Software Processes:
a Retrospective and a Path to the Future[1]

Gianpaolo Cugola and Carlo Ghezzi

[cugola, ghezzi]@elet.polimi.it
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Piazza Leonardo da Vinci,.32
20133 Milano - Italy

Abstract

Software engineering focuses on producing quality software products through quality processes. The attention to processes dates back to the early 70's, when software engineers realized that the desired qualities (such as reliability, efficiency, evolvability, ease of use, etc.) could only be injected in the products by following a disciplined flow of activities. Such a discipline would also make the production process more predictable and economical. Most of the software process work, however, remained in an informal stage until the late 80's. From then on, the software process was recognized by researchers as a specific subject that deserved special attention and dedicated scientific investigation, the goal being to understand its foundations, develop useful models, identify methods, provide tool support, and help manage its progress.

This paper will try to characterize the main approaches to software processes that were followed historically by software engineering, to identify the strengths and weaknesses, the motivations and the misconceptions that lead to the continuous evolution of the field. This will lead us to an understanding of where we are now and will be the basis for a discussion of a research agenda for the future.

Keywords and phrases:
Software process, software quality, process-centered software engineering environment, computer-supported cooperative work, workflow management systems, inconsistency, deviation.

1 Introduction

The ultimate goal of the theories, techniques, and methods that underlie any engineering field is to help engineers in the production of quality products in an economic and timely fashion. Usually, this is obtained by providing a careful distinction between products and processes [36]. The product is *what* is visible to the customers, and thus it is all that matters in the end. The process is *how* this goal can be achieved. "What" and "how", however, are two

---

[1] This paper is an expanded version of the keynote presentation given by Carlo Ghezzi at the 5th International Conference on Software Process (Lisle, IL, 14-17 June 1998).

sides of the same coin. It is through the process, in fact, that engineers inject quality into their products, they can reduce time to market, they can control (and, possibly, reduce) production costs.

These general considerations are valid for every engineering field. However, they are especially relevant in the case of software engineering. The very nature of software, in fact, makes product quality difficult to achieve, and assess, unless a suitable process is in place. It is not surprising, therefore, that the focus of software engineering research and practices on software processes can be traced back to the early stages of the field. The purpose of this paper is to look back critically to what has been done, to evaluate where we are, what has been achieved so far, and what remains to be done in the future.

The paper is organized as follows. Section 2 explores further the concepts of quality, product, and process in the case of software. Section 3 reviews the historical evolution of software processes from the sixties to the eighties. Most of the modern research on software processes was started in the late 1980s: we will review its main achievements in Section 4. Section 5 will provide a critical assessment of the results produced by this research. In particular, it will try to identify where the major weaknesses are and will outline a possible research agenda for future developments. Finally, Section 6 draws some conclusions.

2 Why are software processes relevant

In the past decade, there has been an increasing concern for quality in most industrial sectors. In addition, there has been an increasing awareness of the central importance of the production processes. Processes are important because industry cares about their intrinsic qualities, such as uniformity of behaviors across different projects and productivity, to improve time to market and reduce production costs. But they are also important because experience has shown that processes have a profound influence on the quality of products; i.e., by controlling processes we can achieve a better control of the required qualities of products.

*Product requirements*

process

product
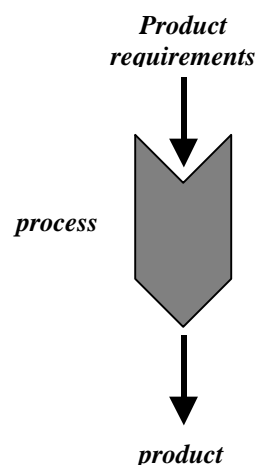
Figure 1: The process as a black box (1)

This is especially true in software factories due to the intrinsic nature of software. If an explicit process is in place, software development proceeds in a systematic and orderly fashion. This prevents errors from being introduced in the product and provides means for controlling the quality of what is being developed. Figure 1 provides an intuitive view to reinforce this point. If no explicit notion of "process" is in place, product development can be considered as a black pipe where the only visible flows are at pipe's input and output ends. At the input side, the flow into the pipe represents product requirements. At the output end of the pipe, hopefully, the desired product is delivered. Unfortunately, in many practical cases, the

product appears at the output side months or years since the development started (and thus after much money has been invested). When the product appears at the output side of the process pipe, it is often too late and too expensive to care about quality. It is therefore necessary that the concern for quality permeates the whole process; it cannot be delayed to the end of development.

There are some deep reasons that make the approach of Figure 1 even worse for software than for other more traditional kinds of artifacts. The first major difficulty has to do with requirements elicitation, as shown in the view of the software development process provided in Figure 2. According to such a view, a product development starts (in most cases) with some informal requirements that originate in the customer's business world. The problem is that, in many cases, the customer does not know exactly what he or she wants. The customer has a perception of his or her problems, but is unable to translate this perception into precise requirements. As a consequence, the input requirements to the process are likely to be very informal, fuzzy, largely incomplete, maybe contradictory, maybe even not reflecting the user's real needs [47]. If the development process is structured as a black box, there is no visibility of what is going on as long as the development process progresses. Eventually, when the product is delivered, it is very likely that the product does not match the expectations of the customer. Although software is easier to modify than other traditional kinds of artifacts, the cost of post-development modifications is very high and its effectiveness is often very low.
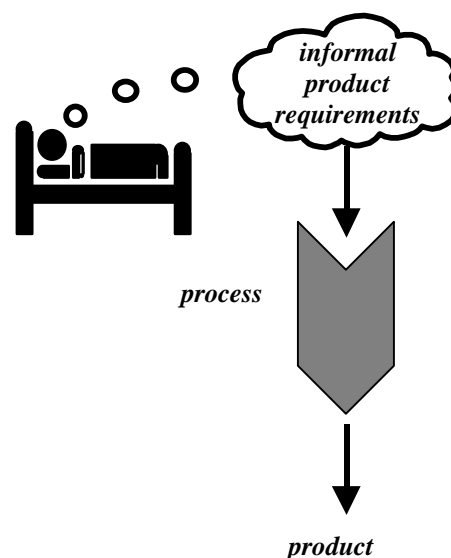
Figure 2: The process as a black box (2)

Thus, it is extremely risky to base all design decisions on the assumption that the initial requirements acquired by the software engineer faithfully capture the customer's expectations. To reduce the risks, it is necessary to open the black box. One must define a suitable process that provides visibility of what is being developed. By viewing inside the black box, one may hope to be able to validate what is being developed against the customer's expectations. The process may thus provide continuous feedback to the developers[2]. This may reduce the time between making a decision and discovering that the decision was actually wrong, thus reducing the costs needed to develop an acceptable product.

Another difficulty has to do with the inevitable tendency of software requirements to change during the process. As we mentioned, often customers do not know exactly what they want; and even if they do, their requirements keep changing during the process. Another common case that results in rapidly changing requirements is typical of the development of

---

[2] In the sequel, the terms software engineers and software developer will be used interchangeably.

new kinds of products, as for example happens now for Internet applications. In such cases, the initial requirements are only partially known and there are no customers out there to provide a precise list of features that the application should provide. New demands arise and new potential customers appear as prototypes are delivered and feedback is provided from initial users to improve the product.

These concepts suggest an alternative process scheme, which is described pictorially in Figure 3. As suggested by Figure 3, a transparent production process allows the customer to understand what is going on inside the process. In particular, it allows the customer to observe the artifacts produced during the process. Such artifacts (e.g., use cases, intermediate prototypes, preliminary and incomplete versions, design documentation, test case definition, etc.) may be used to provide some form of validation of the current process. As a result of the validation, it is possible either to decide to proceed to the next step, or to re-iterate the previous step.
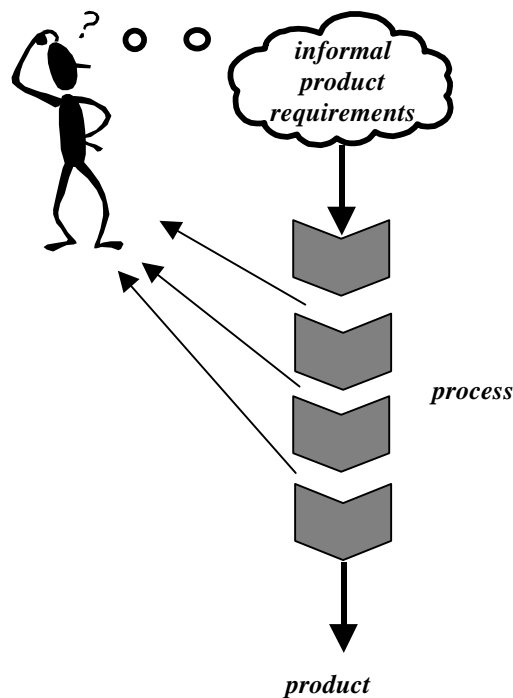
Figure 3: A transparent process

Another fundamental reason that makes the black-box process of Figure 1 unacceptable for software is that one cannot expect to assess the quality of the product by simply looking at the product itself. This may differ from the case of other kinds of artifacts. For example, in the case of a bridge, one can certify that the bridge can sustain certain load conditions by physically applying those load conditions. This ensures that the bridge has an acceptable behavior for all other load conditions that are less than the one used for certification. Thus a single test verifies correctness of infinitely many cases. Conversely, it is well known in the case of software that a successful test execution, in general, does not tell us much about other possible executions [58]. Testing and all other kinds of practical analysis techniques have intrinsic weaknesses. They cannot assure the absence of defects in software. To achieve more confidence in the correctness of software, it is recommended that the process be structured in a way that makes development systematic and therefore less error prone. By combining prevention and continuous verification and validation, an explicitly and clearly defined process may improve the software engineer's confidence in software qualities.

3 A retrospective

Historically, as the problems outlined in Section 2 became better understood, an increasing attention was dedicated to software processes. Hereafter, we will try to understand the main approaches that were followed, in order to identify their benefits and weaknesses. Our presentation is structured as a kind of idealized historical chronicle through which we will trace the evolution of the field. In the real world, however, the boundaries among the different approaches were fuzzier. Each approach existed in (and continues to exist) in many different forms and variations. Each went through many changes that were suggested by the experience gained in its practical application.

To stress the different principles, upon which the various approaches are based, we will present the historical evolution as a sequence of *myths*. The term myth was chosen because often they were presented as "the" solution to the software process problem. Their acceptance was based more on an act of faith than on objective—let alone, quantitative—measurements of data. Our overview of myths is constrained by obvious space limitations. This explains why, in order to emphasize the conceptual differences among myths, we were forced to provide some oversimplifications.

3.1 Myth 1: The software lifecycle

The initial solution proposed in the sixties, and subsequently elaborated in many variations, is the concept of the software lifecycle [67]. The lifecycle defines the standard "life" of a product, from its initial conception until deployment and maintenance. This means that the development process is decomposed into a predefined sequence of *phases*, each of which is structured as a set of *activities*. Each phase receives inputs from the previous phase and provides output to the following phase. The chosen lifecycle model standardizes both the decomposition of the process into phases and activities, and the artifacts (documents) that flow from each phase to the next.

The *waterfall* lifecycle [67] is perhaps the most widely known and most idealized form of a lifecycle. The basic principle underlying the waterfall lifecycle is that the process should proceed in a linear fashion, without cycles. The motivation is that orderly developments are linear. Recycling should be avoided as much as possible since it allows the developers to undo what they did before, in an unconstrained manner. Recycling encourages the sloppy attitude of doing things without carefully thinking in advance. Moreover, the lack of linearity makes the process difficult to predict and control. Since one can always go back to previous phases, it is hard to know how far one is in the process and it may be impossible to control if the process progresses according to the budget and to the schedule.

feasibility study
requirements
design
implementation
system test
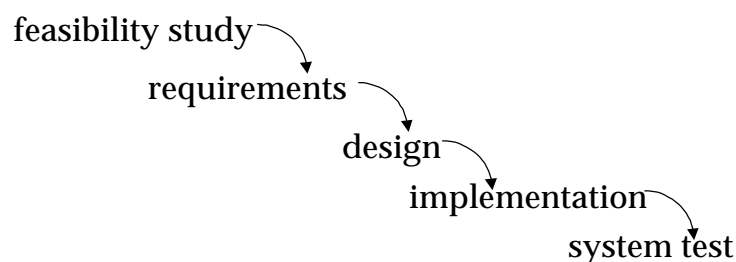
Figure 4: A sample waterfall lifecycle

A typical waterfall lifecycle is shown in Figure 4. It is easy to realize that the model is inspired by the traditional manufacturing models, where products are developed through a fixed sequence of well-defined (and often automated) processing steps. All that matters is that production proceeds from a step to the next as scheduled in the workplan.

Many software companies adopted (and many still adopt) a waterfall lifecycle as part of their standard mode of operation. The experience, however, has shown that strict versions of this model work rarely for software, and in some cases they do not work at all [37]. As we

already observed in the previous section, since requirements are hardly known beforehand, redoing becomes a necessity. If the lifecycle is not structured in a way that intermediate artifacts can be used to provide effective feedback on what is being developed, eventually the need for changes manifests itself as high maintenance costs. Moreover, unfortunately, waterfall lifecycle models do not help much with maintenance. Maintenance is seen as a nuisance, an undesirable effect that occurs after development is completed. This, however, is a serious mistake. Maintenance is an unavoidable phase of the process, and its costs often exceed development costs [37]. Maintenance activities include defect removal (*corrective maintenance*), adaptation to environment changes (*adaptive maintenance*), and evolution to improve existing functionalities, or add new functionalities to meet new market conditions, new uses, new desires, new ideas about how to do business (*perfective maintenance*). This last category accounts for the largest portion of maintenance costs. Its relevance indicates quite clearly that in practice one cannot hope to develop the "complete" and "stable" requirements of an application before design and implementation start. Customers often have limited knowledge of the requirements when the project starts. Initial requirements are almost inevitably incomplete and imprecise; sometimes, they are even wrong. A strict waterfall lifecycle, on the other hand, tries to organize software development as a linear sequence of steps: it assumes that all requirements are acquired before proceeding to design, that design should be completed before one proceeds to implementation, and so on. Since often this cannot be achieved, all the inevitable changes manifest themselves as (unanticipated) maintenance. Furthermore, many waterfall lifecycles tried to provide a rigid decomposition into a standard development activities. In practice, however, software developers found it hard to follow such a fixed, predefined, and standard process model. Inevitably, software production contains creative design steps; unlike manufacturing, it cannot be completely predefined. Thus there is no unique, universal software lifecycle that can be used in any organization and for any product. Software development requires flexible and adaptive lifecycles.

In conclusion, lifecycle models, such as the waterfall model and its many variants, are based on the assumption that software processes can be standardized once for all. Furthermore, they require that complete knowledge on a project is available before the project starts. Since these assumptions are seldom valid, the myth that a standard lifecycle model can be defined to guide software development failed. As Parnas pointed out [61], however, the waterfall lifecycle defines an idealized process which can describe the rationale of a project *a posteriori*. Even if the process that developers followed in practice was not linear, it can be described afterwards as an orderly sequence of well-defined phases with well-defined interfaces. This makes the documentation well structured and clearly understandable.

3.2 Myth 2: Methodologies

Another approach that became popular in the 60s and 70s was the definition of "development methodologies", intended to provide expert guidance and wisdom in the development process. Well-known examples are JSP [45], which later evolved into JSD [46], and SA/SD (Structured Analysis/Structured Design) [77,28]. For example, SA/SD popularized the use of data-flow diagrams as a specification method, and provided guidance in deriving the structure of an application from such kinds of semi-formal specifications.

These methodologies were all based on a number of common points. First, they provided a number of notations to be used in specification. Second, they provided a number of detailed guidelines and recommendations on how such notations could be used and how to move through the development process from the initial phases down to coding.

Methodologies did not come out of some magic, but rather they were the result of the experience gained in previous successful developments. They can be viewed as the encoding of the distillation of the best experiences gained in the development of software.

In spite of the enthusiastic claims provided by methodologists, mostly through professional seminars and textbooks, the proposed methodologies suffered from many problems, among which:

- Methodologies were applied in contexts other than those in which the experience was initially gained. For example, a methodology developed in the field of business information systems was then applied to developing a real-time embedded application. In many cases, there was no empirical evidence that application to other contexts would work, and in fact this lead to many failures.
- Developers took methodologies as recipes, rather than general guidelines. The negative side effect was that developers felt less responsible and tended to focus more on the external frills of the methodology rather than on its substance.
- Most methodologies required a lot of paperwork, and no tools were available to automate the clerical parts. As a consequence, they were expensive to use properly;
- Methodologies were based on informal notations. The lack of precise semantics made descriptions ambiguous and difficult to check for correctness or consistency.
- Their goal was to provide predictable results given the same premises. Unfortunately, most of them did not achieve this goal [14].

3.3 Myth 3: Formal development

In contrast with the informality of the previous approaches, a new stream of research started in the late 60s, which tried to develop the foundations for an approach to software development based on mathematics. The underlying basic assumption was that programs are mathematical entities that can be formally specified, proven correct, and even developed correctly by means of calculi for program derivation that are guaranteed to transform a specification into a correct implementation, via well-defined refinements [29,76,26,38,10].

Indeed, this approach provided a new impetus for research in computer science and led to a much better understanding of the foundations of software development. It helped understand precisely what a specification is and what it means that an implementation conforms to its specification. It led to a better understanding of the limitations of traditional approaches as far as program correctness was concerned. The intrinsic limitations of testing, as a way to certify software, became more explicit.

As a general solution of the software problem, however, this approach failed. The first problem was that it assumed that a formal specification of software functionality is available as a starting point. We already mentioned that this assumption does not hold in general. The second problem had to do with scalability. The methods proposed to develop correct programs worked fine in the small, but could not scale up to the development of complex systems. It became soon evident that mastering individual programs does not imply that one can also deal with large systems. The third problem had to do with non-functional requirements. Formal methodologies do not take them into consideration, although it is well known how non-functional requirements represent a fundamental part in any complex application.

Because of these considerations, most of the proposed methods could be viewed more as methods for systematic programming, rather than methods for software development. But it is well known that there is much more to software development than just programming. Many researchers who work in formal methods now acknowledge these criticisms. They now agree that formality is just one tool that software engineers should master. They also agree that it is not important just to provide a formal notation, but it is equally important to show how and where it can be used, and have a method that guides in its use.

3.4 Myth 4: Automation

The 70s saw the rapid growth of software platforms that provided a rich set of tools for software development. UNIX and the UNIX workbench became the popular environment for the development of C applications [53]. The underlying idea was that simple, neutral, and small-grain tools could be flexibly combined to achieve power and complexity. The main goal of software development became "automation". Since software was the major enabling

technology behind automation in most industrial sectors, why would not software be able to support automation of software production?

Other interesting approaches were taken to support automation. One was the study and implementation of Software Development Environments (SDEs) [62]. Language-based environments were a popular research topic [39,66]. Later, the efforts around APSEs—environments supporting Ada program developments—gave new impetus and more generality to this field of research [73]. PCTE was probably the richest and latest result of this research stream [72].

A similar trend could be observed in the business information systems field. Fourth-generation languages and environments, which became popular in the 80s, are based on the assumption that several software development tasks can be automated by a suitable SDE. Report generation and input forms can be automatically generated from the description of database entities; their form can be manipulated, if necessary, by direct screen manipulation, without writing code. Code is automatically generated by the SDE.

As more automated functions were provided by SDE, it became clear what the limits of automation really are. Simple steps can be automated, and this relieves software engineers from the painful details of certain programming tasks. But, as we observed, there is much more to software development than programming. The need for mastering complexity is still there. The need for integrating technical and equally important non-technical aspects is still there. Requirements acquisition and specification cannot be fully automated. Critical design decisions, which require analyzing the trade-offs among several possible alternatives, cannot be fully automated.

3.5 Myth 5: Management and improvement

In the 80s, industry became more and more concerned with quality. Japanese factories became known for their attention to processes that would guarantee quality products. International quality standards, like the ISO9000 series [44], dictated ways to certify organizations, and certification was increasingly perceived as an indirect assurance that an organization would deliver quality products. Quality standards are heavily reliant on company-wide management procedures that ensure a predictable and orderly flow of activities.

International standards do not support organizations in understanding how good they are in their business and how they can improve their process. This is exactly the purpose of the Capability Maturity Model (CMM), developed by the SEI, which became extremely popular in the late 80s [43]. The CMM defines a series of maturity levels, which characterize a software development organization. It further defines the recommended practices that would allow an organization to progress along the maturity scale, to achieve higher maturity.

Industry standards and the CMM have been important because they stressed the managerial aspects of software development. But this is of course also their limit. It may be observed that in some cases they did not result in better organization, but rather in increased bureaucracy. The question that often arises is whether they are mostly oriented towards large and highly structured organizations, and how good they are for highly flexible, dynamic, market-driven, innovative organizations. Finally, the assumption that certified organizations perform well and produce high-quality products has never been proved and should be challenged by sound empirical assessment.

3.6 Myth 6: Process programming

In 1987, Leon Osterweil gave a keynote speech at the 9th International Conference on Software Engineering (ICSE-9) whose title was "Software processes are software too" [59]. This work can be considered as a pivotal paper in modern research on software processes. Although some of the ideas presented by Osterweil were introduced and discussed in several previous research workshops, the ICSE-9 paper raised much attention and gave new impetus to software process research.

Osterweil started from the observation that all organizations are different. They differ in culture, people skills, products delivered, commercial and development strategies. Even within the same organization the different projects present huge variations. They aim at developing different products having specific characteristics. As a consequence, there is no unique, ready-made software development process. The process must be defined based on the problem to be solved; it must be tailored to the specific development project and should take into account all the particularities of the organization and product being developed. Also, the environments that support software development should be tailored to the specific development process. To satisfy these goals, Osterweil concluded, we need languages to describe software development processes. The resulting *process models* should be *verifiable*, to check if they meet the organization's goals. The models should be also *executable*, to provide runtime process support. All that demands for new software development environments capable of supporting the development, documentation, analysis, execution, and evolution of process models. Process model development is like software development (in this sense "software processes are software too"). It requires specific skills; it goes through different phases (from requirements to design to implementation to verification to evolution); it requires specific tools to be supported.

Osterweil's controversial viewpoint stimulated an active research area, which focused on the development of *Process-centered Software Engineering Environments* (PSEEs) [2,35]. PSEEs can be considered as a new generation of SDEs that can be tailored to each specific software project, to provide it with the best possible automated support. PSEEs support the engineering process used to conceive, design, develop, and maintain a software product through an explicit process model. Such a model is described by means of a suitable *Process Modeling Language* (PML) [34]. The model specifies how people should interact and work, and also how and when the automated tools used in the process should be used and/or automatically activated. A *process engine* can then *enact* (i.e., execute) the process model, which is part of the PSEE. The process engine guides and supports people in performing the different activities that are part of the process, and automates the execution of the activities that do not require user intervention.

The process engine is the core of a PSEE. It is composed of three main logical components:

- An *interpreter* $\Im$ of the process model. $\Im$ executes the process model by controlling the tools used during development, guiding people, and verifying that the constraints embedded into the model are satisfied.

- A *user interaction environment* (UIE). The UIE is composed of the tools used by the people involved in the process. Examples of such tools are editors, compilers, test profilers, user agendas, project management tools, and so on. They are controlled by $\Im$, which uses them to receive feedback from the users and to support them during the process.

- A *repository* $\Re$. $\Re$ stores the *artifacts* produced during the process and managed by the PSEE (e.g., source code modules, documentation, executables, test cases, reports, Gantt charts) together with the current state of the process model being enacted. In other words, it supports $\Im$ by providing persistency of its data.

Figure 5 describes the architecture of a generic process engine.
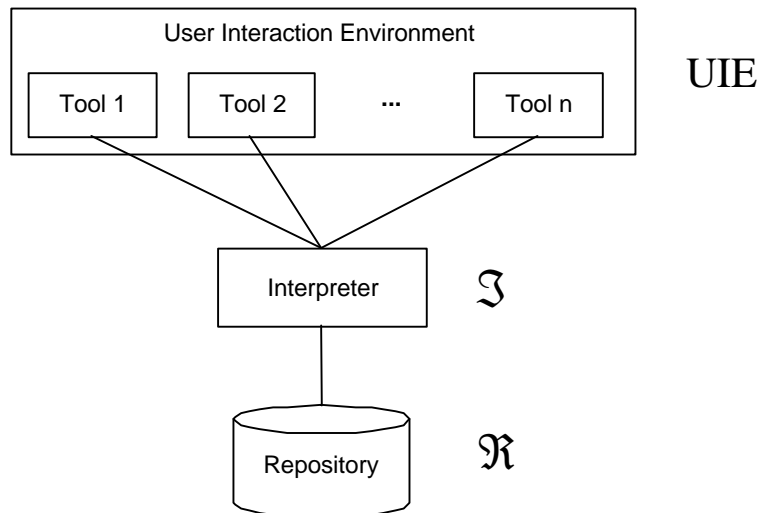
Figure 5: The architecture of a generic process engine

A large number of prototype PSEEs was developed in the late 80s early 90s. Many of them are available free of charge to the interested parties to stimulate cross-fertilization and to favor the interchange and feedback among researchers. Among these we recall Adele [12], ALF [16], AP5 [4], Arcadia [71], EPOS [17], HFSP [51], Marvel [49], Merlin [48], OIKOS [57], and SPADE [7]. Others became commercial products, like LEU [30], IPSE 2.5 [75], SynerVision [41], and ProcessWeaver [33].

Each PSEE is characterized by its PML. The PML is used to describe the process model that has to be followed, which is analyzed and enacted by the environment. A PML supports the description of several concepts that characterize a software development process, such as:

- *Activities*. They are the process steps used to produce and maintain artifacts.

- *Artifacts*. They are the input and output of activities. They are stored in the proces engine's repository.

- *Roles*. They describe the rights and responsibilities of the agents in charge of activities. A role is a static concept while the binding between a role and an agent can be dynamic. In general, agents during the same process can play several roles, and a role can be played by several agents.

- *Human agents*. They are in charge of executing certain activities that compose the process while playing certain roles.

- *Tools*. They automate execution of certain activities.

PMLs can be classified with respect to the paradigm they adopt to model the process. We can distinguish among four main paradigms:

- *Programming language based*. Process modeling languages that belong to this class extend existing conventional programming languages by introducing concepts related to the software development process. One of the best known examples of PMLs belonging to this class is APPL/A [40], an extension of Ada.

- *Rule based*. This class of PMLs comprises of languages that use production rules to describe the software process. Activities are described by rules with a precondition, an action, and a post condition. Rules have an associated role, which is responsible for the activity, and a set of resources like tools, necessary to perform the activity. Examples of rule-based PMLs are Marvel and Merlin.

- *Extended automata based*. Graphical state-machine based languages, like Statecharts or Petri nets, have been used to model software processes [52]. In many cases, these

formalisms were extended to provide a more expressive notation oriented to software processes. Leu and ProcessWeaver are two examples of Petri net based PMLs.

- *Multiparadigm*. Process modeling languages that belong to this class combine two or more distinct paradigms to describe the different facets of a software development process. SPADE can be considered as an example of a multiparadigm PML. Although its underlying structure is based on Petri nets, it provides an object-oriented data model to describe artifacts, and uses an operational language to describe actions associated with net transitions [8].

Another important feature of PSEEs is the support they can offer to their users. One can distinguish among four kind of possible support [5]:

- *Passive role*. The user guides the process and the PSEE operates in response to user requests.

- *Active guidance*. The PSEE guides the process and prompts the users as necessary, reminding them that they should perform certain activities. The users are still free to decide if they will perform the suggested actions or not.

- *Enforcement*. The PSEE forces the users to act as specified by the process model.

- *Automation*. The PSEE executes the activities without user intervention.

Often, the same PSEE adopts more than a single form of user support. As an example, SPADE adopts the automation approach for activities that do not require user intervention and the enforcement approach for the other activities.

A final classification of PSEEs distinguishes them with respect to the way they control and guide the process. In *proactive* PSEEs it is the environment that initiates and controls the operations performed by humans. Conversely, *reactive* PSEEs are passively subordinate to users. Most of the first generation PSEEs that we mentioned before are proactive.

3.7 Other approaches

The focus of process research during the past decade mostly concentrated on process programming. Other directions, however, were also investigated. Some of them can be viewed as widening the initial scope of process programming. For example, [9] and [18] focus on understanding real-life processes by viewing and capturing the relevant events that occur as the process progresses in the real world.

Other, completely different, research directions were also investigated. Several researches carried out a series of empirical studies on software process to analyze how real software factories operate and how their processes could be improved (for example see [63,64]). Cusumano [24] also did an empirical analysis of how Japanese software factories operate at a more macroscopic level. Cusumano and Selby [25] did a well-known recent study analyzing how Microsoft works.

Yet another approach was investigated in [1], which defined a general model for software processes based on systems dynamics. Similarly, Lehman [55] modeled a software process like a feedback system and used this metaphor to study and analyze it. The latter is a continuation of previous work, which aimed at identifying laws for software evolution [11].

Yet another approach was proposed by Watts Humphrey [42], who elaborated the Personal Software Process (PSP), an empirically guided process improvement methodology scaled down to the individual developer. Humphrey's work was motivated by the observation that software development is a human intensive activity that can be improved only by improving the way each individual developer operates.

4 Lessons learned and on-going efforts

In Section 3 we examined the historical evolution of software process research by identifying a series of myths, which were proposed as general solutions to the software process problem. Indeed, each myth identified a relevant aspect of the problem, but failed because it ignored other, equally important aspects. The lifecycle myth pointed out that a process model is needed to understand and control how development progresses. It failed when it tried to propose the same reference lifecycle for all software development processes. The methodology myth stressed the need for methods to support developers in their tasks. But methods should be based on sound formal foundations (as stressed by myth 3), enforced, and supported by automated tools (as stressed by myth 4). Since processes are project specific, one should be able to program the specific required process by using some process notation supported by a PSEE (myth 5). Based on this view, process programming looks like the glue that can put together all previous software process myths to make them part of a composite solution in which they complement each other.

After more than 10 years of research in the areas of process programming and process-centered software engineering environments, it is important for the researchers in the field to evaluate whether this ambitious goal has been reached. The purpose of this section is exactly to understand the achievements and the failures of these research streams, and to identify possible paths for future research in these areas.

We can see the impact of process programming and PSEE research on state-of-the-art configuration management tools, which incorporate a great deal of process notions. Tools like CCC [68], Continuus [78], and PCMS [79] provide features to describe and implement a different software development process for each project. The set of activities that can be carried out in each step of the process can be described, usually through an executable scripting language. In some cases, the constraints that regulate process enactment may be described, too. This gives process designers the ability to tailor the support offered by the tool to the specific needs of the adopted software development process.

In spite of this and other successes, however, we must acknowledge that process programming did not succeed in the task of pulling all previous myths together, and no PSEE gained general acceptance or widespread use. We will try here to contribute to identifying the causes of this failure and to refocusing on-going research.

We argue that process programming failed primarily in the aspects of software development that involve humans. The emphasis of most PMLs and PSEEs has been on describing process models as *normative* models, i.e., on describing (and prescribing) the expected sequence of activities and pushing automation to enforce them. Process-centered software engineering environments have been developed and used as a mean to impose good practices and uniform behaviors. They were often described as the facilities that can overcome the lack of quality in people.

"The actual process is what you do, with all its omission, mistakes, and oversights. The official process is what books say you are supposed to do". This quotation from Watts Humphrey represents quite well the implicit viewpoint that underlies most software process research. People are fallible, they make mistakes, and they are unreliable. The process model is the guideline that is provided for their benefit: to guide them through the right path and to prevent them from making mistakes. This pessimistic viewpoint is perhaps responsible for the prescriptive approach adopted by most PSEEs, and we argue that this is also the main reason for their failures.

We view software processes as *human-centered processes*. Humans have a central role in performing the activities needed to accomplish the process goals. They do so by interacting and cooperating among themselves and with computerized tools. The goal and purpose of such tools is not to replace humans, but to facilitate their work and increase their effectiveness. Human-centered processes are characterized by two crucial aspects, that were largely ignored by most software process research: they must support *cooperation* among people, and they must be highly *flexible*.

Supporting cooperation is crucial since software development is a complex process, which involves cooperation and collaboration of many people for long time. Software engineers cooperatively negotiate requirements, develop specifications, design the software architecture, develop and test program modules, etc. Cooperating designers are often working in a distributed environment, where a LAN connects individual workstations. Increasingly, cooperative workgroups became geographically distributed, and complex cooperation patterns are possible. PSEEs can thus be viewed as examples of environments for Computer Supported Cooperative Work (CSCW) in the software development field.

Flexibility is another crucial aspect. The final goal of a software process is not to constrain people to follow a predefined pattern of activities, but to provide support to their creative tasks. The responsibility of what to do, how to do, and when to do certain activities must be in the hands of the designers. The process is too complex and intrinsically dynamic to be definable in all details advance. Moreover, no matter how carefully the process is defined, in practice people often need to deviate from the normative description embodied into the process model. A PSEE has to be flexible enough to allow the process to be modified as it is running and to allow people to cope with the unexpected situations that arise during the process [19]. The former problem requires the PSEE to support *process evolution*, by allowing process engineers to improve the process description as experience is gained as the process is in progress [56]. For example, the SPADE environment supports process evolution by making its process language SLANG reflective, and defining process change processes [6]. In this section we will elaborate on the latter problem (*process deviation*), which is of paramount practical relevance.

In cooperative workgroups, there is an intrinsic tension between the need for defining a process on which people agree, which encodes the current understanding of how work should progress and defines the expected course of actions, and the fact that inevitably, no matter how carefully the process is defined, unexpected situations arise that are not reflected in the process model. To cope with such situations, software engineers should be allowed to *deviate* from the prescribed process. The decision to deviate is the designer's responsibility. Deviations are problematic, because they cause *inconsistencies* between the process model and the actual process [22]. The challenge here is to provide the PSEE with ways to manage deviations and inconsistencies in a controlled fashion. These points will be taken up more systematically later in Section 4.1.4.

Until now, the dominant approaches to software processes and process technology adopted a closed world assumption[3]. Once a process is defined, the user is limited in what she or he can perform. The actions that can be performed are only those that are pre-specified by the process designer. Moreover, PMLs tend to force process designers to over-specifying the process for completeness. The focus on automation also results in process over-specification. Process designers focus on the details that lead to process automation and fail to capture the higher level constraints that characterize the desired process. Moreover, process models focus on what people have to do, while it is often more important to say what they have not to do, leaving them free to decide how to operate under these constraints. In summary, they focus on details, which are more subject to change than higher level constraints, and prescribe behaviors from which developers often need to deviate. This problematic approach in process programming is further complicated by the fact that process evolution is difficult and on-the-fly process deviations from the model are impossible.

Given these considerations, we may ask ourselves if "process programming" is the right focus for process research. It is fair to observe, in fact, that the closed world assumption and the emphasis on prescriptive behaviors are rooted into the "process programming" concept. As observed by Leon Osterweil during his presentation at ICSE 19 [60], however, "Programming is not the same as coding, it entails the many diverse steps of software development. Software process programming should, likewise, not simply be coding, but seemed to entail the many non-coding steps usually associated with application development.

---

[3] Some of the problems of this dominant approach were raised by a number of researchers (e.g., see Kishida and Perry's "Session Summary on Team Efforts" [54].

[Furthermore], there are many examples of application code that are not inordinately prescriptive, authoritarian, or intolerable to humans. Thus there should be no presumption the process code must be overly prescriptive, authoritarian, or intolerable either. Process programs need not treat humans like robots".

The motivations that underlie research on process programming (i.e., the need for tailoring the process to the organization and to the product developed, and the need for modeling software processes to analyze, improve, and guide them) are still valid. The problem is in the approach adopted to reach this goal. We need to look for new kinds of programming principles that can support human intensive processes, like software processes. Process evolution and process deviation should be treated as first-class citizens, not as minor and undesirable exceptions. Modern software factories operate in a highly dynamic market, under tight time and cost constraints. The adoption of software processes should not be viewed as a way to slow down the process, but rather as a way to better support its flexibility and timeliness. Moreover, people should not be considered like robots to be guided in a step-by-step fashion. They should be viewed as the most precious resource in software development: PSEEs are there to serve developers, not vice versa.

4.1 A sampler of on-going efforts

As we observed above, software processes are a kind of distributed, cooperative human-centered processes. This has been recognized by a number of more recent research efforts, some of which are discussed below. We explicitly acknowledge that this is not an exhaustive survey of the field, but rather a sampler of some representative research efforts.

4.1.1 JIL

Developed by the LASER research group at University of Massachusset at Amherst, JIL [70] is the result of Osterweil's reflection on the successes and failures of first-generation research on process programming and PSEEs. It is an interesting and ambitious effort aiming at providing a semantically rich language, which features high-level, process specific constructs. This research was driven by the belief that the process programming research failed because the proposed PMLs did not provide the appropriate constructs and mechanisms that are needed for programming this very special kind of "software" like the "software process".

According to Osterweil's assessment, first generation PMLs bear close remembrance to programming languages or to workflow languages. The former are computationally powerful, but their abstraction are too low level to easily describe complex software process, while the latter tend to be higher level but relatively limited computationally. JIL tries to keep the best from these two approaches by providing a PML that is semantically rich and also features high-level, process specific constructs.

JIL is an activity-oriented language combining proactive and reactive control flow together with a powerful mechanism to describe the actions that have to be undertaken if something goes wrong and an exception arises. The central construct in JIL is the *step*. A JIL step is intended to represent a step in a software process. A JIL program is a composition of steps. The elements of a step specification include:
- the declarations of the software artifacts used in the step;
- a specification of the resources needed by the step, including people, software, and hardware;
- a set of substeps that contribute to the realization of the step;
- a set of constraints on the relative execution order of substeps;
- an imperative specification of the order in which substeps are to be executed (direct invocation);
- a reactive specification of the conditions or events in response to which substeps are to be executed (indirect invocation);

- a set of preconditions, constraints, and postconditions, which define artifact consistency conditions that must be satisfied (respectively) prior to, during, and subsequent to the execution of the step;
- a set of exception handlers for local exceptions, including handlers for consistency violations (e.g., precondition violations).

To provide the necessary flexibility to software processes, JIL provides constructs to specify the control flow, including the ORDERED, UNORDERED, and PARALLEL operators to describe the desired sequence of process steps and the REACT construct for programming reaction to events.

As this brief description hints, JIL is a very complex language, whose ambition is to cover all the requirements posed by a complex cooperative and distributed environment like the software process. We view JIL as an example of a *maximalist* approach to the process programming problem. This is in contrast with a *minimalist* approach, which would try to develop a very minimal set of constructs and a lightweight support environment. We will outline an example of a minimalist approach in Section 5, as a manifesto for our future research in the area.

It is also interesting to discuss JIL's approach to handling unexpected situations, which is based on exception handling. Exception handlers can be used to specify the actions that have to be pursued if an anomaly arises during the process. Examples of anomalies include violating process constraints (i.e., step preconditions or postconditions) or failing in accomplishing an operation. Based on our previous discussion, we would argue that this approach to managing unexpected situations is not general enough. It allows users to cope with a predefined number of situations, sometimes called *expected exceptions* (see [31]), which are captured by one of the exception handlers provided as part of the model. If an exception arises, which does not have a corresponding exception handling procedure, the PSEE cannot provide any help to its users. In other words, JIL does not provide support to managing deviations and inconsistencies as they were defined in Section 4.

4.1.2 Oz and Oz Web

Developed at Columbia University as a successor of Marvel [49], Oz [13] was the first "decentralized" PSEE. Several PSEEs developed before Oz support physical distribution of the developers that cooperate in a software process by adopting a client-server architecture to implement the structure described in Figure 5. People may access the services provided by a PSEE by running the front-end tools on their client, and connecting to a server providing the interpreter and a centralized repository. This solution is suitable to supporting small-to-medium, strongly connected teams, which cooperate on the same process through a LAN. Usually, these teams are composed of people working for the same organization. The increasing complexity of software development processes, however, requires the adoption of new development paradigms to support the cooperation of different teams, possibly belonging to different and geographically distributed organizations. The standard client-server PSEE architecture we mentioned in Section 3.6 cannot support such a kind of "multi-team development". Each team has to be able to use its own set of tools, procedures, and data. At the same time, these "autonomous" teams need to collaborate in order to develop the product. For example, they may need to share tools, to exchange files and other data, and they may need to agree on some common policies or procedures, at least for the part of the work that involves collaboration. Oz supports such kind of "decentralized" (as opposed to "distributed") development.

The Oz PSEE is composed of a *federation* of two or more sub-environments. Each sub-environment has complete control of its process, tools, and data, while allowing access by remote sub-environments under restrictions that are solely determined by the local sub-environment. In order to support cooperation of sub-environments a common sub-process has to be defined. This is obtained by defining one or more *treaties*. A treaty defines a common sub-process, a common sub-schema for accessing data, and a set of access constraints, to allow the cooperation between two sub-environments. The common sub-process defined by a

treaty is executed by adopting a *summit* model. During a summit two sub-environments cooperate by executing a sub-process previously defined by means of a treaty. The treaty and summit negotiation model allows the required level of interoperation among separate PSEE, still keeping each PSEE autonomous.

Based on Oz, OzWeb [50] allows a set of users to collaborate by accessing and manipulating a set of hypermedia documents according to a well-defined workflow model. It uses standard web technologies (i.e., the HTTP protocol, the HTML language, and conventional web servers) to support access and manipulation of hypermedia documents, but it improves the standard web infrastructure by introducing workflow modeling and support facilities. OzWeb is based on two concepts: *subwebs* and *groupspaces*. A subweb organizes on-line material of plausible interest to a process or organization over its lifetime. It supports structured associative and navigational queries, and unstructured information retrieval and navigation via hyperlinks. A groupspace provides the services needed to define and enact a process model that defines the way hypermedia documents managed by subwebs have to be accessed and manipulated to fulfill a particular task. OzWeb extends the Oz server to operate as a subweb server exporting all the needed services to implement a groupspace. In particular, the Oz process modeling language, the Oz process engine, and the Oz object-oriented DBMS have been extended to use standard web technologies. The result is a *hypermedia collaboration environment* in which people dispersed across the Internet may collaborate to pursue a common goal.

4.1.3 APEL

Developed at "Laboratoire Logiciels, Systèmes, Réseaux", France, APEL [32] is a PSEE, which pursue two main goals:
1. To support interoperability among heterogeneous PSEEs, allowing a process designer to build a a federation of PSEEs to manage complex, distributed processes;
2. To support process evolution, in order to cope with unforeseen situations during enactment.

To support federation of existing PSEEs, the APEL team has identified two basic architectures:

- In a **control based** architecture interaction between PSEEs is based on "process routine calls". Each PSEE is an autonomous entity, which encapsulates the part of the process it is responsible for. A supervisor PSEE exists, which holds the common process model. Such model expresses the relative ordering among sub-processes to be executed by other PSEEs. It formalizes, at a high abstraction level, the process performed by the federation. According to the common model it executes, the supervisor PSEE invokes other PSEEs to perform the required sub-tasks.

- In a **state based** architecture, each PSEE shares a common representation of the state of the global process. Interaction among PSEEs is implicit (i.e., direct call are never performed) and based on the common state. In particular, each PSEE in the federation can observe the common state, and update its local state accordingly, or change the common state according to changes performed in its local state during execution of its local model.

Each of these approaches has pros and cons. In the control based approach, PSEEs have a very limited view of the process they are participating in, but there is formal knowledge of what will be executed (i.e., the overall process is formalized into what has been called the common process). In the state based approach, each PSEE has a precise view of the current state of the process, but a description of (and consequently the control over) the overall process is lacking. APEL adopts a mixed architecture, which provides the benefits of both the previous approaches and allows a smooth transition between control and state based federations, if required. The APEL environment includes the following components:

- A common state, available to each PSEE and kept by a process server. During enactment, the process sever holds both a reification of the common process model and a reification of all the entities created during execution. The process server interface allows

components to create any entity and to change the current process as well as the process model. This capability is the basis for evolution support.

- A common PSEE, which executes the common process model with respect to the common meta-model semantics [27], and makes the common state evolve. The common PSEE ignores other PSEEs, which coordinate through the common state. The common process model includes a high level description of the "functional" aspects of the overall process, ignoring the operational aspects.

- An interoperability PSEE, which executes an interoperability model. This model contains operational information related to the consistency control of the common model, like ordering of tasks and transaction control.

- A set of heterogeneous PSEEs, which can access the common state and can be controlled by the interoperability PSEE.

- An event server used to support event-based communication among the process server, the common PSEE, the interoperability PSEE, and the other PSEEs which compose the environment.

4.1.4 Endeavors

Developed at the University of California at Irvine, Endeavors [15] is an open, extensible, Internet-based PSEE whose main goal is to support software process flexibility by minimizing the effort needed to change the process model on the fly (i.e., during enactment).

Endeavors supports an object-oriented definition and specialization of activities, artifacts, and resources associated with a software development process. Endeavors' *activity networks* define the inter-relationships among activities, artifacts, and resources as well as sub-networks. Networks include the definition of control flow, data flow, and resource assignments, and can be easily defined using a graphical network editor.

To enable cooperation among large groups, Endeavors supports both distribution of people and distribution of artifacts and process fragments via WWW protocols. The process fragments that compose the process model being executed can be downloaded through the network and the artifacts produced during the process can be maintained in a distributed repository and accessed via standard protocols like HTTP and FTP.

To support on-the-fly change of the process model being executed, Endeavors allows dynamic modification of object fields, methods, and behaviors at runtime. Stakeholders can customize the abstraction levels for behavior and data that are appropriate for their site and their customization skill and authorization level. For example, technically sophisticated stakeholders may customize behaviors while non-technical people may be limited to simply setting the value of some fields (essentially, they perform some kind of parameterization). Activity networks may be also changed at run-time through a graphical interface, thus allowing users to change the control and data flow.
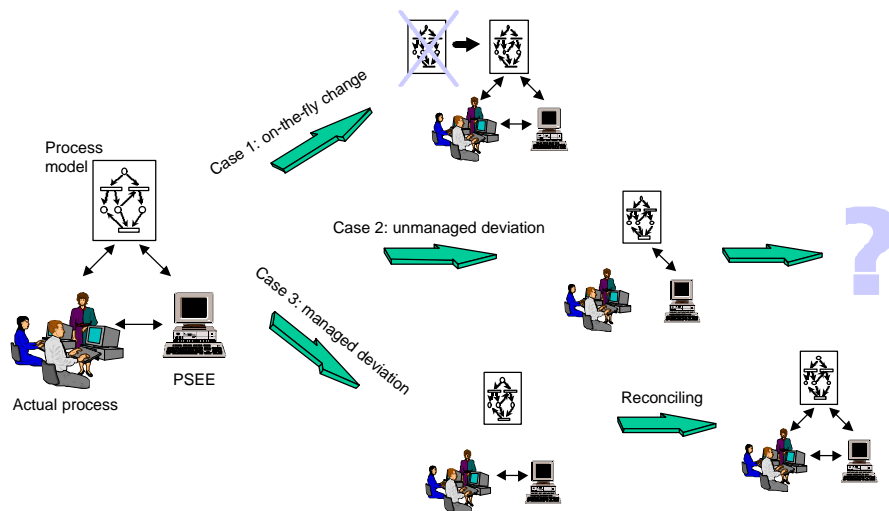
4.1.5 Sentinel

To support people when something unexpected happens, two approaches are possible:
(i)   change the process model on-the-fly in order to describe the new situation and then operate according to the new model; or
(ii)   allow people to explicitly deviate from the process model.

A few first-generation PSEEs identified the importance of managing unexpected situations during the process and adopted the former approach. As an example, SPADE provides a reflective PML that allows process modelers to formalize not only a software development process but also the process of changing the model itself (i.e., the *meta-process*). The experience gained using SPADE has shown that supporting process change is fundamental to cope with major deviations, that are likely to occur again in the future, but it is

not adequate to cope with minor deviations that require an immediate answer. The effort needed to change the process model makes this solution inadequate in the latter case.

Figure 6 [20] illustrates the problem of managing deviations in a PSEE. Case 1 describes the previous approach (i); cases 2 and 3 describe two ways of supporting deviations. Case 2 describes the situation where people deviate from the model by performing some actions outside PSEE's control. For example, the PSEE does not allow deletion of certain document at a certain stage, thus the developer decides to exit the PSEE and deletes the file storing the document. As a consequence of case 2 deviations, the environment may enter a critical situation in which its internal state does not reflect the state of the actual process (we call this situation an *inconsistent state*). As a consequence, it cannot further support the process in a sensible way. Case 3 describes the situation where the PSEE allows developers to deviate from the process model in a controlled fashion The PSEE is aware of the fact that developers deviate from the prescribed course of actions. Its internal state continues to reflect the state of the actual process. It may analyze the deviation, continue to support the process, and even suggest the actions needed to reconcile the actual process and the process model. To the best of our knowledge, SENTINEL [23] is the first example of a PSEE that supports case 3 deviations. The approach adopted by SENTINEL to manage inconsistencies has been inspired by the work of Balzer [3], which set the initial stage for research efforts aiming at tolerating inconsistencies in PSEEs.

Figure 6: Possible approaches to cope with unexpected situations during enactment

Activities are modeled in LATIN (SENTINEL's PML) as collections of *task types*. Each task type describes an activity as a state machine and is characterized by a set of *state variables*, a set of *transitions*, and a *state invariant*. State variables determine the structure of the internal state of a task type. The state invariant is a logical predicate that has to hold during the process. It models process-specific constraints. State transitions are characterized by a precondition, called ENTRY, and a body. The ENTRY is a logical predicate defining a property that must be satisfied to execute the corresponding transition. LATIN offers two kinds of transitions: *normal transitions* and *exported transitions*. A normal transition is automatically executed by the process engine as soon as its ENTRY evaluates true. If more than one transition precondition evaluates to true, one of them is chosen nondeterministically. An exported transition is executed if the user requests it and its ENTRY is true. However, the user can force the execution of an exported transition even if its ENTRY is not verified. In such a case, we say that the transition *fires illegally*. By forcing exported transitions to fire illegally, users can deviate from the process model to deal with unexpected situations. SENTINEL records the relevant events occurred during enactment in a knowledge base for later use. The enactment is suspended only if one of the invariants is violated. If this happens a *reconciling activity* starts to reconcile the actual process and the process model. To perform

this activity the process modeler may benefit from accessing the information stored in the knowledge base to perform *pollution analysis*. Pollution analysis identifies illegally fired transitions and potentially polluted variables through a logical reasoning on the knowledge base.

5   A path to the future

Modern software factories have changed the way they produce software. Just as an example, the experience of Microsoft [25] shows that the adoption of loose processes, which can provide maximum flexibility, has been a crucial success factor. More generally, this proved to be true for software that has to compete in highly dynamic markets (e.g., software for the Internet, multimedia software, and more generally software for the mass market). In such fields time to market is a key factor. Moreover, the network is not only increasingly becoming "the" environment, but it is also becoming the global marketplace in which competition takes place. Products can be easily advertised and distributed through the network and this enables small companies to compete with large ones. Small companies are usually more flexible and dynamic than large ones. To remain competitive and to continue expanding, large software factories have to adopt flexible processes similar to the ones adopted by small companies.

To compete on these new markets, it is often necessary to change product (and process) requirements up to the point of delivery, to respond to new market trends or to new products of competitors. The underlying management philosophy is that one should be allowed to delay process and product related decisions to the latest possible point in time to respond to changes in the market with changes in the product (and in the process). Ideally, product changes should be accommodated up to the delivery time. As an example, according to Cusumano and Selby [25], Microsoft managers consider product specification an "output" of the development process instead of an "input", and they adopt a very flexible process based on a limited number of strong rules. [4]

This need for flexibility should guide researchers in designing a new generation of PSEEs that could succeed where old PSEEs failed. When the design of a new PSEE starts several strategic decisions have to be taken, which will guide the design process. In the following we try to identify the most crucial decisions and provide an initial answer to them.

Minimalism vs. maximalism. A maximalist approach aims at providing a PML that can model all the possible situations that may arise in a software process. Its ultimate goal is completeness of process descriptions, from the general properties down to the details. Completeness is viewed as a way to prevent deviations. It is also viewed as a way to achieve the highest possible degrees of automation. The inevitable consequence is that developers aim at a complete description of the process before the process starts, and therefore they spend their efforts in trying to anticipate detail process aspects that are subject to change later. In addition, the maximalist approach requires the PSEE to provide a complex and heavyweight infrastructure. Conversely, a minimalist approach does not overspecify the process and results in lightweight support, where humans play a crucial role in deciding how the process has to progress.

We argue that a minimalist approach is preferable in the context of highly flexible and dynamic processes, which require support systems to easily adapt to changes in the process. The PSEE should help in identifying the minimum set of relevant constraints that developers should follow to achieve cooperation, leaving them free to decide their preferred way to accomplish their process tasks.

---

[4] The question whether the emphasis on timeliness in software development detracts from quality of the products is open to the discussion. We believe that the real challenge is to achieve the right balance between the two, which depends on the type of product. A discussion of this issue, however, is out of the scope of this paper. Here we simply acknowledge that current market strategies for software products demand shorter time to market and late binding of product and process decisions.

Infrastructure for cooperation vs. automation. The goal of a PSEE is to support cooperative software development and to automate the portions of the process that are amenable to mechanization. As such, they integrate the features of CSCW (Computer Supported Cooperative Work) with the features of CASE (Computer Aided Software Engineering) tools. Although automation is an important goal, we argue that it should not be the primary goal of a PSEE. We also argue that the exceeding importance given to automation was one of the main cause of failure of first generation PSEEs. Software processes are human-intensive processes. The automated environment should help developers in making design decisions by managing complexity. It should support collaborative design. Automated tasks are a small number (even if they are usually performed quite often during the process, e.g., build or test activities). As a consequence, we feel that next generation PSEEs should focus more on supporting communication and cooperation.

Centralized vs. decentralized process support infrastructure. First generation PSEEs adopted the architecture shown in Figure 5, where a server provides both the process engine and the repository. As we mentioned, this architecture has severe limitations. First of all, it centralizes process enactment, which results in a reduced scalability. Second, it depends on a DBMS to store the artifacts and the process state, which may be an obstacle when external tools have to access the artifacts managed by the engine. Third and last, it is characterized by a strict coupling between external tools and the process engine. The engine must know about the existence of the tools it has to control and they have to be able to interact with the engine. This may complicate the activity of integrating off-the-shelf tools into the environment.

Modern PSEEs need to be based on an open infrastructure capable of simplifying the integration of external tools and supporting the dynamic reconfiguration of the system to cope with changes in the external environment. Moreover, PSEEs should be able to support nomadic users, who use mobile devices (like Notebooks or PDAs) to interact with the environment. Nomadic users connect to the network from arbitrary locations, move from a location to another during system interaction, and may also be disconnected for a period of time. Supporting nomadic users poses additional constraints to the PSEE infrastructure. Nomadic users have to be supported during their migration and even when they are not connected.

Activity-oriented vs. artifact-oriented. In an artifact-oriented PML, processes are modeled through a description of the artifacts produced during the process and the actions that can be applied to these artifacts. In an activity-oriented language, processes are described as activities, composed of sequences of steps. Most of the arguments raised to demonstrate the benefits of object-oriented languages with respect to procedural programming languages could be applied to artifact-oriented PMLs versus activity-oriented PMLs. The most compelling one, in our opinion, is that focusing on activities results in a premature concern for control flow, which, in turn, may result in over-specifying the process. Conversely, an artifact-oriented PML focuses on semantic aggregations. Constraints are associated with the artifacts, rather than being implemented by suitably sequencing the steps of the activities. Furthermore, as we will further describe in Section 5.1, an artifact-oriented PML can support the management of deviations during process enactment in a natural way.

Section 5.1 outlines PROSYT, a prototype PSEE in which we are trying to address the above research challenges.

5.1 PROSYT: a step towards second generation PSEEs

PROSYT [20,19] exhibits two main distinguishing features: (i) it allows developers to deal with unexpected situations by deviating from the predefined process in a controlled fashion and (ii) it supports geographically distributed workgroups through a distributed cooperative

infrastructure. The infrastructure is based on open technologies, which simplify the integration of existing tools into the environment.

More specifically, PROSYT offers the following main features:

1. As for process modeling, the PROSYT PDL (called *PLAN*: the Prosyt LANguage) adopts an artifact-based approach. Each artifact produced during the process is an instance of an *artifact type*, which describes its internal structure and behavior. Each artifact type is characterized by

    (i)   a set of *attributes* whose values define the internal state of its instances;

    (ii) a set of *exported operations* that may be invoked by the users upon artifacts; and

    (iii) a set of *automatic operations* that are automatically executed when certain events happen (like invoking an exported operation on another artifact). Automatic operations are used to automate the process and to react to changes in the state of the tools controlled by the environment.

    It is possible to express the constraints under which exported operations are allowed to start and organize them in different classes, depending on the type of condition they express. As an example, a class of constraints is used to describe the preconditions of operations and another class is used to control the users who are allowed to invoke each operation. It is also possible to specify a set of artifact *invariants*, to characterize acceptable process states.

    To describe activities and invariants that refer to a collection of artifacts, PLAN provides the concepts of *repository* and *folder*. Each repository is an instance of some *repository type* and contains a set of folders organized in a tree structure. Each folder (instance of some *folder type*) is a container of artifacts and other folders. Attributes, states, exported operations, automatic operations, and invariants may be associated either with repository types or with folder types. Exported operations and invariants for folders and repositories may be used to describe business activities and constraints that refer to structured collections of artifacts.

    Finally, PLAN provides the concept of *project type*. Each PLAN process is described as an instance of some project type. It is characterized by a statically defined set of repositories, by a set of *groups* (each user belongs to one or more groups), and by a set of exported operations, automatic operations, and invariants, which refer to the entire process.

2. As for process enactment, to improve flexibility, PROSYT users are not obliged to satisfy the constraints stated in the process model. They can invoke operations even if the associated constraints are not satisfied. PROSYT keeps track of the results of these deviations and controls that the invariants are not violated as a result of such deviations.

    To better control process execution, PROSYT allows process managers to specify a *deviation handling* and a *consistency checking* policy. Such policies state the level of enforcement adopted (i.e., the classes of constraints that can be violated during enactment) and the actions that have to be performed when invariants are violated as a result of a deviation, respectively. Both these policies may by changed at enactment-time, and may vary from user to user.

3. As for system architecture, PROSYT adopts an event-based communication paradigm. The experience gained with previous event-based frameworks, like the Field [65] and Sun ToolTalk [69] environments, shows that the event-based coordination infrastructure simplifies the integration of third-party tools into the environment. Moreover, it simplifies system reconfiguration. New components may be added, existing components may be removed, and components may be moved from a host to another without affecting the remaining components. The PROSYT run-time architecture (see **Error! Reference source not found.**) adopts JEDI [21] as the communication middleware. JEDI is an

event-based Java framework, which includes features to move running components from host to host [74].

6 Summary and conclusions

The attention to software processes dates back to the early 70's, when software engineers realized that the desire to improve the quality of software products required a disciplined flow of activities to be defined and managed. Most of the software process work, however, remained in an informal stage until the mid 80's. From then on, the software process was recognized by researchers as a specific subject that deserved special attention and dedicated scientific investigation, the goal being to understand its foundations, develop useful models, identify methods, provide tool support, and help manage its progress.

In this paper we described both the initial approaches to software processes and the evolution of software process research from the early 80's up to now. By analyzing this evolution, we observed that several important results have been attained but also that some fundamental research issues are still open. In particular, we discussed process programming, which was the main research focus for more than a decade. We analyzed the reasons why process programming did not deliver what it promised, i.e., why none of the PMLs and PSEEs developed so far gained general acceptance or widespread use.

Until recently, the primary emphasis has been on describing process models as normative models. The focus of PMLs and PSEEs was on describing the expected sequence of activities and on pushing automation as far as possible. The result of this choice was the development of a class of PSEEs that are unable to achieve a high degree of flexibility. However, modern software factories compete in a highly dynamic market, which requires them to adapt to frequent changes in the "environment" in which they operate.

In Sections 4 and 5 we identified some strategic issues that should be taken into consideration in developing new PSEEs and we outlined the approach followed by our current research in the development of the PROSYT prototype. PROSYT adopts a lightweight process support, manages process deviations, supports geographically distributed workgroups, and handles nomadic users. It increases system flexibility by allowing developers to explicitly deviate form the modeled process, still continuing to control that the most relevant constraint regarding the overall process are verified.

Our work is just a first step in these directions and several issues are still open: How to support the reconciliation between the modeled process and the process actually followed after deviations occur? How to find the best balancing between support to cooperation and automation? How to provide effective support to widely distributed processes and nomadic users? How to find the best balancing between flexibility and control? Only when adequate answers to these questions have been found will PSEEs have a better chance of being accepted by industrial practitioners.

Acknowledgements

References

1. T. K. Abdel-Hamid and S. E. Madnick, "Lessons Learned from Modeling the Dynamics of Software Development", In *Communications of the ACM*, 32(12), December 1989.

2. V. Ambriola, R. Conradi, and A. Fuggetta, "Assessing Process-Centered Environments". *ACM Transactions on Software Engineering and Methodology*, 6(1), July 1997.
3. R. Balzer, "Tolerating Inconsistency". In *Proceedings of 13th International Conference on Software Engineering*, Austin (Texas - USA), May 1991.
4. R. Balzer and K. Narayanaswamy, "Mechanisms for Generic Process Support". In *Proceedings of the First ACM SIGSOFT Symposium on Foundations of Software Enginering*. ACM, Software Engineering Notes, 18(5), December 1993.
5. S. Bandinelli, E. Di Nitto, and A. Fuggetta, "Supporting Cooperation in the SPADE-1 Environment". *IEEE Transactions on Software Engineering,* 22(2), December 1996.
6. S. Bandinelli, A. Fuggetta, and C. Ghezzi, "Process Model Evolution in the SPADE Environment". *IEEE Transactions on Software Engineering*, IEEE Computer Society, December 1993.
7. S. Bandinelli, A. Fuggetta, C. Ghezzi, and L. Lavazza, "SPADE: an environment for Software Process Analysis, Design, and Enactment". In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors. *Software Process Modelling and Technology*. Research Studies Press Limited (J. Wiley), 1994.
8. S. Bandinelli, A. Fuggetta, C. Ghezzi, and A. Morzenti, "A Multi-Paradigm Petri Net Based Approach to Process Description". *In Proceedings of the 7th. International Software Process Workshop*, Yountville, California (USA), October 1991.
9. N. S. Barghouti and B. Krishnamurthy, "Using Event Contexts and Matching Constraints to Monitor Software Processes". In *Proceedings of 17th International Conference on Software Engineering*, Seattle (Washington - USA), April 1995.
10. F. L. Bauer, B. Moeller, M. Partsch, and P. Pepper, "Formal Program Construction by Transformations: Computer Aided, Intuition Guided Programming". IEEE Transaction on Software Engineering, 15(2), February 1989.
11. L. A. Belady and M. M. Lehman, "Characteristics of Large Systems". In *Research Directions in Software Technology,* P. Wegner Editor, The MIT Press, 1979.
12. N. Belkhatir, J. Estublier, and W. L. Melo, "Adele2: A Support to Large Software Development Process". In *Proceedings of the 1st International Conference on the Software Process*, Redondo Beach CA (USA), October 1991.
13. I. S. Ben-Shaul and G. E. Kaiser, "A Paradigm for Decentralized Process Modeling and its Realization in the Oz Environment". In *Proceedings of the 16th International Conference on Software Engineering*, May 1994.
14. G. D. Bergland, "A Guided Tour of Program Design Methodologies", *Computer*, vol. 14, no. 10, Oct. 1981, pp. 13-37.
15. G. A. Bolcer and R. N. Taylor, "Endeavors: A Process System Integration Infrastructure". In *Proceedings of the Fourth International Conference on Software Process (ICSP4)*, Brighton, UK, December 1996.
16. G. Canals, N. Boudjlida, J.-C. Derniame, C. Godart, J. Lonchamp, "ALF: A Framework for Building Process-Centered Software Engineering Environments". In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors. *Software Process Modelling and Technology*. Research Studies Press Limited (J. Wiley), 1994.
17. R. Conradi, J. Larsen, M. N. Nguyên, B. P. Munch, P. H. Westby, W. Zhu, M. L. Jaccheri, C. Liu, "EPOS: Object-Oriented and Cooperative Process Modelling". In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors. *Software Process Modelling and Technology*. Research Studies Press Limited (J. Wiley), 1994.
18. J. E. Cook and A. L. Wolf, "Toward Metrics for Process Validation". In *Proceedings of the 3rd International Conference on the Software Process*, Reston (Virginia - USA), October 1994.
19. G. Cugola, *Inconsistencies and Deviations in Process Support Systems*. Ph. D. Thesis, Politecnico di Milano - Dipartimento di Elettronica e Informazione, February 1998.
20. G. Cugola, "Tolerating deviations in Process Support Systems via Flexible Enactment of Process Models". In *IEEE Transactions of Software Engineering*, 24(11), November 1998.

21. G. Cugola, E. Di Nitto, and A. Fuggetta, "Exploiting an Event-Based Infrastructure to Develop Complex Distributed Systems". In *Proceedings of the 20th International Conference on Software Engineering (ICSE98)*, Kyoto (Japan), April 1998.

22. G. Cugola, E. Di Nitto, A. Fuggetta, and C. Ghezzi, "A Framework for Formalizing Inconsistencies in Human-centered Systems". *ACM Transactions On Software Engineering and Methodology (TOSEM)*, 5(3), July 1996.

23. G. Cugola, E. Di Nitto, C. Ghezzi, and M. Mantione, "How to Deal with Deviations during Process Model Enactment", In *Proceedings of the 17th International Conference on Software Engineering*, Seattle (Washington - USA), April 1995.

24. M. A. Cusumano, *Japan's Software Factories.* Oxford University Press, 1991.

25. M. A. Cusumano, R. W. Selby, *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People.* Free Press, October 1995.

26. O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming.* Academic Press, 1972.

27. S. Dami, J. Estublier, and M. Amiour, "APEL: a Graphical Yet Executable Formalism for Process Modeling". Kuwler Academic Publisher, pp. 60-96, Boston, January 1998.

28. T. De Marco, *Structured Analysis and System Specification.* Yourdon Press, 1978.

29. E. W. Dijkstra, *A Discipline of Programming.* Prentice Hall, 1976.

30. G. Dinkhoff, V. Gruhn, A. Saalmann, and M. Zielonka, "Business Process Modeling in the Workflow-Management Environment Leu". *In Proceedings of the Entity Relationship Conference*, December 1994.

31. J. Eder and W. Liebhart, "The Workflow Activity Model WAMO". In *Proceedings of the 3rd International Conference on Cooperative Information Systems (CoopIS)*, Vienna, Austria, May 1995.

32. J. Estublier, P. Y. Cunin, and N. Belkhatir, "Architectures for Process Support System Interoperability". In *Prooceedings of the Fifth International Conference on the Software Process*, Lisle, IL, June 1998.

33. C. Fernström, "Process Weaver: Adding Process Support to Unix". In *Proceedings of the 2nd International Conference on the Software Process*, Berlin (Germany), February 1993.

34. A. Finkelstein, J. Kramer, and B. Nuseibeh, editors. *Software Process Modelling and Technology.* Research Studies Press Limited (J. Wiley), 1994.

35. P. K. Garg and M. Jazayeri, Proces-Centered Software Engineering Environments. IEEE Computer Society Press, 1996.

36. C. Ghezzi, M. Jazayeri, and D. Mandrioli, "Software Qualities and Principles". In *The Computer Science and Engineering Handbook*, A. B. Tucker, Jr editor, ACM Press, 1992.

37. C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fudamentals of Software Engineering.* Prentice Hall, 1991 (2nd Edition forthcoming).

38. D. Gries, *The Science of Programming.* Springer-Verlag, 1981.

39. A. N. Habermann and D. Notkin, "Gandalf Software Development Environments". In *IEEE Transactions on Software Engineering* 12(12), pp. 1117—1127, December 1986.

40. D. Heimbigner, S. M. Sutton, and L. Osterweil, "Managing Change in Process-Centered Environments". In *Proceedings of the 4th ACM/SIGSOFT Symposium Software Development Environments*, ACM Software Engineering Notes, vol. 15, December 1990.

41. Hewlett/Packard Company, *Developing SinerVision Processes.* Part Number: B3261-90003, 1993.

42. W. S. Humphrey, *A Discipline for Software Engineering.* SEI Series in Software Engineering, Addison Wesley, 1995.

43. W. S. Humphrey, *Managing the Software Process*, Addison-Wesley, SEI Series in Software Engineering, 1989.

44. ISO 9000-3 task force, *Quality management and quality assurance standards - Part 3: guidelines for the application of ISO 9000 to the development, supply and maintenance of software*, ISO--International Organization for Standardization, 1991.

45. M. A. Jackson, *Principles of Program Design.* Academic Press, 1975.

46. M. A. Jackson, *System Development.* Prentice Hall, 1983.

47. M. A. Jackson, *Software Requirements and Specifications*. Addison Wesley, 1995.

48. G. Junkermann, B. Peuschel, W. Schäfer, and S. Wolf, "MERLIN: Supporting Cooperation in Software Development Through a Knowledge-Based Environment". In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors. *Software Process Modelling and Technology*. Research Studies Press Limited (J. Wiley), 1994.

49. G. E. Kaiser, N.S. Barghouti, and M. H. Sokolsky, "Preliminary Experience with Process Modeling in the Marvel Software Development Kernel". In *Proceeding of the 23rd International Conference on System Sciences*, 1990.

50. G. E. Kaiser, S. E. Dossick, W. Jiang, and J. J. Yang, "An Architecture for WWW-based Hypercode Environments". In *Proceedings of the 19th International Conference on Software Engineering*, Boston (MA), USA, May 1997.

51. T. Katayama, "A Hierarchical and Functional Software Process Description and its Enaction". In *Proceedings of the 11th International Conference on Software Engineering*, 1989.

52. M. I. Kellner, "Software Process Modeling: Value and Experience". In *SEI Tech. Rev.* Software Engineering Institute, 1989.

53. B. W. Kernighan and R. Pike, *The Unix Programming Environment.* Prentice Hall, 1984.

54. Kouichi Kishida and Dewayne Perry, "Report on Session V: Team Efforts". In *Proceedings of the 6th International Software Process Workshop*, 28-31, Hakodate, Japan, October 1990.

55. M. M. Lehman, "Evolution, Feedback, and Software Technology". In *Proceedings of the 9th International Software Process Workshop,* October 1994.

56. N. H. Madhavji and M. H. Penedo editors, *IEEE Transaction on Software Engineering: Special issue on process evolution*. IEEE Computer Society Press , December 1993.

57. C. Montangero and V. Ambriola, "Oikos: Constructing process-centered SDEs". In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors. *Software Process Modelling and Technology*. Research Studies Press Limited (J. Wiley), 1994.

58. G. J. Myers, *The Art of Software Testing*. John Wiley & Sons, New York, 1979.

59. L. Osterweil, "Software Processes are Software too". In *Proceedings of the Ninth International Conference on Software Engineering*, 1987.

60. L. Osterweil, "Software Processes are Software too, Revisited: An Invited Talk on the Most Influential Paper of ICSE 9". In *Proceedings of the 19th International Conference on Software Engineering*, Boston (MA), USA, May 1997.

61. D. L. Parnas and P. C. Clements, "A Rational Design Process: How and Why to Fake It". In *IEEE Transactions on Software Engineering*, 12(2), February 1986.

62. D. E. Perry and G. E. Kaiser, "Models of Software Development Environments". In *IEEE Transactions on Software Engineering*, 17(3), March 1991.

63. D. E. Perry, N. A. Staudenmeyer, and L. G. Votta, "People, Organizations, and Process Improvement". In *IEEE Software*, July 1994.

64. A. A. Porter, L. G. Votta, H. P. Siy, and C. H. Toman, "An Experiment to Assess the Cost Benefits of Code Inspections in Large Scale Software Development". In 3rd IEEE Transactions on Software Engineering, 23(6): 329-346, June 1997.

65. S. P. Reiss, "Connecting Tools Using Message Passing in the Field Environment", *IEEE Software*, July 1990.

66. T. Reps and T. Teitelbaum, "Language Processing in Program Editors". Computer, 20(11), November 1987.

67. W. W. Royce, "Managing the Development of Large Software Systems: Concepts and Techniques". In *Proceedings of WesCon*, August 1970.

68. Softool Corporation. *CCC User's Guide*, August 1995.

69. SunSoft Inc., *The ToolTalk service: An inter-operability solution*, SunSoft Press/Prentice Hall, December 1992.

70. S. M. Sutton Jr. and L. J. Osterweil, "The Design of a Next-Generation Process Language". In *Proceedings of the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurich, Switzerland, September 1997. LNCS 1301, pp. 142-158.

71. R. N. Taylor, R.W. Selby, M. Young, F.C. Belz, L. A. Clark, J.C. Wileden, L. Osterweil, A.L. Wolf, "Foundations of the Arcadia Environment Architecture". In *Proceedings of the Third ACM SIGSOFT/SIGPLAN Symphosium on Software Development Environments*, 1988.

72. I. Thomas, "PCTE Interfaces: Supporting Tools in Software Engineering Environments, IEEE Software, 6(6), November 1989.

73. United States Department of Defence, *Stoneman: Requirements for Ada Programming Support Environment*, February 1980.

74. J. Vitek and C. Tschudin editors, *Mobile Object Systems: Towards the Programmable Internet*. Springer-Verlag, LNCS 1222, April 1997.

75. B. Warboys, "The IPSE 2.5 Project: Process Modeling as the Basis for a Support Environment". In *Proceedings of the First International Conference on System Development Environments and Factories*, 1990.

76. N. Wirth, "Program Development by Stepwise Refinement". In *Communications of the ACM,* 14(4), April 1971.

77. E. Yourdon and L. Constantine, *Structured Design.* Prentice Hall, 1979.

78. http://www.continuus.com

79. http://www.sql.com