Why is it so hard to find Feedback Control in Software Processes?

Meir M. Lehman, Dewayne E. Perry & Wlad Turski

Abstract

Early data on the phenomenology of software system evolution suggest that such evolution involves and is, to some extent, governed by feedback. This feedback may take the form of information fed back to individuals or groups as a form of learning from experience or may take the form of observation and data that are used to control some aspect of the process. For the moment, we shall put the former to one side and concentrate only on feedback to explicit control mechanisms.

Initial investigations, using a basic model for feedback control, have exposed a variety of reasons why software processes are not amenable to classical feedback control: software processes are design, not production processes; control-directed process changes tend to be step functions, not regulatory ones, and are often as creative as the processes they control; and system development and evolution processes are still immature with little theory to guide the design and application of regulation control mechanisms. Despite these limitations, we have found promising examples of feedback control and, on the basis of more recent phenomenological evidence, believe this area of research to be critically important and vital to understanding and controlling the development and evolution of software systems and improvement of software processes.

Introduction

Software development and evolution processes have become an significant area of software engineering and software engineering research. Among topics of importance are process formalisms, process support, process assessment, process architecture and process improvement. One of the underlying motivations for the emergence of this relatively new direction in research and practice is the need to move the development and evolution of software systems from a craft venture to an engineering one. An expected consequence of this move is that the methods and techniques by which software systems are built and evolved will be open to scrutiny and evaluation by the community rather than considered to be secrets passed amongst the initiated.

Given that the functionality of systems we build (and hence the systems themselves) can be exceedingly complex, that the processes we use to build and evolve these systems are complex, that the organizational structures that provide their development and operation context are equally complex, and that there is undoubtedly extensive feedback in the processes used and organization that executes them, it is surprising that feedback and feedback control have been so little investigated in the context of software systems evolution.

Despite the fact that the role and the impact of feedback has received little attention as a research topic, it has long been recognized as a significant factor in software processes. It was, for example, referred to in passing by several people at the software engineering workshop in Garmisch [nau68] and is also discussed briefly in Lehman's Programming Process report [leh69].

*** os360 figure here *****

Lehman and Belady [bel72, leh85] provide one of the earliest examples of feedback control at work in the evolution of OS/360. Figure 1 `depicts the growth of OS/360` in terms of modules over a period of 26 releases. The cyclic pattern evident in the plot from release 1 through release 20 is characteristic of feedback systems. They observed

> ... the ripple is typical of a self stabilizing process with positive and negative feedback loops. From a long-range point of view the rate of system growth is self-regulatory, despite the fact that many different causes control the selection of work implemented in each release, with budgets varying, increasing numbers of users reporting faults or desiring new capability, varying management attitudes towards system enhancement, changing release intervals and improving methods ...

It is in this context of evolutionary software development that we have for some time studied feedback and feedback control, a study for which the FEAST[1] project has provided a formal framework for this past two years [leh95, fea1, fea2, and fea3]. In this paper, we first examine the definitions and nature of feedback and control. We then present a research manifesto and feedback control model as the initial basis for our investigations. On the basis of this groundwork, we consider various facets of feedback and control in the context of software evolution processes: what feedback control means in design processes as opposed to production processes; what feedback control means when it leads to a change in processes rather than in their regulation; and finally, the contrast between feedback influence and feedback control in relatively immature processes. We then summarize our current results and discuss where to go from here.

Feedback and Control

As the term *feedback* is used in a wide variety of contexts, it is worthwhile to take a look at the basic meanings of the word. *Webster's New Collegiate Dictionary* defines ``feedback'' as follows:

> **feedback: 1:** the return to the input of a part of the output of a machine, system, or process (as for producing changes in an electronic circuit that improve performance or in an automatic control device that provides self-corrective action) **2 a:** the partial reversion of the effects of a process to its source or to a preceding stage **b:** the return to a point of origin of evaluative or corrective information about an action or process <student ~ was solicited to help revise the curriculum> <we welcome ... ~ from our readers - brickbats as well as bouquets – *Johns Hopkins Mag.*> **also :** the information so transmitted

The mere return of information, even if it is *evaluative or corrective*, does not guarantee that it will have any effect. To have an effect, this information must be somehow used, i.e. it must produce a change in something. And, while there are a variety of ways in which feedback may have an effect, we are here interested in one specific means of producing such effects --- namely, feedback control.

The verb *control* also has two principal (families of) meanings. Once more we quote from the *Webster's New Collegiate Dictionary*:

[1] FEAST stands for Feedback, Evolution, And Software Technology.}

> **1:** to check, test, or verify by evidence or experiments **2a:** to exercise restraining or directing influence over: REGULATE **b:** to have power over: RULE

These two meanings are often used interchangeably in everyday speech. However, when applied to software evolution processes, the activities denoted by "control—1" are quite different from those denoted by "control-2".[2] The confusion is amplified (or, perhaps, generated) by the fact that a single person (or a single group) often performs *both* actions, "control—1" and "control—2" with respect to a productive activity. In addition, it may happen that the same person or group performs "control—2" over several activities, particularly when "control-2b" is meant.

Nevertheless, a precondition for any sensible approach to a scientific and technological treatment of software evolution processes is that the meanings of "control" are *disentangled*. From now on we will use *check* for "control—1" and *regulate* (or possibly *rule*, if it is needed) for "control—2". Thus *checking* is distinct from *regulating*.

In a disciplined work environment, all productive work actions are checked: **do** *action* **until** *check-successful*. This qualitative function of checking is a part of production, not part of control. It guarantees an established level of completeness or quality of the production.

Regulation, on the other hand, is the control of the production process on the basis of the production results. It is this meaning of control that we use in the combination *feedback control*. The whole idea of applying feedback control to software evolution processes rests on the assumptions that there is a stream of similar production tasks and that regulation of the production processes is required to maintain an ideal production state.

There are two factors that effect the maintenance of this ideal production state: instability and random events. It is difficult to deny that software evolution processes are often unstable, or that random events occur in and impact these processes. Thus, many software development processes require feedback control to contain the tendency towards instability and to control the consequences of randomness.

In contrast to checking, regulating may have one of the following effects as a result of evaluating feedback.

- Change the processing --- that is, change various parameters that govern the production process
- Change the process --- that is, change the process structure itself rather the parameters that control the processes. There are two ways in which this change may be achieved:
    - statically --- use an alternative part of the production process
    - dynamically --- change the existing process or create a new process

Technology vs Sociology

[2] The distinction between "control--2a" and "control--2b", although important in many contexts, is less fundamental in our considerations as---usually---one has to have power over something if one is to exercise restraining or directing influence over it. With some hesitation we may accept that in the context of software evolution processes "control--2a" implies "control--2b".

Given our definitions of feedback and control, there are still a wide variety of feedback control phenomena that we want to exclude from our investigations. One such general category is that of *learning* as an example of feedback control. In this case, feedback is the information returned to a *person* placed at the point of origin, who absorbs the information and via an act of human learning modifies his or her future behavior -- for example, the way this person manages whatever happens to be his or her activity domain.

This interpretation is acceptable for the *sociology* of software evolutions processes. It can be a part of a manager's or developer's education: ``thou shalt pay (more) attention to the feedback you are getting"; or even more aggressively: ``thou shalt seek more feedback about the actions you manage". It can be elaborated by supplying a list of sources from which the feedback is to be considered or sought categorized into ``important", ``vital", and ``irrelevant" classes. Suitable case studies may be conducted, yielding instances of the benefits that accrue when one heeds the feedback message and of the disasters that follow when the feedback information is neglected. This, no doubt, can (and will) be a useful part of education and training for both managers and developers alike.

However, this type of feedback and control cannot easily be interpreted and modeled as a technological view of software evolution processes. The point is that the evaluation and control machinery is all in the human brain. Moreover, even if we accept that feedback provides the stimulus and basis for learning, we still face a dilemma: either we explain what the appropriate reactions are that need to be learned, or we leave that to intuition or creativity.

In the former case, i.e. when it is ultimately known what are the recommended, beneficial, profitable reactions to a particular combination of feedback signals received, we do have explicit control machinery (``when you get too many error reports coming from the customers strengthen the quality control", ``when you are late with delivery, cut down on the most time-consuming activity" etc.)

In the latter case (invoking intuition) such machinery is not readily apparent, but it is hard to see what advice can be given to a manager or developer as the (necessary) second part of the admonition to pay more attention to the feedback. A rational person will almost certainly ask: What am I supposed to do when I collect all this information fed back to me? How can I act on it? Unless we are prepared to answer ``use your head" or some similarly profound platitude, we are inextricably bound to construct control machinery.

Whatever other kinds of feedback are considered, if they are to be used for improving the software process they must be turned into explicit control mechanisms. Thus, we concentrate, at least initially, on feedback control as a technological rather than a sociological endeavor.

Manifesto and Model

As a prelude to our investigations (in the FEAST project), we laid down a manifesto defining our goals, identifying supporting postulates, describing a basic model and enunciating our research hypothesis. One of the advantages of this approach is that the manifesto provides the primary inputs to defining a project and developing a workplan.

We define two general goals for our investigations.

- To produce specific recommendations, guidelines, methods and toolsfor software evolution process improvement
- To contribute to a science of software process and software evolution

These goals are to be pursued in the context of process systems that satisfy a set of requirements about their structure and composition. That is, the FEAST project is limited to process systems that implement the evolution of software systems and that satisfy the following postulates.

- These systems have rich networks of feedback
- Some of the feedbacks stabilize characteristics in these systems
- Some feedbacks are controllable

The basis for our investigations is a process model of feedback and control. This model consists of a process element (PE) which applies resources (R) to transform inputs (I) into outputs (O). If one of the destinations of the output is a controller (C), where output is fed back into the process element, we obtain a general controlled feedback loop (as in Figure 2). We term this general controlled feedback loop a process unit (PU). Process elements can contain process units.

*** *.eps here ***

Our hypothesis is that *a process or process system that satisfies the postulates above can be usefully decomposed into a manageable number of process units.*

A number of important issues arise in the investigation of this hypothesis using the model we have proposed. The first of these issues is that of how to model software evolution processes --- in particular, what do we model and what is the basic unit of modeling.
There are two general approaches we might use: one is to model people and organizations, the other is to model what people and organizations do --- that is, their activities. Choosing *people and organizations* would lead to a decomposition like an organizational chart of a company. For different projects executed by the same company, the charts need not be identical, even when the projects are concurrent. While these organizational charts are useful for some purposes, we think they are not useful here.

We propose, instead, to model the *activities* in software evolution processes. A process element in our model, then, represents an activity performed in evolving a software system. Moreover, this choice represents a focus on the design of the processes and their activities, not their implementation in terms of people, tools, environments and organizations. It should be emphasized that the resulting model may not map readily to a traditional organizational structure --- in particular, the control aspects in the model are associated with the elements they control and not with the parts of the organization which may execute them.

The second issue is that of process element decomposition. Given that activities are both the basic building block and the decomposable building block, how do we structure software evolution processes recursively using our model. A process element may be composed of both process elements and process units. It is not necessary that a productive action have feedback control. It may simply be a activity that produces something necessary for the overall product of the evolution processes. The activities may be composed sequentially or in parallel into a larger unit with or without a controller. The internal structure of a process element then may look like a graph with multiple paths starting with the initial input and resulting in the final output. How the decomposition is arrived at and how far the modeling effort is taken --- that is, how many levels of recursion one has --- is a matter of design choice.

Given that one can recursively decompose the process element of a process unit into a combination of process elements and process units (each with their own controller), the third issue is that of how far a controller can extend its control. Obviously, the controller may affect the parameters that it regulates. These parameters may affect the control of the checking in various process elements (such as how many errors are allowed to be found before rewriting is required) or they may affect the control of the subordinate process units (changing their parameters and thus indirectly changing their range of control). Secondly, the controller may effect a change in the activity structure by selecting a different, but existing, path through the process element. And finally, the controller may modify the internal structure of the process element it controls. These may range from simple changes to the process elements and their interconnections to radical redesigns of the entire activity.

A critical question at this point is the extent to which a controller may effect its control --- how far into the recursive structure can a controller see? Since it is our intent to keep the model as simple as possible and introduce complexity only as it is clear that one cannot do otherwise, we limit that span of control to only one level of nesting. That does not, however, preclude the controller from establishing changes in the controllers it regulates to cause them to carry out desired changes beyond its limits.

The fourth issue is the form and frequency of the output from the process element that is used as input to the controller. A classical feedback control approach would suggest that the output from the process element is discrete and separated in time by whatever delay exists due to the arrival of input and the time to transform that input into output. For sub-elements nested deeply in the hierarchy, this time delay may not be significant, but at the top level of a process that evolves a very large software system the delay may be on the order of months or even a year or two.

In this latter case, the delay means that the controller will be able to effect its regulation only infrequently. In practical terms, however, we see control exercised on a much more frequent basis, especially by such organizations as project and process management. Moreover, we certainly see in our current processes and activities the production of project, process and product information which can be considered output of a sort --- though different from that of the product itself. If we permit this sort of output, we get something more akin to continuous output that can be assessed and evaluated by the process element controller and used to regulate the process in a more timely manner during, rather than between, the transformation of input to output.

This more continuous stream approach raises a side-issue: what determines the extent of visibility of these project, process and product data? We certainly want to avoid an information explosion because that is as poor a data modeling technique as having too many lines of control. In the end, it is the controller that determines what information is needed if its job of regulation is to be effectively

performed. Thus, the information output (other than the product itself) is precisely that required by the controller as necessary to properly regulate its process elements and its
subelements.

A number of extensions to our model, that may be allowed if we find that we cannot properly model our evolution processes without them, suggest themselves at this point.

- Allow a controller to be recursively decomposable into a collection of subunits that together define the controller.
- Allow arbitrary inputto the controller where now the only input is that which is producedby the process element and, indirectly, that from the ruling controller.

As with many software engineering analytic tools, the very act of decomposing a process in a particular fashion may yield substantial dividends, quite apart from any benefits that may accrue from applying subsequent steps. A very important kind of dividend is the listing of a regulators' admissible actions and required inputs. Quite likely one will discover how *badly* defined are the regulators' prerogatives, how arbitrarily they are distributed between various regulators, and how little justification there is for allowing some regulators to do things that are just as groundlessly denied others. If this hunch proves correct, a very concrete improvement to many software evolution processes would be instantly available: the unification of regulators scope under similar (or even more so under identical) stimuli. Translated into shop-floor terms, one would advise giving similar powers to people who control similar activities.
This piece of advice is of course trivial; the difference is that with our decomposition in hand we can flesh out the similar parts of the advice.

The desired result of a fully realized multi-level control model is the identification of controllers, their settings and their predicted results so that the well-regulated processes so modeled reach a steady-state --- that is, reach a state of stability and predictability.

Influence vs Control

On the basis of the OS/360 phenomenology [leh85] and our model, we undertook various process modeling exercises to explore the various issues in feedback and control. The general result was a paucity of feedback control examples. The most frequently encountered kind of control is that where control changes or redesigns the controlled process element. Examples of changing process or control element parameters --- that is of regulation in the classical sense --- were almost impossible to find. We did, however, find anecdotal evidence of several examples of this type of classical regulation. We discuss these examples in the next section.

Despite the difficulty in identifying predictable control mechanisms, it was clear that there is a wide variety of feedback effects --- that is, feedback control that is implicit and unpredictable rather than explicit and well-defined.

We offer a number of reasons for this state of affairs:

- first, the fields of software engineering (in general) and process engineering (in particular) are relatively immature;
- second, there may well be feedback overload in which the various feedback paths interact in unknown ways and hinder the understanding of individual feedback and control mechanisms;
- third, process changes as a result of control tend to be step functions, not regulation; and
- fourth, classical feedback control mechanisms are generally applied to production. Their applicability to design processes such as software production and evolution processes have not been widely studied.

Immaturity

As a field, software engineering is relatively young and as a subfield of software engineering, process engineering is very young. One might characterize most software evolution processes as in the ``chitty, chitty, bang, bang'' stage --- that is, the entire enterprise is just barely held together and all the effort goes just keep the enterprise afloat. As such, the evolution processes are workable, but only just, and all the time is spent tuning and repairing the enterprise with no resources left for more formal feedback and control mechanisms to be put in place.

While the previous description may be somewhat of a caricature, it is undeniable that we have little theory for software evolution processes, process improvement, or even of process systems and their architectures [per94, per95]. Because of this lack of theory, we do not know what controls are available, and if we do, we do not know (or know very little of) what their settings are and what effects they have.

Clearly, we need research to establish appropriate theories from which to derive necessary control mechanisms and experimentation to establish their settings and effects.

Feedback Overload

A basic result in linear theory may provide an explanation of why there is a lack of readily discernible ``control knobs'' in software evolution processes.[3] While these processes are not linear systems, the analogy is a reasonable initial approximation.

If a systems' open-loop transfer function (that is, with no feedback) is A and a fraction of the system output b is fed back (negatively) to the input, the the system's closed-loop transfer function is $A^* = A/(1+bA)$. Thus, in a system where there is a significant amount of feedback, A^* approaches 1 and the transfer function tends to be merely a function of b more or less independent of A.

While this makes it difficult to find the control knobs in the evolution processes, it does have a possible and very interesting side effect: it is possible for intrinsically poor software processes to produce good products because the actual process execution is dominated by the feedback (the set of

[3] This explanation was suggested by Ray Offen, Macquarie University, in an informal discussion about the problems of finding feedback control in software processes

*b*s) and not the basic process. This is observed in practice in software development and evolution processes when a high degree of corrective feedback is supplied by, for example, capable and experienced lead developers or project managers.

Step Functions vs Regulation

One of the means of regulation was that of changing or redesigning the controlled processes. It is this category of control that we found most often in our explorations. Almost uniformly, however, these process changes represented step functions rather than control knobs by which a process could be regulated --- that is, they change the process (often significantly) by improving one or more of its aspects, not by providing a means of regulation.

Watts Humphrey's Personal Software Process (PSP) [hum95] provides one such example. PSP is introduced in a series of steps where each step concentrates on a particular aspect of the personal process. A fundamental part of the process is measurement of key process factors which provide feedback to the person executing the process. The key to the personal process is *personal defect management* in which the *yield* measure is the most important: yield is the percentage of defects found and fixed before compilation and testing.

In teaching PSP, the students are given a set of 10 programs which are developed sequentially as different parts of PSP are introduced. Design and code reviews are introduced just prior to exercise 7. This introduction represents *the* major change in the evolution of PSP from introduction to a fully fleshed out process. While there is a slight increase in yield between exercise 1 and exercise 6, there is a significant jump after the introduction of reviews, causing the average yield to change from about 8\% to about 50\% and thus representing a significant process improvement. After that through the end of the exercises, there again is some slight improvement.

This introduction of design and code reviews represents a step function that improves the process; it does not represent a turnable knob that enables one to regulate PSP.

Still, this use of step functions is not so different from what happens in other fields -- for example, economics --- and we see progress in these fields by applying principles of feedback control.

Design vs Production

While we often talk about software production as design and implementation where implementation refers principally to code *production*, it is a key insight to understanding software development and evolution processes that the entire design and coding processes are actually design processes; they are not manufacturing or production processes. Code represents simultaneously the lowest level of design and the beginning of construction. Building a software system is like building a new unique bridge. The notions of control are as difficult to express there as they are in software processes. How does one apply feedback to a creative process?

Where, then, is the production part of software processes? It is in the compilation and linkage of the component parts -- that is, in the production of an executable version of the system. This production part is, however, entirely automated and not very interesting from a feedback control standpoint.

As an analogy to software evolution, consider the evolution of an particular brand of automobile. That evolution is not in the production line. For individual instances of cars (the ones we own), evolution takes place in the repair shop. But, evolution of a particular car line takes place in the design laboratory. The governor aspect of control is not applicable at the design level and feedback at the production level is orthogonal to design and evolution. The controllers are changed for assembly and production as a result of design changes. However, evolution of the product line cannot be explained by *its* controllers, nor reduced to *its* controllers.

This analysis suggests that the manufacturing and production process with all the feedback controllers that go along with it are of little direct interest to studying the software process. Instead of production and manufacturing we have invention. Phenomenological data, however, suggests otherwise and this is something that requires further investigation

A first insight indicates that, for example, feedback plays a significant role in the coding and testing part of the processes. One reason for this is that coding and testing are where we are closest to the non-creative aspects of the process. Moreover, at a certain distance of abstraction, we can view the design and coding processes as the transformation of input specifications to output products. It is this latter view, of course, that suggests the utility of feedback control principles.

Examples of Feedback Control

As mentioned above, the most common kind of feedback control we found in our explorations was that which led to changes in the process. Votta and Zajac [vot95] describe this type of example in their study of design process waivers. In the evolution of their large-scale, real-time system, features are the unit of work. These range in size from several lines of code to multiple thousands of lines of code.
The same design process is used for all features. However, for the smaller ones, waivers may be submitted to omit various parts of the process, which while appropriate for large features are inappropriate for smaller ones.

Votta and Zajac acted as the control element of the process and collected a large set of waivers over a period of time. After collecting these waivers (as outputs from the design process), they evaluated the various requests and assessed their merits. The result
was a control action to define three separate paths of design dependent on the estimated size of the feature --- each class of features would have a design process appropriately scaled to their needs as determined by their size.

In this example, the controller changed the process by introducing several extra paths through the internal process elements and process units that are governed by a size switch. While it is an example of feedback control, the control itself is as creative in its effects as the design process it regulates.

In our search for the more classical regulating control, we have anecdotal evidence of one such control in the use of code reviews. While we do not have documentation, it is a case that is entirely plausible. The regulation works as follows: if there are too many errors in the code units being produced, extra code reviews are introduced to reduce the number of errors; if time is critical and the number of errors is sufficiently low, code reviews are removed from the process to speed up the process at the expense of an increase in errors.

We are currently exploring this case further to document it and to determine its utility as a regulation form of control.

Summary

Our work over the past year has been primarily a philosophical or intellectual exploration of the problems of applying feedback control principles to software evolution processes. This exploration has been based on our combined industrial and research experience[4] and that of the various participants in the FEAST Workshops. As we have noted in the discussions in the preceding sections, we have phenomenological evidence that classical feedback control is at work in the evolution of software systems and that there is a significant amount of feedback present in these processes. While it is apparent that these various feedback paths have a variety of effects, our explorations have yielded little that can be counted as predictable control. It will require extensive scrutiny and modeling of current industrial processes to determine the actual impact of feedback and control.

Meanwhile, developers continue to build and evolve software systems. They continue to make progress in their understanding of those processes though there is little record of the their investigation of feedback paths and its impact.

Our current research agenda is focused on exploiting practical, real world experience as the basis for understanding and delineating feedback control in software evolution processes: noticing correlations among feedback and effects, finding patterns in feedback phenomena, and performing engineering and scientific experimentation to determine both useful control effects and their underlying mechanisms.

We have a three pronged approach: collecting and analysing system evolution phenomena, applying systems dynamics modeling, and experimenting with feedback controls.

The phenomenology of system evolution was one of the starting points for our research, a phenomenology of a 1960's operating system. There are questions as to how relevant that phenomenology is today: perhaps it was the result of the specific application, or perhaps the result of the specific environment or organization. However, our intuition is that it is the phenomena of large systems' evolution and independent of the time, application and environment. The very first data on a

[4] Each of the authors has had extensive system development experience
in industrial and commercial settings as well as academic and industrial research experience in software and process engineering

1990s system that we have just begun to study appears to confirm the earlier observations and support our intuitions.

To understand the complexity of feedback paths, control and their interactions, we plan to create systems dynamics models of several currently used evolution processes. In this way we will be able to validate the models with current project data and gain insights into the various feedback phenomena that are at work in building and evolving software systems. Our industrial partners are the source of these processes and data.

Insights that we have gained into feedback control phenomena will be confirmed and explored further by means of both engineering and scientific experiments. In this way, we expect to determine the identifiable impact of feedback controls and determine their range of effects.

Thus we hope to extend the science of software evolution and develop methods, techniques and tools to aid both system evolution and process improvement.

Acknowledgements

We wish to thank the various participants in the FEAST workshops and our industrial partners for their insights, questions and discussions on the various aspects of feedback and control phenomena.

References

[adb91 T. Abdel-Hamid and S. E. Madnick, **Software Project Dynamics -- An Integrated Approach**. Prentice Hall, 1991.

[bel72] L. A. Belady and M. M. Lehman, An Introduction to Program Growth Dynamics. Statistical Computer Performance Evaluation, Academic Press, 1972. 503-511.

[bro86] F. P. Brooks, *No Silver Bullet -- Essence and Accidents of Software Engineering.* **Proceedings of the IFIP Congress 1986**, Dublin, Ireland, 1069-1076 (September 1986, Elsevier Science Publishers.

[fea1] **FEAST Project --- Preprints: FEAST Workshop I**, Imperial College, London UK, 16-17 June 1994

[fea2] **FEAST Project --- Preprints: FEAST Workshop II**, Imperial College, London UK, 24-25 October 1994.

[fea3] **FEAST Project --- Preprints: FEAST Workshop III**, Imperial College, London UK, 28 February - 1 March 1995.

[hum95] Watts Humphrey, *The Power of Personal Data.* Software Engineering Institute, 1995.

[leh69] M. M. Lehman, *The Programming Process.* In [leh85], 39-83, 1969.

[leh85] M. M. Lehman and L. Belady, **Program Evolution -- Processes of Software Change.** Academic Press, 1985

[leh87] M. M. Lehman, *Process Models, Process Programs, Programming Support -- Invited Response to a Keynote Address by Lee Osterweil.* **Proceedings of the 9th International Conference on Software Engineering**, Monterey CA USA 14-16 (March/April, 1987)

[leh95] M. M. Lehman, *Software Process Improvement --- The Way Forward.* **Proceedings CAiSE 95**, LNCS, Springer Verlag, 1-11 (June 1995)

[nau68] P. Nauer and B. Randall, **Software Engineering -- Report on a Conference, Sponsored by the NATO Science Committee**. Scientific Affairs Division, NATO, Brussells, Garmisch 1968.

[ost87] L. J. Osterweil, *Software Processes are Software Too.* **Proceedings of the 9th International Conference on Software Engineering**, Monterey CA USA, 2-13 (March/April 1987)

[per94] Dewayne E. Perry, *Issues in Process Architecture.* **Proceedings of the 9th International Software Process Workshop**, Airlie VA, 138-140 (October 1994)

[per95] David C. Carr, Ashok Dandekar and Dewayne E. Perry, *Experiments in Process Interface De**scriptions, Visualizations and Analyses.*** Software Process Technology: EWSPT'95, Noordwijkerhout, The Netherlands, 119-137 (April 1995)

[tur86] W. M. Turski, *And No Philosophers Stone Either.* **Proceedings of the IFIP Congress 1986**, Dublin, Ireland, 1077-1080 (September 1986, Elsevier Science Publishers)

[vot95] L. G. Votta and M. L. Zajac, Design Process Improvement Case Study Using Process Waiver Data. Software Engineering -- ESEC'95, Sitges, Spain, 44-58 (September 1995.