ELSEVIER

Journal of Systems Architecture 42 (1996) 591-600

JOURNAL OF SYSTEMS ARCHITECTURE

What is a software process?

Gerhard Chroust

Department of Systems Engineering and Automation, Institute of Systems Sciences, Johannes Kepler University Linz, A-4040 Linz, Austria

Abstract

For an industrial product with more than a minimal complexity the only reasonable way to ensure its quality is to guarantee that it is produced via a quality process. Industrial maturity demonstrates itself in the ability to separate the product from its development process, that is separating the WHAT from the HOW. This separation allows to analyze and certify (!) the development process. Especially for software development the need for certified development processes (ISO 9000, ISO 12207) is pressing. In this presentation we explore some of the basic concepts underlying the notion of a software process and some of their implications.

Keywords: Process; Software engineering; Process model; Quality approaches; Software engineering environment

Quality ... is the degree to which a customer or user perceives that software meets his / her composite needs.

1. Introduction

To ensure the quality of any industrial product, different approaches can be used, cf. Fig. 1. In the sequel we describe them and discuss how far they are applicable to software engineering.

- In scientifically well-understood disciplines we find *construction*: Based on a once-and-for-all established transformation, the product is produced, using the specified requirements as inputs. The algorithm is very often directly implemented by a *tool.* This approach can be found in a small area of the software development: compiling. Once the compiler has been proven correct, it suffices to write the program in the appropriate higher level language. The compiler will produce the corresponding machine language text. For software engineering in general we find that many properties of software products (e.g. user friendliness) cannot be formalized with sufficient rigor to allow the construction approach. In reality user requirements are often not fully understood and thus no basis for a construction process. At the same time they might be too subtle or too voluminous to allow formalization.

PII S1383-7621(96)00044-6

- The construction algorithm is (hopefully) proven correct for all future products. Lacking such a universal tool one can try to perform a *proof* of an individual product: The end product is augmented by a description of the sequence of transformation steps which were used to derive it from some accepted true statements (axioms). The checking of these transformation steps (not their creation) must be fully mechanical, based on a set of rules defining valid transformations. Examples of proofs for software products show that they are usually longer and more error prone than the original program [13,42]. If user requirements change, the construction and the proof process have to be repeated...
- Related to the proof we find the *exhaustive test*, exercising completely the behavior of a product. This is a hopeless attempt with software products. In some disciplines one can use *representative tests*, knowing that due to the continuous nature of physical phenomena the correctness of the representative proves the correctness of all values in its range. Due to the digital nature of software we cannot rely on this assumption: Changing a single bit (the famous missing comma in a space shuttle program) will cause the program to behave in unpredictably different ways (a typical chaotic behavior). As E. Dijkstra pointed out: "Testing can never prove the absence of errors, it can only proof the presence of errors!"
- Other approaches rely on mechanical comparison of two representations of a product and a detection of incompatibilities. Two approaches exist: Two versions of the same product are *created in parallel* and then compared (either at delivery time or during the run-time of the system, i.e. redundancy checking). Alternatively the created product is *reverse* engineered to its initial requirements. These requirements are then compared with the original ones. The problem with this approach lies in the high cost (essentially two independent developments) but is used in highly safety critical applications.
- Given the inapplicability of the more rigorous methods above, we have to rely on some informal quality assurance by humans. We see static (*inspections* and *reviews*) and dynamic verification/validation techniques (*prototyping*). One

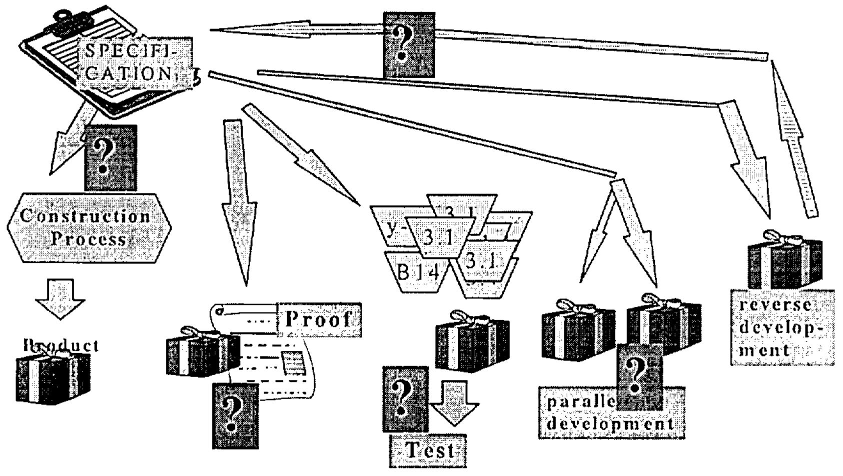


Fig. 1. Quality approaches.

tries to execute some form of the intermediate or final products in order to detect mechanically some unacceptable properties (e.g. collisions) or to enable humans to validate the outcome.

- A completely different approach tries to utilize the observation that the quality of a product is strongly related to the quality of the developer and their level of motivation: they rely on *motivation*, *training* and on accumulation of personal *experience*. This definitely can be brought to bear fruit for software engineering. Due to the youth of this field, however, many seasoned practitioners will not have received a formal education in many of the subjects necessary for proper software development.
- Finally, experience also tells us that proper *organization of work* can reduce the number of errors, often discussed under the topic of *fool proofing* [32]. Part of this approach is also an emphasis on high-quality *documentation*, an approach practiced by bureaucracies throughout the millennia. This approach, too, looks promising for software engineering.

From the above survey we can conclude that looking at the way a product is created (the *development process*) seems to be one of the most promising routes to software quality (Fig. 2), despite the fact that there is no guarantee. This means that one tries to improve *product quality* by improving the *process quality*.

2. Process abstraction

Industrial maturity demonstrates itself in the ability to abstract the development process from the specifics related to the production of the individual product. One of the basic assumptions of CASE-technology is our ability to abstract from the individual product. Such an abstracted process acts as a process model (a template) for future development processes (Fig. 3). This abstraction is not straightforward, since it is necessary to decide which features of an individual process are considered to be specific to an individual product and which features are to be considered relevant for the process model and thus for future processes. Conceptually we separate the WHAT (individual product one wants) from the HOW (this should be done in a general fashion [9]). Obviously, one has to choose a compromise between too general a process description – covering all types of products but lacking any specifics about

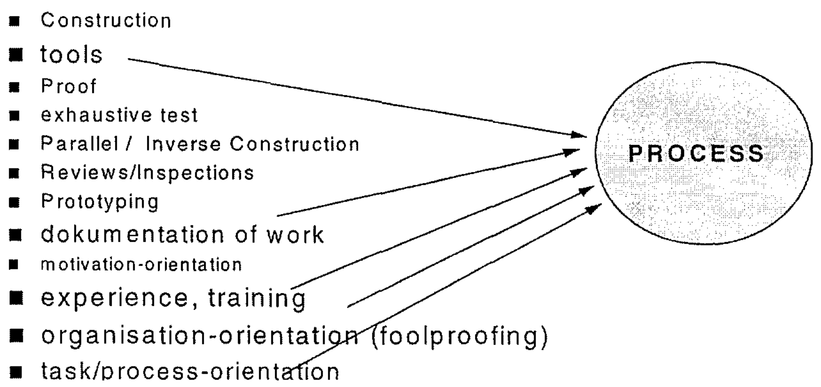


Fig. 2. Approaches to quality.

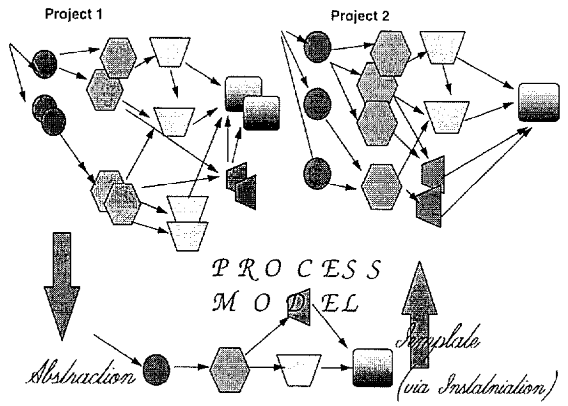


Fig. 3. Abstracting the software process.

an individual product class – and too narrow a description which cannot be applied to a large enough class of products.

Simple processes like building a chair are learnt once and for all (e.g. in apprenticeship). For unknown or complicated processes (e.g. assembling a ready-made chair or cooking an unknown dish, Fig. 4) a written, formalized description is needed. To this aim the desired process(es) are represented in a Process Model (e.g. [5]) and expressed in one of the available Process Model Representation Languages [40].

The notion of process models is actually our daily routine: cooking recipes (Fig. 4), instructions on how to operate the telephone, the video recorder, the car, etc., are essentially process models describing (in more or less detail) a process to be performed

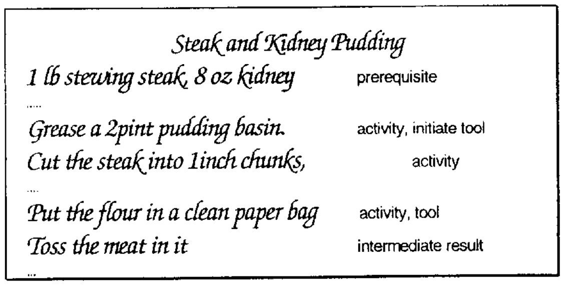
Abstracting the process model offers several advantages:

- The same process model can be applied to different products.
- One can "reason about the software process" [43], about advantages and disadvantages of certain methods and activities.
- One can gradually improve the process based on past experience [26,22,20,37] by adding newly detected process know-how to the process model to be utilized by future projects.

The abstraction of a process is usually based on the experience with already performed processes (Fig. 3) together with some methodological concepts, e.g. as laid down in the ISO 9000 standard [19,36] and ISO 12207.

The abstraction of a process actually has several dimensions:

- *Abstracting from individual products*: Typically the basic software development process to develop a reservation system for a football stadium and one for assigning students to courses will be similar.
- *Abstracting from instances*: Within one project we find different instances of the same type of partial product or activity (e.g. specification of several individual modules of the system). The process model will only contain a description of the types (the "classes") of activities to be performed and of the types (the "classes") of results to be produced (cf. Fig. 3). In the example of Fig. 4 we abstract from several steaks which we might use. Equally we abstract from several C + + modules of the same type.
- *Abstracting from individual engineers:* The process should be essentially the same when performed by different engineers. The process has to be described independently of any personal bias



Steak and Kidney Pudding

1 lb stewing steak, 8 oz kidney	prerequisite
....	
Grease a 2pint pudding basin.	activity, initiate tool
Cut the steak into 1inch chunks,	activity
....	
Put the flour in a clean paper bag	activity, tool
Toss the meat in it	intermediate result
....	

Fig. 4. A recipe.

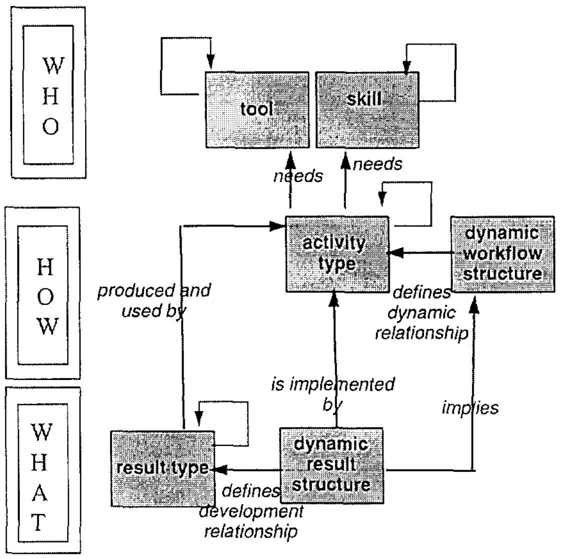


Fig. 5. Cascade model of software processes.

or preference. This is in clear contrast to artisans where the apprentice learns primarily by instruction and observation of the master.

In order to describe a process via a process model, we need several essential components (Fig. 5). We distinguish:

Result types: They describe intermediate and final results of the development process.

Dynamic result structure: It describes in a very general way how results are dependent on one another (e.g. "object module is compilation of source module").

Activity types: They describe the elementary steps of the development process. Activities use results to produce further results. Their description implies the methods to be used.

Dynamic work flow structure: It describes the dynamic relationships of activity types (e.g. "coding must be done after design"). The dynamic work flow structure must consider the constraints already given by the dynamic result structure but may add further sequencing constraints. It defines the strategy of the development process.

Skills: They define the skill(s) a person needs in order to perform the activities of the specified type. This is the key to select personnel for process enactment.

Tools: This defines the tools to be used. Including tool invocation in the process model allows to automate it.

Some components also have a static hierarchy, e.g. tools have subtools, etc. For the purpose of this paper we do not need to discuss these hierarchies any further. We also ignore the problem of generic models and their tailoring [21,5,40].

Note that a model only talks about the "types" of components. When performing an actual project they have to be instantiated (Fig. 3). An activity instance is the executable result of instantiation, cf. [21,40,41]. There may be several instances for a given activity type. The number of instances usually has to be determined at run-time.

Only the written/stored representation of the desired process can be the basis for further improvement of the process. It has to be pointed out that the actual codification of the process is hard, time-consuming, error-prone and full of controversy [28]. Secondly, the perception of the process can be quite different [27]. One has to distinguish

- *The perceived process*: What you think you do.
- *The actual process*: What you actually do.
- *The official process*: What you are supposed to do.
- *The initial process*: What you define initially.
- *The target process*: What you want to do in the future.

The perceived, the actual and the official process are the basis for an initial process model. Based on observation, experience and theoretical considerations this should gradually be transformed into the target process. The target model itself is a moving target, since continuous improvements are expected and actually necessary. In order to become world class, the process model must include the relevant accompanying processes like quality management, project management [6,7], etc. For providing flexibility (e.g. different quality assurance paradigm) this implies that we create partial process models and combine them in a controlled way [11].

At this point one has to point out that process models go far beyond a life cycle model. Life cycle models primarily define several phases of the development process and the results to be passed between them without concern about the detailed steps within each phase. They contain a limited number of phases, usually between 4 and 10; the most well-know being the waterfall model [4]. Process models, on the other hand, give a very detailed description of the intended process, giving details of the intermediate products to be processed and their relationship to the individual activities. Typically IBM's process model ADPS had more than 180 activity types and some 200 result types [7]. Due to its greater level of detail a process model implies a certain development method. Additionally, a process model also describes methods to be applied for the various activity types and often it even defines the tools to be used.

3. Process enactment / software engineering environments

The size of a process model results in a rather voluminous written description. The danger is that a purely printed document will not be consulted during development and, thus, the adherence can not be guaranteed or enforced. A solution is the use of a formalized and machine readable process model which is on-line and interpreted ("enacted") by a computer program. One usually calls such an enactment mechanism a *Process Engine* [43]. One of its

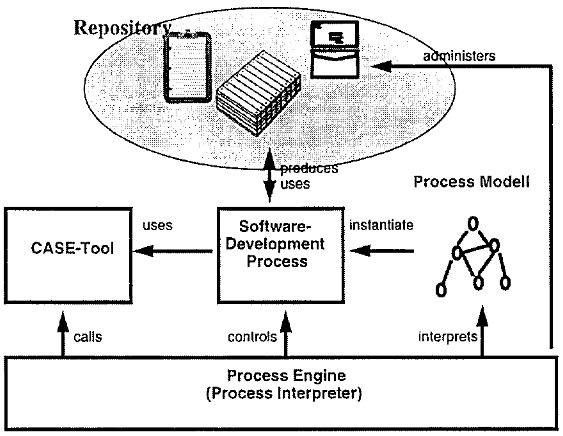


Fig. 6. Basic functions of a software engineering environment.

tasks is to guide the users via the process model through the development process [10,21].

Nowadays the Process Engine provides numerous functions to the users (Fig. 6):

- Interpreting the process model, (gradually) instantiating the process and then controlling its enactment.
- Since the process model expresses in detail the methods to be used it can also define the tools to be used. Tools are the actual workhorse of development and are essential for providing productivity and quality. The process engine can call the tools on behalf of the user.
- Obviously the availability of the description of the results related to the activities allows the Process Engine also to administer the various intermediate and final results on behalf of the user.

An environment providing a process engine, a process model and the necessary tools is usually referred to as an (Integrated) *Software Engineering Environment*, but many other names exist, too [25,29,35].

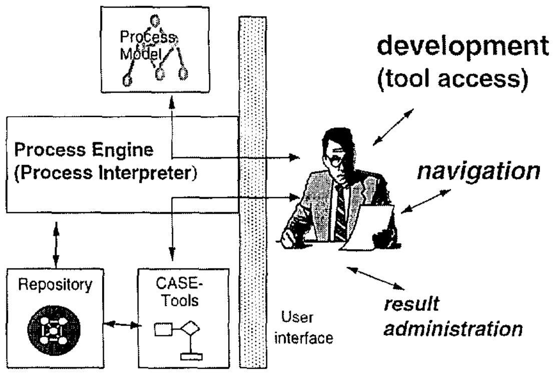


Fig. 7. Components of a software engineering environment.

From a user perspective a software engineering environment provides a structure as shown in Fig. 7. One recognizes

- the process model, describing the intended process;
- the process engine guiding and controlling the development process;
- the CASE-tools for performing the individual development tasks;
- the Repository, containing all intermediate and final results and their relationships;
- the user interface providing appropriate representations to the user.

The user performs three different tasks:

- navigation: deciding what to do next, based on the process model, on the status of results and other activities. The system can give some help by providing priorities, etc.;
- development work, performing the creative part of software engineering, utilizing the existing CASE tools; and
- (to a lesser extent) administration of the results. This is mainly done by the process engine.

4. Existing process models

A number of process models has been widely published and a few are in wide use. Some widely used models became to some extent de-facto standards in some countries or industry segments. In the sequel we will list some of the well-known ones and provide references for further details:

- *ADPS* from IBM [8,7] was a software engineering environment with a very complex process model (some 180 activity types), describing development, quality assurance and project management. Its primary target platform was AD/CYCLE [31]. It was fully operational on a process engine.
- *ESA-Model* [17], the model of the European Space Agency consists of a written set of documents describing in textual form the development process. In order to enact it via a process engine, considerable work would have to go into the definition of activity types.
- *EUROMETHOD* [18] is a framework to manage the technical and organizational aspects of a contractual customer-supplier relationship, during the life cycle of an information system.
- *HERMES* is the model prescribed by the Swiss government [1]. It is largely a variant of the V-Model (see below).
- *MAESTROII* from Softlab [30,44] is a repository-centered software engineering environment. Its process model is a variant of the SETEC model (see below).
- *MERISE* [12,34] is a process model with a long standing tradition in France. It is widely accepted there.
- *SETEC* [24,23,38,39] has as basic philosophy that the users should keep their existing methods. SETEC is used to bind them together into an appropriate life cycle frame work.
- *SSADM* [15,14] is the de-facto standard in the United Kingdom. The process model is very de-

tailed, mostly in form of data flow-type diagrams with associated text. The activities concerned with development, quality assurance and project management are intermixed in one model.

- *V-Model* [5,6] is the standard of the German government and quickly becoming an industry-wide standard. Besides a model for actual software development the standard describes also in detail separate, but interlinked submodels for quality assurance, project management and configuration control. Its translation into an enactable process model is under way.

5. Process models and process improvement

The motive behind introducing process models is the observation that the quality of the development process strongly and positively influences the quality of the product. This is also the incentive to *improve* the process itself.

In the last decade several concepts of measuring the quality of the software development process have been publicized (CMM [33], ISO 9000 [36], BOOTSTRAP [3]). There are also several international methods to measure potential process improvements (e.g. BOOTSTRAP [20], SPICE [37]). Several papers in this special issue discuss this topic.

6. Trends and outlook

The industry, but also the European Community, recognize the fact that an orderly, best-practice based

Fig. 8. ESPITI-logo.

Fig. 9. Austria's ESPITI logo.

process model together with the provision for its observance are a key to a successful software industry. The European Software Process Improvement Training Initiative (ESPITI) honors this fact and tries to disseminate this idea in Europe (Fig. 8 and Fig. 9). It is carried out in all countries of the European community [2,16].

Numerous advantages can be derived from a formal description of the process model. Some of them are [9,28]:

- The way the project should be performed is laid down before the project begins.
- The project can be planned easier.
- The methodology can be taught more effectively.
- The process becomes transparent.
- The process can be improved – a necessary prerequisite for full maturity of software development [33].
- Usually the resulting products have a higher quality.
- The process can be certified, thus giving more creditability to the company.
- Past projects can be better audited, etc.

Summing it all up, it shows that the concept of a Process Model is a central point of control for software production. It will gradually become a prerequisite for mastering the challenges of the software industry in the near and distant future.

References

[1] Bundesamt für Informatik (ed.), HERMES – Führung und Abwicklung von Informatikprojekten, Bundesamt für Informatik, Schweizerische Bundesverwaltung, Bern, Ausgabe 1995.

[2] I. Bey, ESPITI – A European challenge, to appear in *Journal of Systems Architecture* (1996) (in this issue).

[3] A. Bicego, Process maturity and certification, to appear in *Journal of Systems Architecture* (1996) (in this issue).

[4] B.W. Boehm, Software life cycle factors, in: C.R. Vick and C.C. Ramamoorthy (eds.), *Handbook of Software Engineering* (Van Nostrand, New York, 1984) 494–518.

[5] A.P. Bröhl and W. Dröschel (eds.), *Das Vorgehensmodell in der Anwendungsentwicklung – Standard und Leitfaden* (Oldenbourg, München, 1992)

[6] A.P. Bröhl and W. Dröschel (eds.), *Das V-Modell – Der Standard für die Softwareentwicklung mit Praxisleitfaden* (Oldenbourg, 1993)

[7] G. Chroust, Application development project support (ADPS) – An environment for industrial application development, *ACM Software Engineering Notes* 14(5) (1989) 83–104.

[8] G. Chroust, Duplicate instances of elements of a software process model, in: C. Tully (ed.), *Representating and Enacting the Software Process – Proc. 4th Int. Software Process Workshop*, May 1988. *ACM Software Engineering Notes* 14(5) (1989) 61–64.

[9] G. Chroust, *Modelle der Software-Entwicklung Aufbau und Interpretation von Vorgehensmodellen* (Oldenbourg Verlag, 1992).

[10] G. Chroust, Development environments: A multi-level enactment scenario, in: J. Rosenblit (ed.), *AIS-93, 4th Conference on AI, Simulation, and Planning in High Autonomy Systems*, Tuscon Sept 1993, IEEE C/S Press, 2–13

[11] G. Chroust, Partial process models, in: M.M. Tanik, W. Rossak and D.E. Cooke (eds.), *Software Systems in Engineering* (The American Society of Mechanical Engineers, New York, 1994) 197–202.

[12] M. Deltl, MERISE – Theorie und Implementierung einer Analyse- und Entwurfsmethode, Diplomarbeit, Techn. Univ. Wien, 1989.

[13] R.A. de Millo, R.J. Lipton ans A. Perlis, Social processes and proofs of theorems and programs, *Communications of the ACM* 22(5) (1979) 271–280.

[14] E. Downs, P. Clare and I. Coe, *Structured Systems Analysis and Design Method* (Prentice Hall, Englewood Cliffs, 1988).

[15] R. Duschl and N.C. Hopkins, *SSADM and GRAPES – Two Complimentary Major European Methodologies for Information Systems Engineering* (Springer-Verlag, 1992).

[16] K. Eakin, Implementing ESPITI on a European scale, to appear in *Journal of Systems Architecture* (1996) (in this issue).

[17] ESA (ed.), ESA software engineering standards (Issue 2), European Space Agency ESA-PSS-05-0, Paris, 1991.

[18] European Software Institute (ed.), EUROMETHOD, WWW-Information on http://www.esi.es/Euromethod/.

[19] V. Haase, R. Messnarz, G. Koch, H.J. Kugler and P. Decrinis, Bootstrap: Fine-tuning process assessment, *IEEE Software* 11(4) (1994) 25–35.

[20] V. Haase, Concepts for the assessment of software process quality, to appear in *Journal of Systems Architecture* (1996) (in this issue).

[21] S. Hardt, Sprachunabhängige dynamische Ausführung von Vorgehensmodellen, Dissertation, Kepler University Linz, 1994.

[22] J. Herbsleb, A. Carleton, J. Rozum, J. Siegel and D. Zubrow, Benefits of CMM-based software process improvement: Initial results, Software Engineering Inst., Pittsburgh, PA, USA, Rep. No. CMU/SEI-94-TR-13, Aug. 1994.

[23] R. Herrmann, MAESTRO – Die integrierte Software-Produktionsumgebung von Softlab, in: T. Gutzwiller and H. Österle (eds.), *Anleitung zu einer praxisorientierten Software-Entwicklungsumgebung, Band 2* (AIT Verlag, München, 1988) 63–78.

[24] W. Hesse, S/E/TEC – The software engineering environment of SOFTLAB. in: V.H. Haase and R. Hammer (eds.), *Softwaretechnologie für die Industrie (EWICS 84)* (Oldenbourg, Wien, München, 1984) 219–245.

[25] Huenke H. (ed.), Software engineering environments, *Proceedings* Lahnstein, BRD, 1980 (North-Holland, 1981).

[26] W.S. Humphrey, *Managing the Software Process* (Addison-Wesley, Reading Mass., 1989).

[27] W.S. Humphrey, *A Discipline for Software Engineering, SEI Series in SW Engineering* (Addison Wesley, 1995).

[28] J. Lohmeijer and B.T. Smith, AIMS/38 – A working first generation IPSE for the IBM system 38, in: P. Brereton (ed.), *Software Engineering Environments* (Ellis Horwood Ltd., J. Wiley 1988) 147–157.

[29] J. McDermid (ed.), *Integrated Project Support Environments* (P. Peregrinus Ltd., London, 1985).

[30] G. Merbeth, MAESTRO-II – Das Integrierte CASE-System von Softlab. in: H. Balzert (ed.), *CASE – Systeme und Werkzeuge* (Vierter Auflage, B-I Wissenschaftsverlag, 1992) 215–232.

[31] V.J. Mercurio, B.F. Meyers, A.M. Nisbet and G. Radin, AD/Cycle strategy and architecture, *IBM System Journal* 29(2) (1990) 170–188.

[32] T. Nakajo and H. Kume, The principles of foolproofing and their application in manufacturing, *Reports of Statistical Application Research JUSE* 32(2) (1985) 10–29.

[33] M.C. Paulk et al., Capability maturity model, Version 1.1., *IEEE Software* 10 (1993) 18–27.

[34] B.P.T. Quang, MERISE: A French methodology for information systems analysis and design, *Journal of System Management* (1986) 21–24.

[35] W. Schaefer (ed.), Software process technology, *4th European Workshop EWSPT'95 Noordwijkerhout*, Springer Lecture Notes No. 913, Springer-Verlag, Berlin–Heidelberg, 1995.

[36] C.H. Schmauch, *ISO 9000 for Software Developers* (ASQC Quality Press, Milwaukee, Wisconsin, 1994).

[37] J.M. Simon, SPICE – Improving the software process, to appear in *Journal of Systems Architecture* (1996) (in this issue).

[38] Softlab (ed.), S/E/TEC: die Software Engineering Technologie von Softlab, Prospekt, 1982.

[39] Softlab (ed.), V/TEC: Softlabs Technologie Angebot für Anwender mit IBM Umgebung, Prospekt Dec., 1982.

[40] G. Starke and S. Hardt, Process models: The future of software engineering, *Proceedings of the 3rd International Conference of Young Computer Scientists ICYCS*, Juli 1993, Beijing, China (Tsinghua University Press, 9/58–9/65).

[41] G. Starke, Sprachen zur Software-Prozeßmodellierung, Dissertation, Kepler Univ. Linz, 1994, Verlag Shaker, Aachen 1994.

[42] A.S. Tanenbaum, In defense of program testing or correctness proofs considered harmful, *SIGPLAN Notices* 11(5) (1976) 64–8.

[43] J.C. Wileden and M. Dowson (eds.), International workshop on the software process and software environments, *Software Eng. Notes* 11(4) (1986) 1–74.

[44] E. Yourdon, Softlab's maestro, *American Programmer* 3(3) (1990).

Gerhard Ghroust joined the IBM Laboratory Vienna in 1966 and worked until 1968 on the Formal Definition of PL/I, and during 1969–1972 on computer construction. From 1972 to 1976 he was assistant to the Laboratory Director, Heinz Zemanek, handling the cooperation with academic institutions. From 1976 to 1982 he worked on the PL/I Compiler for the IBM 8100. From 1983 to 1990 he was member of the development team for ADPS (Application Development Project Support), mainly responsible for defining the Process Model and from 1990 to 1991 he was responsible for product support of ADPS. Since 1992 he is a full Professor of Systems Engineering at the Kepler University Linz, Austria. Dr. Chroust holds a Diplom-Ingenieur and a Ph.D. from the Technical University of Vienna and an M.S. from the University of Pennsylvania. He is the author of a book on "Microprogramming and Computer Architecture", and one on "Models for Software Development", editor of six more books, and has published numerous articles on microprogramming and software development. His current research and teaching interests are focused on the description of socio-technical processes and on their enactment via engineering environments in areas like software engineering, flexible manufacturing and office automation.