the later 1970s that its meaning gradually shifted from the principle of composition to the principle of decomposition. After 1975, and for almost two decades, the term and concept of software architecture was totally marginalised by the dominant metaphor of software engineering. It was not until the early 1990s that people began to realise again that software development needs architectural design in addition to, or indeed in place of engineering methods. The resurrected concept of software architecture, closely associated rhetorically with Frederick Brooks, quickly became a catch-all term for every unsolved problem and unfulfilled promise of software engineering. Within only a few years, the new profession of 'software architect' and the academic discipline of 'software architecture' came into being *alongside* software engineering. Thus, after three centuries, one arrives at what Ian P. Sharp had demanded as early as 1968 in Garmisch, following the experience he had had with OS/360:

'I think that we have something in addition to software engineering: [...] This is the subject of software architecture. Architecture is different from engineering. [...] I believe that a lot of what we construe as being theory and practice is in fact architecture and engineering; you can have theoretical or practical architects: you can have theoretical or practical engineers. I don't believe for instance that the majority of what Dijkstra does is theory—I believe that in time we will probably refer to the Dijkstra School of Architecture''. [...] Probably a lot of people have experience of seeing good software, an individual piece of software which is good. And if you examine why it is good, you will probably find that the designer, who may or may not have been the implementer as well, fully understood what he wanted to do and he created the shape. Some of the people who can create shape can't implement and the reverse is equally true. The trouble is that in industry, particularly in the large manufacturing empires, little or no regard is being paid to architecture'. (p. 9)

## 4 Summary

In my survey of pre-Garmisch software design methodologies, and my interpretation of the design treatises of Herbert D. Benington, William A. Hosier, Thomas A. Holdiman, Perry E. Rosove, Robert V. Head and James Martin, I wanted to show

- 1. that engineering concepts for software production, process and even life cycle models, as well as methods for managing the software design process were being developed and tested before the Garmisch conference even took place, but were not acknowledged by those taking part at the NATO conferences;

-2. that the transformation of software production was anything but a 'switch' or 'step', that leap-frogging the manufactory phase was not possible, particularly in the software industry, that consequently a manufactory phase in software design did indeed occur, and that this phase developed between 1955 and 1970 in the field of large-scale software systems in the military and semi-governmental contracting sector;

-3. that software developers in the contracting sector likewise hoped for rapid transition to industrial software development along engineering lines, but very soon realised the peculiarities of software as a product, which necessitated a high level of collaboration and communication, also in development processes involving division of labour;

- 4. that dispensing with purely hierarchical control of the software development process in favour of cooperation based on division of labour was reflected in the modelling tools, with the result that phase models were created during that period which clearly differed from the rigid waterfall models of the 1970s and 1980s, and finally,

- 5. that years before the term software engineering was propagated, the term software architecture was already circulating within the community as a name for the discipline that was aimed at expressing not only the structuration of product and process, but also the design aspect of software development, and that was not supplanted by software engineering once and for all until the mid-1970s, only to experience a renaissance in the 1990s, namely in the sense of both superceding and consummating the software engineering concept.

My conclusion thus consists in calling into question any historical development of the discipline along all too straight lines, and in criticising, as Mahoney has also done, the equating of software development and industrial production. Software development turns out, in a manner rather similar to computer architecture, to be a technical design-based discipline in which changing contexts in respect of participants and uses cause continual shifts and renegotiation of design features. A scientification model focused on mathematics and on engineering as applied natural science, like the dominant model at the NATO conferences, has only limited validity in this connection.