First Dagstuhl Seminar on

Future Directions in Software Engineering

February 17 – 21, 1992, Schloß Dagstuhl

Walter F. Tichy
Institut für Programmstrukturen und Datenorganisation
Universität Karlsruhe
W–7500 Karlsruhe, Germany

Nico Habermann
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA

Lutz Prechelt
Institut für Programmstrukturen und Datenorganisation
Universität Karlsruhe
W–7500 Karlsruhe, Germany

List of participants:

Richards Adrion, U. of Massachusettes, Amherst, Massachusettes
David Barstow, Schlumberger Laboratory, Montrouse Cedex, France
Veronique Donzeau-Gouge, CNAM, Paris, France
Phyllis Frankl, Polytechnic Univ., New York
David Garlan, CMU, Pittsburgh, Pennsylvania
Morven Gentleman, National Research Council, Ottawa, Canada
A. Nico Habermann, CMU, Pittsburgh, Pennsylvania
Dan Hoffman, U of Victoria, Canada
Stefan Jaehnichen, TU Berlin, Germany
Gilles Kahn, INRIA Sophia Antipolis, France
Gail Kaiser, Columbia U., New York
Bernard Lang, INRIA, Le Chesney, France
Wei Li, U des Saarlandes, Saarbrücken, Germany
Hausi Müller, U of Victoria, Canada
Axel Mahler, TU Berlin, Germany
Roland Mittermair, U Klagenfurt, Austria
Manfred Nagl, RWTH Aachen, Germany
David Notkin, U of Washington, Seattle, Washington
Harold Ossher, IBM, Yorktown Heights, New York
Dewayne Perry, AT&T Bell Labs, Murray Hill, New Jersey
Erhard Ploedereder, Tartan Inc., Monroeville, Pennsylvania
Lutz Prechelt, U Karlsruhe, Germany
Dieter Rombach, U Kaiserslautern, Germany
William Schaefer, U Dortmund
Gregor Snelting, U Braunschweig, Germany
Walter Tichy, U Karlsruhe, Germany
David Wile, U of Southern California, Marina del Rey, California
Michal Young, Purdue University, West Lafayette, Indiana

Contents

The individual session reports consist of the abstracts of the talks followed by a short and extremely informal transcription of the discussion in that session (protocoled by Nico Habermann).

Extract from the invitation letter sent by Walter Tichy:

Some time ago, you should have received an invitation to a workshop on "Future Directions in Software Engineering", to take place in Schloss Dagstuhl, Feb. 17 to 21, 1992. I'm writing this letter to take the mystery out of this invitation, and perhaps to convince you to come, should you need convincing.

Schloss Dagstuhl is a newly established conference facility for conducting intensive, week-long workshops on topics in Computer Science. The facility is subsidized by the German government and industry, so the cost is quite low (about $85 for room and board for a week, including conference fee). Workshops can be completely informal, so no call for papers, proceedings, or even a formal program are necessary. For those of you who have been to the famous Oberwolfach facility: Dagstuhl is (or should become) to Computer Science what Oberwolfach is to Mathematics.

The workshop on "Future Directions in Software Engineering" is intended to bring together some of the leading scientists in the software area for discussing future directions. Nico and I would call the workshop successful if we can identify the ten most pressing problems in software engineering today and outline attacks on these problems. With your help, we might create a meeting-of-the-minds similar to the first NATO conference on Software Engineering in 1968.

What would help us in preparing a schedule is if you sent me a position statement outlining your view of the most important problems in software engineering, and what one should do about them. Note that a natural tendency is to call the most important problem whatever one happens to be working on. So let me encourage you to think about areas that are not on your current research agenda. Here is my attempt at a partial list of the most important problems:

- *mastering the development and maintenance of* 10^6 *LOC*
- *programming 10000 processors at once*
- 10^9 *hours MTBF for safety critical applications*
- *managing the human interface*
- *education*
- *raising the level of reuse to 50%*

1 Session "Methodology"

Research Methodology in Software Engineering

W. Richards Adrion,
University of Massachusettes, Amherst, Massachusetts

We continue to struggle with a community consensus on appropriate research methodology. Without such a consensus, there remains a problem in assessing papers, research proposals, and research programs. We discuss a recent paper by Cohen (AI magazine 12(1), 1991, pp 16–41) which analyzed research methods in AI. Cohen concludes that there are two methodologies in use in AI (system-centered and model-centered) and that neither alone is sufficient to meet the goals of AI research. The lessons to be learned for SE research include to carefully specify the problem being addressed and the assumptions and "environment"; state hypotheses; provide methods used to address the problem; validate the hypotheses; and suggest or indicate what changes or modifications are necessary for future research.

In an NSF-sponsored workshop held in 1989, four methodologies were identified. The *scientific method*: observe the world, propose a model or theory of behavior, measure and analyze, validate hypotheses of the model or theory, and if possible repeat; the *engineering method* (evolutionary paradigm): observe existing solutions, propose better solutions, build or develop, measure and analyze, repeat until no further improvements are possible; the *empirical method* (revolutionary paradigm): propose a model, develop statistical or other methods, apply to case studies, measure and analyze, validate the model, repeat; the *analytical method*: propose a formal theory or set of axioms, develop a theory, derive results and if possible compare with empirical observations.

How these methodologies might be employed in SE research is discussed.

Evaluation of Software Engineering Techniques

Phyllis G. Frankl
Polytechnic University, Brooklyn, New York

While many Software Engineering techniques have been proposed, relatively little solid evidence has been gathered to indicate which of these are effective. New techniques are often justified only by appeals to intuition. Sometimes anecdotal evidence indicating the effectiveness of a technique is gathered through case studies.

Ideally, new techniques should be justified through rigorous analysis or controlled experiments, but it is often very difficult to do this. It is sometimes possible to facilitate experimentation by introducing some abstract modeling of the technique being investigated. For example, experimental investigation of the fault detecting ability of software-test-data-adequacy criteria is facilitated by using randomly generated test sets, while in reality test data adequacy criteria are usually used with test sets which have been generated in some non-random way. However, it is then necessary to validate that the "real world" behaves similarly to the abstract model. Alternatively, it is sometimes possible to change the real world (i.e. the SE technique) to conform to an analyzable model. Development of (informal) standards of experiment designs for various particular problems would help make

evaluation of techniques more scientific. Individual experiments could then provide data points, which, taken cumulatively, provide more compelling evidence than is currently available to indicate which techniques are effective.

We Need To Measure The Quality Of Our Work

Gail E. Kaiser
Columbia University, New York City

There is a gaping hole in current academic-style software engineering research: We as a community have no generally accepted methods or benchmarks for measuring and comparing the quality and utility of our research results.

The only metrics for environments research are the evaluation methodology promoted by the Software Engineering Institute and the International Software Process Workshop example problem for process-centered environments.

I see even fewer options for measuring the quality and determining the scalability of research prototypes for parallel and/or distributed applications, programming languages, operating systems, databases and software engineering techniques.

In addition, a perhaps fatal problem is that academic researchers are still making incremental improvements on 15-year-old tools such as make and RCS available on our educational computers rather than starting from the state-of-the-practice available to industry.

Alternatives to Quantitative Evaluation

Michal Young
Purdue University, West Lafayette, Indiana

Evaluation is an important problem in our field, but we should not succumb to physics envy and insist always on *quantitative* evaluation. Quantitative evaluation should be encouraged where it is appropriate, but alternatives should also be developed.

If software engineering papers are becoming less credible (as I believe they are), it is not because current papers have fewer quantitative evaluations. I would not find Parnas' *On the Criteria for Decomposing Systems into Modules* more convincing if it reported that 100 programmers had implemented KWIC indices in each fashion, and 80% achieved greater reuse with the new approach. I would not learn more from Liskov and Zilles' survey of abstract data type specifications if they reported that algebraic specifications were on average 30% less likely than model-based specifications to overconstrain an implementation.

One alternative to quantitative evaluation is development of approaches and techniques in the context of systems research, so that the examples offered as evidence are more credible. This was a strength of early work in software engineering. One may fear that mixing software engineering research with systems research will limit novelty and discourage long-term thinking, but more often challenging systems problems force reevaluation of accepted ideas and suggest unorthodox approaches. As examples, consider the impact of object-oriented programming and the growing importance of event-based coordination for constructing large systems, both of which grew out of systems research.

Another alternative is the "challenge problem" paradigm, in which research communities agree on representative and appropriately difficult problems to which a variety of approaches or techniques can be applied. Even in the absence of measurement, comparison of solutions to a single problem improves our understanding of their relative merit.

```
Rick Adrion, Methodology for doing research.
Observations:
    1) model-centered, without serious experimentation to validate model;
    2) empirical, descriptive, lacking justification and evaluation.
Giles: our products are so temporary, models valid today are out of date
       tomorrow. Industry tells us academics we are behind and can't compete
       with the products they are able to develop and market.
Nico:  We lack a tradition of doing research in comparison to disciplines
       such as physics, astronomy and biology.
Wile:  Engineering must be based on science. Don't confuse engineering with
       tinkering.
Dewayne: Problem of getting people in production shops to participate in
       an experiment.
Rick:  We don't specify our experiments precisely enough and in sufficient
      detail for others to repeat the experiment.
Bernard: CompSci&Eng has close ties with industry (not so for physics and
       other disciplines); they want us to work on their problems.

Phyllis Frankl, Testing.
Goal is either 'detect errors' or 'measure reliability'; Phyllis talk about
the first, not the second.
Observation: people design methods, but do little validation.
Look for "abstract model of reality": which model catches the most errors?
Example: branch testing versus data flow testing, which is more effective?
Wish: standardization of how to do experiments.
Dieter: testing is often misused, e.g., apply to detect errors, but used
       to make claims about reliability.
Giles: what if the design of the test software is of the same order of
       magnitude as the software to be tested?
Testing might be an additional topic for separate discussion.

Gail Kaiser, Measurements.
What to measure? Productivity, functionality, overhead, complexity, reliability
Giles: difference between systems is hard to measure. Request for "hard data"
       by industry is an excuse for them to use your words to support their
       decision not to use your stuff.
Nico:  you don't want to measure on someone else's conditions, but you want to
       measure for your own sake on your own conditions, e.g. performance
       improvement, memory use, etc.
Giles: but you don't want to measure the industrial notion of 'productivity'
Garlan: indeed, these measures often overlook what is really important. E.g.,
       coding time or code length may be reduced, but maintenance may increase.
Erhard: what really matters is how your measurements change over time.
Wile:  if you have to measure, you don't understand the thing you measure.
Bernard: examples of things to measure: space, time, frequency, . .
Dewayne: must give line manager an argument to introduce a new tool.
Notkin: is that our task?
Bernard: two kinds of measurements: product and development process.
Gail:  Productivity, reliability, portability, team collaboration, scalability
       are things we should measure, but do not know how to yet.
Rick:  difficult, because disconnect between interest groups, e.g.
       real time research, practice and software engineering.
Many:  we must accept the idea that systems may be designed with permissible
       failure in mind (compare telephone system).
Gail:  complaint from industry: acad research works with outdated tools and
       cannot compete with industry in making usable tools.

Michael Young, wearing my anti measurement hat.
Observation: we need particular challenges in the sense of: can you do this
and/or can you do that?
All:  issue of intellectual property rights. (deserves separate discussion)
Wile:  Opposite of earlier statement is also true: if you don't understand,
       you should measure. (This sounds close to the 'empirical research
       approach of Rick this morning)
Giles: we do (and should) measure what we believe is necessary, but we must
       recognize that there are things we cannot measure (in numbers) and
       things we need not measure.
```

2 Session "Industrial Practice"

Issues for Software Engineering

David Barstow
Schlumberger Laboratory for Computer Science, Paris, France

Five major issues that must be addressed by the software engineering community are:

(1) There is not yet a firm scientific basis for software/hardware systems. Developing such a basis will require decades of theoretical and practical work, including such topics as domain analysis, formal methods, and techniques for describing system architectures and distributed systems.

(2) There is not yet a well characterized and widely used set of "standard software engineering practices". Such a set could be developed from techniques that are already well known, and the SEI maturity level model and the ISO 9001 standard represent good progress. Competitive pressures in the commercial world are likely to increase the use of good practice: those groups that do not use good practice will probably fail financially.

(3) There is a very large base of installed software that is becoming increasingly unmaintainable. Either we will succeed in replacing the code through restructuring and reverse engineering, or the software (and possibly the companies that use it) will die like old dinosaurs.

(4) The currently available systems that allow naive users to develop software (e.g. spreadsheets, HyperCard) offer little or no support for good software engineering practice. Unless this situation changes (which seems unlikely to me) we will probably have to live for many years with a large body of badly written small programs.

(5) There are not yet good models of the evolutionary processed that software goes through. One important aspect for which models are needed is the interaction between evolutionary change and customization.

Technical Software Engineering Problems of Commercial Software Products

W. Morven Gentleman,
Software Engineering Laboratory,
National Research Council of Canada, Ottawa, Canada

I think of Software Engineering as the use of technological means, such as theories, methodologies, tools, and techniques, to address the practical problems that arise in the development, deployment, maintenance, and evolution of software. There are many different practical issues software engineering can address, and progress on any of them can help industry.

The problems that represent barriers to success are different for different kinds of software, or at least the relative importance of various problems are different. Much of the software engineering literature has addressed the problems of very large custom built systems, or

of system software, or of safety critical systems. I believe we should direct some of our energy into solving the problems of the software products industry, where software is sold in multiple copies that are identical or at least not to be substantially modified by the user.

The milieu in which such software product companies operate is quite different from the large teams with unlimited resources in some of the more commonly studied areas. Problems in this field range from planning an evolution path so that a viable partial product can be shipped to fill a market niche while the full intended functionality has yet to be implemented, packaging a product for automated installation and adaptation, interoperability with products independently obtained from other suppliers, and coping with product inadequacies and failures that are revealed by shipped products in the field.

Given the significant and rising economic portion that this is of the software industry as a whole, it is quite surprising how little research has been done on these problems, even when technological solutions seem feasible.

Industrial Practice $\ll$ State of the Art

Daniel Hoffman
University of Victoria, Canada

Richard Hamming proposes two criteria for important research problems: (1) the solution must offer substantial benefits and (2) there must be a reasonable attack: an approach likely to lead to a solution. It is the attack that distinguishes important problems from wishful thinking. We consider three important problems in software engineering and their attacks.

Too little specification. Today's software is not developed "to specification". While specifications are written, they are imprecise and incomplete. Compliance is not systematically enforced and is therefore not achieved. Decisions that should be made by the best designers are made instead by less-skilled implementors and maintainers. Substantial improvements are possible, with existing specification methods and unsophisticated mathematics. Object specification should be attacked first, because it is simpler and better understood than requirements specification.

Too little mathematics. There is disturbingly little use of mathematics in industrial software development. The ad hoc methods used instead are expensive and ineffective. We see three attacks on this problem. First, select a set of mathematical techniques that is small, easily mastered, and suitable for frequent application. Second, choose the right balance between formality and informality. The academic over-emphasis on formality has stifled the industrial use of mathematics in software development. Third, adopt a proof-based approach in software reviews. These proofs may be formal or informal, and are presented by software developers to their peers. Tailor the software documentation to support the proofs.

Too much old code. The installed base is huge, in poor condition, and extremely expensive to maintain. Successful reengineering will be based on three principles. First, apply the reengineering changes in small increments; wholesale replacement is rarely an option. Second, apply the changes to code already being modified, for error removal or enhancement. In this way, the additional cost of the reengineering will be small, because the code

must be read, tested, and installed anyway. Third, from the start, establish the target: specification methods, code standards, etc. Otherwise, the reengineered code is likely to have the same problems as the old code.

```
Dave Barstow, Industrial Practice
Observation: software development in industry is client (and profit) driven
Picked four topics:
    1) practice of software engineering (SE): scientific basis is lacking
    2) need applied science leading to sound engineering practice
    3) judge products, tools, techniques, etc. in economic context
    4) socio-economic: large install base gets worse because of maintenance
    5) SE by the masses: BASIC, Hypercard, etc. support the individual, but
       no support for team or people working together.
Wile: good engineering needs sci basis, but is not being developed by
       researchers
Giles: why should researchers change their agenda?
Nico: no change required, but there is room for people developing this
       sci base comparable to chem eng using chem or mech eng using physics
Barstow: need to apply cost model (not routinely taught in univ??) no
Garlan: cost model often seriously flawed: important parameters not included,
       cost often amortized over many products and projects.
Wile: reverse engineering necessary to capture good ideas in old code.
Erhard: what is research doing for maintaining and evolving new software?
       does research work at least on tools for this purpose?
Nico:  but we develop new languages, new concepts, tool ideas; this contributes
       to basis for supporting the evolution of software.
Barstow: what to do with Europe > 1992? (Industry wants to serve
       European community)
       uniform credit card payment? telephone? gas station service? ECU?


Gentelman, Industrial perspective
Observation: <1990 emphasis on large systems for large companies.
    2) sizable market: > $60B
    3) large number of small companies: > 12,000
    4) product must evolve in order to stay in market
      a. growing sophistication of users
      b. new companies that do a better job (Persuasion > Powerpoint)
    5) great turnover in personnel: 30%; not known whether this applies to
       all, or to the 1/8 = programmers, or to sales staff.
    6) interoperability is crucial for market share
    7) feedback from customer and customer satisfaction must be collected
Barstow: resources must be applied at the right moment: late-to-market
       implies great loss in market share.
Walter: list of issues: but what solutions do you propose?
Gentelman: SE was looked upon as a management issue for a long time.
       better: develop technology for problems you have chance of solving.
Dieter: but what kind of research would industry like to see?
answer:  1) teach systems and not programming from a blank piece of paper
         2) how to get data back from users
         3) how to deal with flexible customization
         4) reverse engineering and support for system evolution
Barstow: add to this list
         5) system architectures usable by many
         6) domain modelling
         7) formal methods for being precise in specs
         8) think in terms of distributed systems.

Hoffman, Gap between software research and practice
Observation: Humphrey has alerted us to organizational aspects precondition
       for success.
Hamming: how to do important research? 1) result must have effect 2) outline
       an attack (don't undertake to solve the perpetual motion problem)
compliance: the specs that exist must be obeyed as accurately as possible;
       (result of a heated discussion)
       extensions or changes in specs must be discussed with all parties
       involved and not be implemented on initiative of single individual
Look for mathematical tools for software development description.
Garlan: not black/white choice: mixture of formal & informal is possible (z)
Frankl: specs can be for different audience: implementors, customers, users,
       automatic generation of test cases, etc.
Axel:  not all problems lend themselves for mathematization
Bernard: disagree: then domain must look for math specification.
What to do with old code for systems that are still heavily used and operate
on large volumes of important data.
Hoffman: restructure and 'picky-bag' change on the fly
Erhard:  disagree; practice shows people make a great mess of it. Alternative:
         first restructure and then change.
in favor of simultaneous approach: code is understood only once
         while being analyzed and restructured;
against: mess, and better overview of the whole before
         understanding change that is effective.
question: is incremental change of old code possible? (no answer provided).
```

3 Session "Modeling and Design"

The Need for a Science of Software Architectures

David Garlan
Carnegie Mellon University, Pittsburgh, Pennsylvania

As the size of software systems increases, the design problem goes beyond the algorithms and data structures of the computation. Designing and specifying the overall system structure emerges as a new kind of challenge. Currently good software designers develop large-scale systems using structuring techniques that have evolved informally over the years. But these architectural paradigms are understood only informally and idiomatically. As a consequence, we see the proliferation of systems that have many of the same architectural properties but are difficult to compare with any precision. Moreover, when designing a new system it is almost impossible to to make principled decisions about which architecture to use for a given application.

The development of a scientific basis for software architecture will enable new systems to be built, compared, and analyzed in rigorous ways. It will allow engineers to more fully exploit existing designs. It will enable us to document and understand systems far better than we now can. It will enable us to produce effective tools and will form the foundation for a new discipline of large-scale system design. To achieve this goal, research is needed in five areas: Notations, formal models, tools and environments, domain-specific architectures, and education.

How to See the Forest Behind the Big Tree

Roland Mittermair, Universität Klagenfurt, Austria

Structured, implying *tree-structured* and *Top-Down* has helped Software Engineering to overcome so many problems during the last 20 to 25 years that it is hard to accept that this notion constitutes a barrier for the future development of the field.

We all agree that the challenges of building a 10^2xyz system and the challenges of building a 10^4xyz or 10^6xyz system are uncomparable (xyz is LOC, DSI, n, or any other measure). Nevertheless, we act as if $10^6 = 10^{n+1}$ and try to approach the new challenges with the old cures.

Hence, my questions to our discipline: (1) how to solve $10^6 = 10^{3^2}$ in software development; (2) when is $10^6 = 10^{3^2}$, when is it $10^{2^2} \cdot 10^2$, and when is it $(10^2 + 10^2 + \ldots 10^2)^3$; (3) what is the notion of "approximately" in Software Engineering; (4) how to express "negotiability" in designs formally; (5) how to design, specify, and maintain software in manageable pieces for cooperation in a huge system; (6) how to maintain operating old software, integrate it with software created according to new paradigms, and bring the whole system gradually up to the state of the art; (7) how to integrate software developed according to different paradigms in a neat (and efficient) way; (8) how to live with or integrate software that satisfies its specification "most of the time"; (9) how to preach the new gospel, containing more formality, to the lovers of the quick fix & the dirty hack.

In summary: How to deal with forests of mixed nature, such that each tree "naturally" supports the others ? What is a "federated software system", what questions and what answers are provided by such a concept ?

The State of the Art of Software Engineering:: Analysis, Suggestions, and Steps Towards a Solution

Manfred Nagl
RWTH Aachen, Germany

The state of the art in software engineering, even for conventional software, is rather limited. We still do hardly understand what software is, what properties software should have, and how it should be developed, maintained, and reused. Especially for the most risky areas of software engineering like requirements engineering, architecture modelling, and project planning we do not have ready and applicable solutions. The state of the art in industry is more craftsmanship than engineering based on a scientific discipline.

The problem, in my opinion, besides the implications of immateriality of software is that software problems have to be specialized in order to get deep solutions. At the moment, software engineering is the claim of a general engineering discipline. Software engineering has to be split into (1) an application area technology (business software, system software, process engineering software, etc.), and into (2) a system structure technology (batch systems, dialog systems, distributed systems, etc.).

Having these two (or more) dimensions in mind, for different working areas (requirements engineering, architecture modelling, etc.), for software process, and configuration control tailored concepts, languages, methods, and tools have to be developed. To demonstrate this, a quality compiler (system software, batch system) was taken as an example. This proceeding of carefully studying relevant representatives of software systems will produce a quantum jump with respect to (a) quality of software and its realization effort, and (b) quality, applicability of suitable means to produce and maintain this software.

Our group has made some modest progress with respect to the above approach. We have carefully studied the representative "software development environments" (IPSEN project) belonging to intelligent dialog systems, and we have developed special concepts, languages, and tools for developing software development environments. As a basis for such specializations we worked on requirements engineering, architecture modeling, and configuration control in general. Process control is under investigation.

Research Challenges from Reuse Technology

Erhard Ploedereder
Tartan Inc., Monroeville, Pennsylvania

In the world of compilation environments, reuse and recombination of large software components to form new products is an economic necessity. Only through those methods, compilation environments can be constructed and maintained that each consist of 10^5 to 10^6 LOC. Extrapolating from this experience, it appears that significant leverage can be obtained from domain-specific analysis in order to obtain generic system architectures

and to identify reusable components for the respective application domain. Somewhat surprisingly, there are no commonly agreed formalisms to describe system architectures, enabling wider accessibility of architectural experience.

Reuse poses significant issues for configuration management, as its true economic leverage is achieved only if instances of shared components share in common maintenance. Organizational structures in a company have a fundamental impact on the ability to reuse and share the maintenance of reused components. Also, they emphasize different aspects of configuration management. For example, an organization congruent to the functional components of products will facilitate their version control; yet the integration of the components becomes rather difficult. Alternatively, an organization structured along product lines will generally ease the integration of reused components, but makes their version control much more difficult, since temporary deviations from baselined components are implicitly encouraged. Finding the right organizational structures and the right configuration management approach to support large-scale reuse is a most challenging problem. Thus, organizational aspects exert an important influence and need to be taken into account in research.

Today's university graduates may be good programmers and passable designers on small-scale problems, but often are completely unprepared to integrate their designs into larger system architectures. What is needed is educational exposure to large system designs, taught by examples and critical evaluation of good and bad architectural decisions. Yet, such examples, and notations to describe them in, seem to be sorely lacking. Egocentric designs ("I solve it all in my limited area of responsibility in the overall system") are encouraged by the educational system, but are actively detrimental in commercial application generation. An appreciation of "design-in-the-many" approaches and problems needs to be created. While programming-in-the-many issues have been researched extensively — but are not a big problem in a highly-structured system architecture — design-in-the-many deserves more attention by the research community.

```
Dave Garlan, Software Architecture
Observations: it takes several iterations to come up with a satisfactory
system description;
What matters is agreement on the architecture;
Great advantage in time and cost resulted for new generations of
products from the suitable system architecture and its formal description;
Nagl: surprising that your architecture has such a procedural flavor.
Notkin: it is clear that abstract data types are not the total answer to
       describing systems.
Nico: what is needed in addition to functions and data types?
answer: protocols describing the connection and interaction of objects
Hoffman: why did you show us pictures and not Z-code?
answer: because the idea underlying the architecture can be better described
       that way; the Z-code serves the purpose of being precise and as such
       is the right interaction vehicle for designers and implementors.
Observation: one should expect architectures to be fairly domain specific.
Notkin: but can't you abstract to a higher level of specs for general
       architectures? (no answer)
Young: do you have to reason (from scratch) when you change the topology?
answer: no, you understand in advance what kind of changes affect properties
       such as deadlock-free, etc.
Wile: do the engineers still use Z after your departure? Do they express
       their design changes this way?
answer: (vague?) the specific is not so important; it is important that they
       use a formal expression and vocabulary.

Mittermeier, how to see the FOREST behind the TREE
SE issues:
    1) how to introduce a scientific method into SE; (foremost: decomposition)
    2) how to maintain a manageable collection of software components;
    3) how to maintain old software and integrate it with new
    4) must allow tolerance in specifications
    5) conceptual modeling must include a dynamic view
Nico: is your conceptual model basically the same as Garlan's architecture?
answer: yes
```

Garlan: essential is that you are willing to represent more than nature of
objects, in particular the connection between objects.
Dewayne: these connections are described by constraints (really enough?)

Nagl, State of the Art in SE
Observations:
1) no fixed set of concepts and/or paradigms
2) Software Engineering = Configuration Management
Notkin: do you really mean 'not fixed' or do you mean 'not shared'
answer: yes, not fixed; paradigms shift all the time
Garlan: should we look for a single framework or model for CM?
answer: no, a combination of (accepted and generally applied) methods&tools
Nico: I would not like to do your CM without automation
answer: practice does not use automated CM systems; this is reality.
Walter: I challenge the statement CM = SE
Notkin: it is not correct to say that there are no tools and languages for CM
Nico: some people are looking for the unique right way to do CM, but I
count myself in with those who believe CM is project and company
dependent: thus CM becomes a programming problem for which you
need good tools and languages (and a lot of automation).
Walter: can you automate a lot of CM? answer: yes
Bernard: to make an existing CM description usable for a new programmer
there must be a language to make it operational and a formalism
to make it precise and meaningful.
Nagl: we are missing an application technology. (strongly denied by many)
all: we must help application writers to design good languages and tools
for their (domain-specific) work.
many: the application writers must come to us and ask for help
(I don't agree; in my opinion we should not start blaming each other)
Nagl: use your whole setup for your existing compiler's CM for the new
compiler project, whether for a new language, a new system or both.
Gentelman: but people do exactly that and the flaw is that it is often
entirely inappropriate.
Nagl: don't use buzzwords and new paradigms, but use established standards.
Bernard: obj.orient is a lot of words, but is used as an unacceptable way of
expressing sloppy semantics. This an example of lack of maturity
Walter: your CM requires incredible manpower; people should not be burdened
with this kind of work and brain overload.
Nico: that is why I said you can't do this without automation.
Bernard: you can get a lot of reuse from your specialized language that
incorporates frequently used functions and structures.
Notkin: you seem to have a negative view on a multi-paradigm; it may be good
to explore new ways and be flexible about what tools or methods to add.
Axel: what is meant is: don't adopt a fad, think twice before you change.

Erhard Ploedereder, Challenges in Reuse and Configuration Management
Observation: my company has a well defined architectural setup (compiler
technology for a family of languages close to and including Ada).
1) cost of maintaining a large collection of small reusable components
is not worth the trouble. Maintenance cost kills the idea.
Hoffman: is this going to change in the future? (No)
Bernard: but function call instruction sequence is an example of an
extremely useful small reusable component.
answer: sure, there are useful small reusable components, but the point is
that maintaining a LARGE number of them does not pay.
2) high-leverage in domain-specific analysis of what might be reusable
Observation: a component is not designed as a reusable component, but is
reusable when it has been reused. (really? can you not design for reusability?)
Observation: greatest value is in shared use (see also Bernard's earlier remark
about specialized languages), because can maintain that component across
multiple usage groups.
Gentelman: but must be willing to retrofit after making an improvement of a
reusable component.
Axel: why not have parameterized reusable modules and have families of reusable
components?
answer: again, the maintenance problem kills you; so, parameterization takes
place when reusable component is reused.
Observation: if you organize by function, reuse ok, but system integration
difficult. If you organize by product, integration ok, but reuse difficult.
Gentleman: these matters are a challenge for education: students need to
develop taste and common sense. Often these are developed by example
from senior people in a company (with potential disastrous result!)
Walter: we don't TRAIN our students in system design.
Bernard: can you teach 'system architectures'? they can be very different
for various domains.
Notkin: but there are commonality in design and in issues such as space/time
trade-offs, I/O, etc.
Erhard: we come up with an isolated solution and forget to ask how this
solution fits in the larger context.

4 Session "Formal Methods"

Another look at Computer-Assisted Proofs

Gilles Kahn
INRIA, Sophia Antipolis, France

We try to understand how to engineer proof assistants better because (1) making proofs may become part of a software development job, (2) proof construction gives new ideas on program development, (3) today's provers have an absolutely atrocious user interface.

The principle of the construction is a client–server approach, which implies the design of a protocol to communicate with existing provers (data, control) and an open-ended user interface architecture (based on events). The building of a proof is a parallel task where the theorem database is used as a synchronization mechanism. Constructing each proof step is where many ideas from structure editors converge to help the user.

There are many open problems for the future such as: minimizing the traffic between the prover and the front end, subcontracting work to external decision procedures, postprocessing and paraphrasing the proof.

This talk describes joint work with Laurent Thery and Yves Bertot.

Reuse and Declarative Formalisms

Bernard Lang
INRIA Rocquencourt, Le Chesnay Cedex, France

Being more a computer scientist than a software engineer, I see significant improvements to current software production problems more in tools and techniques than in methods, processes or management.

A fundamental paradigm of programming practice evolution and improvement (and, for that matter, scientific practice in general) is to hide the obvious and routine in order to concentrate on what is new and difficult. Hence, as programming techniques are mastered, they get integrated (hidden?) in languages, systems, and libraries and are thus unconsciously *reused* by everyone. This may concern application-specific know-how, which tends to be incorporated in libraries, or *architectural know-how*, which gets incorporated in languages. Most people agree that object-oriented techniques, though immature and poorly identified or defined, have considerably improved the programming practice. This is a strong hint for the need to further identify and analyze programming or architectural techniques that can be incorporated in our languages to permit and encourage better practice.

Note that there is a strong relation between languages and libraries. For one thing, components that used to be parts of languages (e.g. I/O routines, basic types) are now to be found in associated libraries (e.g. in ADA). From a different point of view, there is a continuum from simple reusable components, to parametric components, to component generators (i.e. "compilers"), where the parameters have evolved in complexity into a specialized programming language.

Another important aspect of software development is to try to emphasize what is to be done (specification) over how it is to be done (implementation), i.e. in more formal words, denotational formalisms over operational ones. In practice, it amounts to a direct compilation of the specification, thus skipping design and coding phases (e.g. direct compilation of grammars into parsers). Of course, this implies the development of sophisticated and specialized compilation techniques.

At INRIA-Rocquencourt we are developing two converging projects along those lines. The first one concerns the study of declarative formalisms based on Horn clauses. They include context-free grammars, datalog (deductive databases), Prolog like inference formalism, natural semantics, and a variety of syntactic formalisms used in natural language processing. Various applications are possible in software engineering, including notably component description languages, automatic reuse, configuration control, man-machine interfaces. For these formalisms we are developing implementations that (unlike Prolog) give all solutions for a given problem, with hopefully acceptable performances.

The second project concerns the development of an "object oriented" type system, which may be seen as an extension of ML polymorphism to abstract data types. The (much simplified) idea is that the programmer simply uses the functionalities he needs for manipulating his data, without having to worry on how this data is to be implemented. An inference system then deduces from the program the minimum signatures (sets of function names with their types) for modules offering these functionalities and implementing the corresponding data types. Finally the signatures are given as queries to a proof system, like the system developed in our first project, that searches a library of modules, module generators, and functors to identify or construct a proper implementation for the signatures. One of the tenets of this approach is that defining a type by its signature (standing for any implementation that satisfies it) is more abstract than choosing a specific implementation, and thus allows for greater modularity and reusability.

The Challenge of Controlling Change in Software Development

Wilhelm Schäfer
Universität Dortmund, Germany

Software is made up for being changed. Due to its characteristics is the possibility of changing it easily, at least physically. This mental picture of changing software easily is, however, a major reason for a non-systematic ("trial and error") approach for constructing even large software systems. It is consequently also the source of many major bugs and non-maintainable, non-documented software.

Our discipline (Software Engineering) has proposed a number of approaches to deal with that problem. Two of them are sketched here. One is to change the CS curricula. The university education has to include courses in team development, collaborative work with industry, etc., such that students get aware of the problems of substantial software development before they start to work on it in practice.

Second, controlling change can be supported by advanced software development environments. Based on a precise executable definition of the software process, those environments provide all information to a software developer (or manager) that is necessary to perform a

particular activity. This helps to foresee consequences of changes, to see effects of previous changes, and to automatically broadcast messages about changes to all team members.

Basic technology to build such automated support is partly available in other areas of CS and also in SE. It has to be taken and combined in an appropriate way and it has to be improved in one way or the other. Such technology includes goal-driven (rule-driven) descriptions of a software process combined with advanced configuration management, the use of object-oriented database systems to support persistency and safety during the course of a software project, group transaction concepts and basic concurrency control mechanisms (like events, triggers) to support multi-user distributed work, which in particular means access to shared information, and last but not least precise semantic models of the description formalisms applied.

Where the Formal Approaches to Software Development Go in the Next Decade

Wei Li
Beijing University of Aeronautics and Astronautics, Beijing, China

Formal approaches to software development is one of the important research directions of software engineering. In the last decade, the research in this direction was in the stage of program development. It focuses on the problems of how to formally develop programs for a given specification and how to verify the developed programs to meet the specification. The mathematical machinery employed is the deduction techniques of first oder logic or its varieties, such as intuitionistic logic or various kinds of modal logics.

In the next decade, it is most likely that the formal approach to *specification* development will become a new direction. The target will be how to formally develop a specification according to user's requirements. The difficulty is that, in general, we can not describe a user's requirement formally, because if it were, it would be a specification. To solve this kind if problem, the deduction tools based on first order logic mentioned above are not enough. We need a mechanism to formally describe the interaction between the logical (mathematical) information and the empirical information about the problem to be specified. A possible solution is to employ non-monotonic logic.

In this talk, the speaker will propose a logical theory of epistemic processes which is a non-monotonic approach to first order logic. In this theory, we will define the basic concepts involved in specification development, such as developing sequences of specifications, new-properties and user's rejections for a specification and reconstruction of a specification. All these concepts will be defined model-theoretically in first order logic. Non-monotonicity is defined as a characteristic of developing sequences. We will give a procedure to build a reconstruction of a specification when a new property or a user's rejection arises. We will further propose a paradigm of specification development. The concept of complete specification will also discussed, and the existence of complete specifications for a closed specification will be proved. We hope that this theory can be used to build the next generation of proof editors. Finally, we will compare our approach with other non-monotonic theories.

```
Giles Kahn, Computer assisted proofs
The major idea is to separate the proof engine from the Front-End.
This makes it possible to care about visualization, operations on the
```

representations of proofs separate from the prove technique. Moreover,
various provers can be attached to the Front-End.
Observations: 1) don't have control over the language the prover is written
in => need parsers and protocols
2) the user will not be happy, so it is mandatory to design for change
Adrion: mismatch between prover speed and display.
answer: not necessarily, because prover can build up results and history that
the user can scroll through.
Garlan: where is the database of theorems? (in the Prover)
does that not lead to a lot of duplication?
answer: not all provers maintain a database of theorems, so facility must
also be in Front-End. (Also use a cache of most recently used theorems?)
Observation: 75% of theorems are used as rewriting rules.
post processing on proofs is done on Front-End node.
(here I forgot to take notes for a while)
Wile: it seems that in your approach the reason for activating a particular
tactic is lost. All you do is click.
Dewayne: it seems that tactics in your approach become arbitrary programs
or rather macros. (answer?)

Bernard Lang, Reuse; Use of Deductive (or Declarative) Formalisms
Observation: going from reusable components to parameterized generic programs
means the parameterization becomes writing a program and the reusable program
a compiler.
2) Prolog straddles the fence: imperative programming and proof systems.
Garlan: where is natural semantics on your scale from context free to your
declarative formalism? answer: > decl. form.
Garlan: aren't some of your inference rules higher order? (yes)
Other work: abstract polymorphism, based on signatures instead of on types.
Erhard: does your inference engine automatically generate a type hierarchy?
answer: no, a signature hierarchy.
Garlan: do your signatures basically correspond to ML structs? (yes)
Erhard: I am not that fond of type inference. Redundancy can be very beneficial
Ossher: there are cases in which type inference fails.
topic: database queries (and browsing?) based on signatures (like Wing et alii)
Bernard: can have inference rule that composes components.
Phyllis: how is component semantics represented? (ultimately syntactically)
could lookup be based on signature + additional info? What kind of
additional info would be permissible? (no clear answer)
Dewayne: additional information can be handled in Inscape (what kind?)
Erhard: what is the ultimate goal?
Bernard: 1) object-oriented type system (for architectural reasons)
2) efficient Horn clause evaluator.
Snelting: you can infer a query?
Bernard: example: type checking of a function like 'mapcar'.

Wilhelm Schaefer, Controlling Change
1) change education: interact with industry = opportunity for SE experience
2) Document change and make visible to others
Erhard: is your control of change 'prescriptive' or 'advisory'?
answer: 'advisory'
Giles: are we really reinventing the wheel in SE? (have seen database do this)
answer: yes, occasionally; example, process programming reinvented the
distinction compiler/interpreter.
Nico: but sometimes close terminology, slightly different meaning; example:
transactions.
answer: true, Gail good paper on the difference between atomic transactions
in database queries and distributed systems versus long-range trans
actions for software module checkin-checkout.
Observation: don't go for an attempt to solve a problem without checking
whether other (sub)disciplines solved a similar problem.
Nagl: how is dynamic change described in the process programming model?
answer: can be expressed as a rule
Walter: Erhard, does process programming do any good?
Erhard: yes it does some good in making us aware of things to be done
without telling us how to do these things.
It is info providing, not imperative.
But does not work without serious customization for particular company
Bernard: how do you express process control? can't imagine that programmers
like to work under rigid task schedule demanded by system. I prefer a
declarative approach over a procedural approach. I want to express the
desired end state, not in terms of how to get there.
answer: yes, agree. That is why we work with Prolog.
Notkin: fear information explosion.
answer: automation can help uninteresting stuff migrate out (is that enough?)
Dieter: is project programming able to describe the different goals of many
people on a project?
Nagl: isn't a CM description more valuable than process control information?
answer: no, I want dynamic info in the object descriptors.
Wile: yes, that is useful. (would Nagl's CM not be able to do that?)
Gentelman: what about system integration and CM across company boundaries?
Axel: it should be known in advance how relevance of info is computed or
changed. (by 'role', 'responsibility' and 'time').

Wei Li, Whereto with formal approach?

```
1980-1990: formal approach to prog specs and to verifying prog = F(spec)
future: continue this approach and predict new: formal development of
requirement specs.
Wile: the problem is not 'formal rep' but indeed establishing formally that
       implem = spec (this seems counter to Garlan's use of formalism to
       show design differences and in general being precise).
Observation: the result of the development looks like algebraic specs.
Dewayne,Notkin: odd to use programming for writing specifications, while we
       are used to write specifications for programming.
Young: but what good does it do to program specifications this way?
Wile:  the design history helps you avoid making a mistake the second time
      around.
Bernard: you address correctness, but not completeness.
speaker believes you can theoretically also show completeness.
Dewayne: you need a deterministic programmer
Snelting: how do you guarantee that the development process terminates?
```

5 Session "Tools and Components"

Generating two-dimensional user interfaces out of graphical specifications

Stefan Jähnichen
TU Berlin, Germany

After some provocative introductory remarks concerning the topic of the workshop, the talk collapsed into an interesting and stimulating discussion. Main points were:

- Do we need to shift the software industry towards a precision industry ?
- To what extent do we have to base our research on our vision of future technology ?
- To what extent do we have to take the needs of our society into account ?

As the discussion was very vivid and fruitful, the author finally cancelled his talk and hopes for another opportunity to give the originally planned talk.

Fine-Grained Tool Integration

Harold Ossher
IBM T.J. Watson Research Center, Yorktown Heights, New York

The goal of tool integration is to permit construction of new tools or applications from existing, simpler ones. Not only is it important that the effort involved be significantly less than the effort needed to build the new tools from scratch, but it is also important that the quality and usability of the resulting composite tool be as high as a custom-written tool.

Consider a simple example: the integration of an editor and a spelling checker. Using traditional approaches, the tools would interact infrequently. When a file is saved, the spelling checker would check it and present any errors to the user, probably separated visually from the presentation used by the editor. This is much less satisfactory for the user than, for example, having every incorrectly-spelled word in the text presented by the editor be automatically highlighted at all times. Accomplishing this requires much tighter

cooperation between the editor and spelling checker. Writing the editor and spelling checker separately, yet so that they can cooperate as tightly as this not only with each other but also with other tools, written and not yet written, foreseen and not yet foreseen, is an interesting and important challenge.

We propose a fine-grained, object-oriented approach to tool integration. Tools are made up of individual methods and state variables of objects. Tools thus share data at the granularity of (small) objects. They are not explicitly aware of one another, but communicate via operations and event calls on objects. Multiple tools can respond to a single operation or event.

Tools built this way can be viewed as *matrices* of implementations, and can be combined using *extension* and *merge* operators. Given suitable underlying language support, this makes it possible to add new tools, and the new state they require, in a way that is both separate from and yet tightly integrated with the existing tools.

Though many open questions remain, our thesis is that this is a promising approach for building high-quality, extensible systems.

Inference Techniques Can Help a Lot

Gregor Snelting
TU Braunschweig, Germany

Automated deduction and unification theory have made significant progress in the last years, but the mathematics and algorithms developed in this field are not utilized in Software Engineering. One outstanding counterexample is polymorphic type inference in functional languages, which allows for greater flexibility and reusability, while at the same time still preserving the security of strong typing. I propose some other applications of inference techniques, which might be useful and should be investigated:

- use of generic type inference techniques allows for polymorphic and reusable software components with secure interfaces even for traditional languages like Modula-2 or C.
- advanced unification algorithms (e.g. AC1, order-sorted, higher-order) allow to use inferred properties of software objects as search keys for component retrieval.
- unification techniques can be the basis for inference-based configuration control, where consistent configurations can be inferred from partial specifications and inferred information can guide interactive configuration editing.

Thus, use of unification technology might be very useful.

Calculi vs. Abstraction Mechanisms

David S. Wile
Information Sciences Institute
University of Southern California, Marina del Rey, California

A primary tenet of engineering disciplines is that creativity must be limited and chan-

neled into only those areas of the problem being solved that truly distinguish it from similar problems. Hence, engineering disciplines build up reusable artifacts — models, analytic techniques, problem decompositions, and procedures — for each specific area of engineering. Careful study of effects from combinations of models and their properties are worked out in advance, cataloged in engineering "handbooks" for each engineering discipline. Fabrication from "first principles", with untried models whose properties are not well-understood, is entirely the purview of the scientists supporting the engineering technology.

"Software engineering" is in its infancy when measured against this yardstick of limited, disciplined, focused creativity leveraged through reuse of well-understood models and procedures. Software engineers have very few disciplines available to prevent them from treating every problem totally afresh. Worse, when computer scientists are unable to establish "general purpose models" that solve all problems, they provide software engineers with abstraction mechanisms that allow engineers to invent models that neither they nor anyone else has used before, whose properties and consequences are completely unexplored.

Of course, good abstraction mechanisms are extremely important for computer scientists establishing the bases for specific areas of software engineering, but they must be recognized as that: tools for experts to provide a sound engineering base for practitioners. I believe that the development of domain-specific languages and calculi for particular areas of software design will ultimately turn software engineering into a credible profession.

```
Jaenichen, societal Needs and technological Vision
Observation: Algol68 was a good research vehicle.
Conviction: software industry must be made into precision industry
       (attributed to Sintzoff) Focus is on correctness
Nico: but absolute correctness is not always needed and sometimes
       too costly to achieve. Other issues are likely to be as
       important or even more so: functionality, awkwardness,
       usefulness, performance, . .
All: correctness is only one aspect and not necessarily the most important.
     We want 'quality' which is determined by a number of factors that may
     carry different weight for different software products.
Wile: precision is not the same as correctness: software may be correct
      but not robust (for instance)
Nagl: is correctness achieved by precision, or precision by correctness?
speaker: challenge: focus research of 100 people on future 10 years.
      must lead them into new research areas (why?) The choice is based on
      NEED for society and VISION of where technology development is heading
many: doubt that this is needed, that it is useful and possible
Erhard: technology development awfully hard to predict
      (who would have thought ten years ago that everybody would have a VCR)
others: but no doubt about speed up, parallel computing and networking
also:   we all expect communication and computer technology to merge
Giles: I am not so sure of this merger: I still have my separate telephone
Wile:  you don't ask society what it needs, but develop what is possible into
       opportunity.
Erhard: but there are certain needs once you introduce technology; e.g.
       networking is available, but not safe. We must provide security and
       privacy (and accounting?)
Hoffman: instead of NEED, what ultimately counts is creation of market
Gentelman: NEED <> VISION; vision = anticipating possibilities
Axel: is software omnipotent? (no: FAX, VCR, . .)
Giles: society does not 'need' our work; society is concerned with issues
       such as unemployment and we cause more unemployment.
Nico:  yes, if we are concerned about 'need', we should be serious about
       reschooling.
Erhard: 'need' is not for precision, but for a useful, usable, cost-effective
       product.
Notkin: about tech.vision: no vision is needed for parallel computing; it is
       simply happening.
Gentelman: a new thing that is happening and we did (again) not foresee is
       the laptop disconnect/reconnect operation which has an impact on
       op sys data management.
Ossher: and these events influence the kind of software we write.
conclusion: it is impossible to foresee new technologies emerge. One should
       be prepared to change direction when it happens.
Dieter: ethics is an issue: we are responsible for what we produce.
Notkin: technical advances can have an impact on what we produce, but the
       scientific base we all are looking for is not much driven by
```

technology development.
Erhard: does society need a large number of small systems or a small number
of large systems (as T.J.Watson thought that US would need 6 computers)
Others: large number of small systems most likely
Garlan: state again the expectation that communication and computer tech are
going to merge.
Giles: not so sure this will happen soon: we have a hard time convincing
industry to use something better than assembly code (let alone the idea
they developed the skill of designing good special purpose
languages).

Harold Ossher, Fine-grained Tool Integration
building systems by composition in order to support system evolution.
Observation: effort of change must be proportional to size of change, not to
size of system.
2) In current systems a tool consists of bits and pieces (data + methods)
distributed over a collection of objects. These bits and pieces
collectively determine a behavior in the environment
Nico: it seems that you introduce a form of superclass definition. (yes, true)
Erhard: but these superclasses make the class hierarchy a lot more complex
and a lot of effort goes into looking for clashes. (does he propose
not to introduce new superclasses? yes, he advocates allowing objects
to be declared of more than one class, which does not affect the
existing class hierarchy.)
Ossher: by introducing the notion of 'perspective' of an object, you can
determine when which (version) of a method will be applied.
Axel: afraid of your extensions because of team design: people don't know
(or tell others) what their mates are doing.
Ossher: yes, but I am creating enabling technology and cannot guarantee that
team mates will talk and keep each other informed.

Snelting, Inference systems can contribute to SE
example: polymorphism, introduces a form of reuse.
Unification theory can be used for various purposes, e.g. database query by
signature (not including function name)
Nico: What about asking for T + 1 and getting T * 1 back?
what about getting all + functions for all types T?
Garlan: Use of Wing/Rolin system is extremely slow.
Giles: cannot do without higher order, because need to express semantics in
query. I want to be able to ask for all commutative functions for . .
Bernard: I do not agree that function name does not matter.
also, you cannot derive from text that function is commutative; you
need a declarative mechanism.
Gentelman: what do you expect from your approach? Tplot of AT&T has 27
parameters. Search by part fixed? and what is it that you understand
when you get an answer?
Notkin: I believe it is not helpful to work with a flat structured database
model. Hierarchy helps in understanding and browsing.
Bernard: but there will always be a need for a query that traverses the
hierarchy without the user's help (look for someone's email address)
Young: but basing semantics on the use of names does not work; e.g. stack
and queue structures have different method names, but abstract semantics
is the same and should be found when signature-based query is issued.
Walter: drawback of approach is that I ask for a function and typically do
not know how many parameters, or their types.
conclusion: there are different opinions about the rigid black box view and
flexible, hierarchical model, browsing approach.
Conjecture: inference tech can be useful for CM: it will e.g. answer which
particular component fits a given system version.
Walter: but the usefulness depends very much on what kind of properties i
can specify.
answer: these can be expressed as attributes and features defined by the
implementor. (are we getting into ordinary pattern matching?)
Snelting: languages and language features are a great contribution to SE
Giles: features are not invented first as part of language. Incorporation
into a language is the end of a thinking process when the idea is
well enough understood to make it available.

Dave Wile, Calculi versus Abstraction Mechanisms
Statement: engineering DISCIPLINE focusses and LIMITS creativity.
observation: distinguish software science, software engineering and software
craftmanship.
Goal: want to capture process of design.
design of special purpose languages is desirable, but still much to
hard for software engineers.
Garlan: are you saying that abstract data types are inductive?
answer: the thing you prove about an ADT instance concerns its entire
history and is therefore indeed an inductive argument.
Speaker shows example of notation that makes induction implicit.
Jaenichen: but induction is THE tool of computer scientists.
answer: yes, for scientists, but for engineers what has been learned must
be captured in routinely (re)usable form.

6 Session "Education"

Introductory Education in Programming

A. Nico Habermann
Carnegie Mellon University, Pittsburgh, Pennsylvania

The ultimate goal is to arrive at a theory of programs. Not just a theory of "how to program", but a theory that includes program models that software designers can rely on in practice. To move in that direction, we should abandon the "programming from scratch" approach and more attention should be paid to the *result* of our proofs (to *what* we prove) than to the fine points of our proof techniques (to *how* we prove). The result of a proof is a theorem which in turn ought to be usable in subsequent proofs and in writing programs. Properly naming and remembering these results leads to understanding and to knowing a coherent set of facts (mostly program models, but also properties of programs regarding space and time requirements) that software engineers can routinely apply in their daily work. It is particularly useful to capture proven results in names and concepts, and even in language constructs (such as functionals), so that proofs don't have to be repreated from scratch for every new instance.

Of the greatest importance for building and understanding a theory of programs are the notions of abstraction and specialization and the notion of program similarity. Students gain insight and knowledge by investigating for every concrete program how in can be seen as a particular instance of a more general program. This type of reasoning leads to *program schemata* that can be used in program design and leads to a form of reuse where programming becomes for a large part a matter of specializing a program schema into a concrete program that meets the specific requirements of the environments in which this program will operate.

Software Analysis

Hausi A. Müller
University of Victoria, Canada

The future of software engineering critically depends on how we look at and deal with software evolution and maintenance. We propose major re-alignments in software engineering education and research to strengthen the foundations of software evolution and maintenance.

Thesis I: Raise the status of software maintenance and evolution in the software engineering community commensurate with their socio-economic implications — software maintenance costs constitute 60-90% of overall project costs.

Thesis II: Software engineering training programs should carefully balance the proportions of software analysis and synthesis in their curricula.

In other engineering disciplines, the study and analysis of existing systems constitutes a major component of the curriculum. In Computer Science recurring concepts such as conceptual models, consistency and completeness, robustness, or levels of abstractions are

usually taught with a construction bias. However, recognizing abstractions in real-world systems is as crucial as designing adequate abstractions for new systems.

Thesis III: Shift software engineering research efforts from software construction to software analysis.

Methodologies, tools, and training for software analysis are clearly lagging behind software construction. A shift in research, from construction to analysis, would allow them to catch up. Software engineering researchers should test and validate their ideas on large, real-world software systems. One promising avenue of research is reverse engineering or recapture technologies which address a small, but important, part of the software analysis problem.

Thesis IV: There will always be old software.

Being able to analyze software more effectively will make software evolution, maintenance, and reuse more tractable.

What should we teach software engineers ?

Walter F. Tichy
Universität Karlsruhe, Germany

The demands on software engineers are clear: they must produce more, cheaper, and better software. The technical approaches to meet this demand include finding better ways of automating the development process, improving reuse and adaptation of existing software at all levels, and developing techniques, tools, and representations for specialized areas. Furthermore, we need to find solutions for new challenges, such as massive parallelism, man-machine interfaces (with gesture, speech, and vision), reactive systems, mobile devices in networks, safety, security, and privacy. But we also need to transmit known techniques to software engineers through education.

The current teaching practice is to cover the phenomenology of software development and the major concepts. But there is a noted absence of hard facts for software engineers to use in their daily work. Furthermore, problem solving skills are not taught to a sufficient degree. A sampling of software engineering texts reveals how unsuitable the problem sections are compared to other textbooks in Computer Science. In my own exams, I find that students memorize definitions faithfully and demonstrate passable understanding of the concepts in essay questions. They do poorly when asked to apply their knowledge to problems. I think we should emphasize problem solving skills in software engineering courses. The question is, in which areas and how ?

In software management, do we need to train students in team work, project planning, and technical writing ? In the specification area, in what types of notations do we demand skill, and which ones should students merely know about ? In design, which examples should we use to teach system architecture and train students in constructing designs, comparing designs, and modifying them ? What implementation tradeoffs do we teach ? What testing and analysis skills are adequate ? How do we teach skills in software maintenance, evolution, and reuse ? What should students know about software tools and programming environments ?

The answer to all questions above is that we *must* train our students in these areas. The

difficulty lies in picking those aspects that can be taught in a reasonable time frame and with university resources, yet are of long-term value. Furthermore, we need to begin to develop the materials for teaching the required skills effectively.

Software Education Engineering

Veronique Donzeau-Gouge
C.N.A.M Paris, France

How can we teach software in a software engineering program ?

I want to emphasize two points. Languages are main tools in software engineering: they were among the first designed tools, and many tools which are presently provided are in fact compilers of specific languages (cf. parser generators, meta interpreters, and so on). Furthermore, since they can only include mature concepts, we can say that the languages incorporate the best of our knowledge.

The second point is about the way programming languages are taught: old and poor languages are very often used for beginners and this does not help in beginner's learning. As the mother tongue influences one's thinking, the first learned computer language determines what can be conveniently expressed. We can choose the computer language.

What can motivate the choice of a programming language? It can be its simple syntax, its industrial use, its availability on PCs and so on. It can also be based on concepts which are well agreed upon by all the community such as abstraction, modularity.

We have to build the right foundations if we want to be able to build over. I propose, taking into account the present state of the art, to chose ML (CAML-light) for its expressive power, its sound foundations, and its using facilities and Ada for its good software engineering concepts. The semantic models of ML and Ada are very closed, and the concepts listed above can be explained first with a functional model before being immersed into the imperative world.

```
Nico Habermann, Introductory Education in Programming
1. Current approach to teaching programming: focus on syntax,
development from scratch.

   Goal: synthesis, models, families of programs, systems view (city
exploration analogy), emphasize tradeoffs (time/space, quality,
reusability), theory of programming (analytic geometry analogy.


Garlan: seems like you are presenting a theory of PROGRAMS, not of SYSTEMS.
Nico: yes, focus on programs represents a limitation of this talk (but
not the approach).

2. Example 1: Tree walk using depth-first and breadth-first search.
Define higher-level terminology (eg. permutable).

Lang: like Garbage Collection where discovered that stop-and-copy was
essentially the same as mark-sweep.
Notkin: why can't you use the recursive version of the example?
Nico: recursive example doesn't generalize in same kind of way.
Garlan: analysis seems to be missing the properties that would
cause you to choose one algorithm versus another.
Nico: yes, that would come up when you actually use one of these programs
        in a particular situation.
Hoffman: Also need to state conditions under which each solution applies.

3. Program Verification: instead of verifying from scratch each time
need reusable "proof nuggets". Also want to prove theorems about
family of algorithms which then allow you to reason about those
algorithms (eg. coloring theorem allows you to reason about
```

termination).

Garlan: "proofs as programs" would not separate proof of the theorem and the program that results -- they would be the same thing.
Nico: but does "proofs as programs" allow you define high-level terminology?

4. Program schemata:
Current examples: generics, polymorphic types, inheritance.

Lang: Inheritance doesn't preserve correctness.
Nico: Example: Series example and generalization to fix-point calculation.
Lang: OK. Your example preserves correctness, but it requires good-will on part of programmers.
Nico: True.
Ossher: "Next" routine normally wouldn't be a parameter, but a method that would need to be filled in by each subclass.

5. Conclusion
Routine engineering needed
Models shared among practioners
Modularity, Naming and Hierarchy

Notkin: I note that you went from specialization to abstraction, and not the other way around.
Garlan: Interesting similarities to architectural level of design, except at architectural level we do not yet have similar logical foundations.
Young: Students need to see examples to appreciate generalities.
Lang: Mathematics educators introduce abstractions only after students get maturity from working with examples (sometime 5-6 years of it).
Nico: Research should be striving to reduce practice to teachable knowledge.

Hausi Mueller, Software Analysis
In hindsight, what advice would you have given the people at the 1968 NATO SE confenerce in the light of the state we are in in 1992?
Young: by and large they spotted the important issues, witnessed by the fact that many of their ideas and observations are still fresh today.
Gentelman: but they had no real appreciation for prog-in-the-large and hardly any for prog-in-the-many.
Dieter: they seem to have believed in top-down approach and that did not work
Schaefer: we should have told them not to let mathematicians build up CS departments, because these guys sent us off in the wrong direction.
Bernard: another thing that did not materialize is the 'universal' programming language they were striving for.
Erhard: they may also not have realized that society is in 1992 depending on software systems it can no longer do without.
Dewayne: the quality of software has not changed much in these 24 years
Wile: I disagree strongly: the software I and many other people use is far better than in 1968, particularly op sys and application programs.
Bernard: in 1968 a CS person could know the entire field; no longer in 1992
Nico: I would recommend them to pay serious attention to the evolution of systems. They typically thought in terms of systems you build once and for all and basically stay the same when implemented and delivered.
Hausi: exactly!
1) maintenance and evolution (with socio-economic impact)
2) balance between synthesis & analysis
opinion: don't put full emphasis on construction; pay at least as much attention to architecture, recognition of ideas and concepts by reading programs, analysis and consider improvements.
Gentelman: but we need tools to help us read programs; text by itself is not very helpful (I think he means multiple visualization)
Notkin: people want their kids to read literature, not junk; reading programs must involve well written, well respected software, otherwise a waste.
Axel: but reading lousy programs can also be instructive (I don't agree) and, more importantly, students must review eachothers programs.
many: this is unfortunately not routinely done (claim: lack of time)
Young: require that code is written by at least two people because it combines reading (and understanding) with writing.
Hausi: shift from construction to software analysis: read, test, validate, reengineer
Nico: would like your analysis have a specific goal and result.
answer: goal of analysis is to enhance maintenance and evolution.
Nagl: reading programs and systems is not fruitful, because the code does not reveal the architecture.
answer: must nevertheless try to derive model from written program.
Notkin: and do you find a better way of writing maintainable software this way?
Walter: I don't like reengineering; transforming a COBOL program into C++ is a job nobody should be asked to do; instead, learn to write systems so that they can evolve over time.
Notkin: but you cannot be sure you have the right tools for evolution, because the next generation is bound to rewrite and change what you did according to their own style and preferences without regard for yours
Erhard: we should distinguish between 'reengineering' and 'reverse engineering'

Walter Tichy, The main Problems in Software Engineering

```
We all want more, cheaper, better software.
proposed answer: streamlining, automation, reuse, (domain) specialization +
      EDUCATION
Giles: are you thinking of CS or do you also include people in other disci-
      plines who do a fair amount of programming?
Walter: talk about people who will do software development (not about PhDs)
Bernard: people in other disciplines will not program in the future
      (I don't agree: physicist, mathematicians, astronomers will program)
Observation: we teach only few hard facts (we don't have many!)
Giles: careful: you can't separate content from didactics: e.g. lab course
      may teach quality (and taste) by experience
Walter: should we teach team development?
Nico: absolutely yes; it is part of education and also important for improving
      software quality and building communications skills.
Giles: I agree: team work is part of software engineering education.
Gail: teaching team work and writing documentation is too late in senior year;
      students should do it from the beginning and apply it in all CS courses
Young: incorporating SE in beginner courses is not a good idea, because there
      is little substance to which team work and documentation can be applied.
      (I think he goes farther than Gail; she said team work, not all of SE)
Walter: should you teach an overview or take one item and go in depth?
Dieter: overview can be useful for showing what the differences are
Walter: I don't have time to apply more than one.
Phyllis: don't confuse education with training
Wile:  you aversion of reverse engineering maskes it difficult for you to
      let students demonstrate their understanding of how to apply methods
      instead of reciting stale facts.
Garlan: formalism is important for being precise; the particular notation
      is secondary
Nagl: it is important to discuss the shortcomings of various methods.
Walter: students learn concepts such as 'information hiding' but must also
      practice the application of these concepts. How can they find time to
      do this, and how do I grade them on it? I need lots of examples.
Walter: do I teach 'optimization' as a topic?
Young: no, you teach them trade offs
Walter: how do I teach evolution & maintenance?
Wile: teach them reverse engineering!

Veronique Donzeau-Gouge Software Engineering Education
Observation: two parts: SOFTWARE and EBGINEERING; these are the two things
      on which education must be based.
opinion: programming language is THE tool for teaching SE
observation: languages are often taught from a historical perspective.
      it is not good to let beginners struggle with the restrictions and
      limitations of a language (=> don't start with Pascal)
good choice of modern languages: ML + Ada
opimion: ML has desirable programming properties, Ada is usable for systems
approach: use languages intertwined.
questions concern the way the languages are used.
```

7 Session "Development Process"

Software Engineering As a Managed Group Process

Axel Mahler
TU Berlin, Germany

In the past — and even today — software engineering research is mainly focussed on the problems incurred by the complex nature of software products. Engineering, however, also implies that less formal issues, such as communication patterns between humans, are to be taken into account.

Current software engineering ideology (as opposed to practice) strongly inclines the paradigm of *prescriptive methods*. All too often, these methods suffer from poor acceptance by the developers and eventually end up as "shelfware". We need a stronger emphasis on *descriptive* approaches towards the process of software development. There are few, if any, formalisms that allow to capture real process characteristics.

Finding an adequate approach to formalize relevant aspects of the development process is a hard problem. Part of the problem is the dynamic nature of the development process: it constantly changes in response to many parameters. Any static process description is likely to become obsolete almost instantly. A possible solution to this problem might lie in *adaptive software development environments*. Environments of this sort need to be open to easy and rapid extension and automatization in order to serve the real needs of the supported development process. By constantly monitoring the characteristics of the tool environment and by carefully analyzing the way it is used, a number of insights about the development process can be derived.

Having a — "living" — document that describes the current architecture and the current state of a truly adaptive environment will eventually be equivalent to having a valid process description.

Software Evolution

David Notkin
University of Washington, Seattle, Washington

Software evolution is a central software engineering problem. It is unavoidable, because its social an technological context changes. It is more costly than is desired. It tends to degrade desirable properties of a software system (such as efficiency, robustness, etc.).

The current software engineering world constantly states "Evolution is too costly.". When considered on a percentage, the cost of evolution relative to the total software cost is about 50 to 70 percent. But this tells us little about what would be reasonable or desirable cost. Two properties may help. First, *proportionality* states that the size of applying actual changes should be proportional to the size of the requested changes. Second, *predictability* states that the size of applying actual changes should be predictable, with reasonable accuracy in reasonable time, based on the size of the requested changes. Although the terminology is imprecise, it is intuitive and perhaps can be made precise. They may then be thought of as a lower bound on how well evolutionary techniques could work. Existing techniques, such as information hiding, may be considered upper bounds.

A necessary condition for achieving proportionality is that there is a strong association between equivalent structures at the specification, design, and implementation levels. A condition like this may be consciously broken — to achieve efficiency, for instance. But the tradeoff must be made with an understanding that evolution will not be proportional. Such tradeoffs, with subsequent properties of the resulting software, form the heart of software engineering.

Evolving Large Systems: Lessons from (over)simplified Development Processes

Dewayne E. Perry
AT&T Bell Laboratories, Murray Hill, New Jersey

It is my thesis that evolution begins very early in the development phase of a software process and that the distinction between development and maintenance should be aban-

doned in favor of an evolutionary process. I illustrate this by three simplistic views of the development process.

The first example is that where we simply implement a specification. The process is "explore, decide, and validate". This is of course too simple as there is local iteration within that process. Two important evolutionary considerations are local dependencies and tolerating incompleteness and inconsistencies. The basic problem with this simplistic view of the process is that it does not account for multiple levels of software.

The second example is that of a waterfall process. Of course, we do not really do things this way, but there is an important insight here. If, for purposes of illustration, we divide the process into the logical steps of requirements, architecture, design, and coding, we note that there is a well of knowledge that builds up behind each dam and only a certain amount of that spills over into the next level. This adds an additional step to the process for each level: rediscovery. Moreover, the evolutionary implications are that we now have multi-level iteration, exploration, dependencies, incompleteness, and inconsistencies. The basic problem with this simplistic view of the process is that is does not take into account "release" evolution.

It is worth noting at this point that rediscovery is not a problem for a single developer or even for a small group; however, it is a significant problem for large group of developers. Moreover, the complexity of the process is increased significantly with an increase in scale: the product can be in multiple states concurrently, and this exacerbates the problems of coordination and synchronization with respect to the artifact, and the problems cooperation and interaction with respect to the process.

The third example is a 3-dimensional waterfall, where successive releases are represented. This example adds an extra step to the process: inter-release rediscovery. Each successive release process is virtually identical to the original development process, only more highly constrained in that there is inter-release as well as inter-level rediscovery, incompleteness, inconsistencies, and dependencies. The fact of concurrent and possibly overlapping releases increases the complexity of the evolutionary process. Of major importance is the problem of multiple interdependent configurations. However, this view of the process does not address the problem of multiple products based on a single base or a single release.

We need more effective processes and support for rediscovery, exploration, decisions, and validation, and more effective support for iteration, dependency analysis and maintenance, version management, and component composition. Some fruitful approaches are semantically rich descriptions that are level and domain specific, codification and classification of architectural elements, architectural templates and styles, and process support for guidance, iteration, and automation.

Experimentation In Software Engineering Or How Can We Apply The Scientific Paradigm To Software Engineering ?

H. Dieter Rombach
Universität Kaiserslautern, Germany

Software engineering research has produced a large number of formal models, techniques and tools in the past. It is long overdue to experiment with these models, techniques and tools in order to better understand their benefits and limitations under varying circumstances and purposes. Such experience needs to be gathered in objective, measurable ways; and packaged together with those models, techniques and tools for future reuse. It is, for example, not sufficient to have well-defined white-box and black-box testing techniques. Instead, we need to understand which testing technique promises what results (e.g., detect 80% of interface faults) under what circumstances (e.g., re-active systems, designed according to object-oriented design principles, implemented in C++, certain fault profiles). Testing techniques packaged together with such experience can be reused in future projects run under similar circumstances.

Such experimental software engineering research requires a laboratory-kind environment where practitioners and researchers can cooperate. I presented a framework for such a laboratory based on experiences from the Software Engineering Laboratory (SEL), a joint venture between NASA's Goddard Space Flight Center, the University of Maryland, and Computer Sciences Corporation. This framework is based on the scientific research paradigm.

Such a paradigm shift (i.e., from purely theoretical and building-oriented to experimental software engineering research), defines a new class of software engineering research topics aimed at capturing and packaging of software engineering experience. Examples include the development of better software measurement approaches as well as techniques for formalizing, generalizing, tailoring and packaging software engineering experience. Additional research topics whose importance will grow over the next decade include better notations for representing software processes, better notations for capturing domain specific knowledge, better understanding of the impact of architectural software patterns on various qualities of the resulting software, and techniques and tools acknowledging evolution/maintenance/reuse of existing software rather than creation from scratch.

This paradigm shift should also be reflected in our software engineering curricula. They need to be revised in order to include the teaching of analytic skills. It is time to go beyond the teaching of languages and techniques. We need to find ways of teaching the skills enabling students to reason about the usefulness and limitations of candidate techniques and tools. It is also time to abandon the "everything is developed from scratch" syndrome. Students should learn from existing examples (i.e., reading before construction), and start to evolve/maintain/reuse existing software rather than build everything themselves.

```
Axel Mahler, Engineering and Management
Observation: SE management has been rather disappointing and has
      potential for great improvement
   2) people talk about 'shelfware', process descriptions that stay
      on the shelf and are ignored by the implementors.
   3) if management too rigid, programmers will ignore it and
```

resent it; if too flexible, it will not sufficiently enforce
4) great mismatch of management in the abstract and in practice
5) need balance between prescriptive and advisory (informative)
opinion: research people put too much effort in product and not enough in studying process.
Notkin: good products have been built with a lousy process; unfortunately a lousy product can be the result notwithstanding a good process. A good process (like a good language) does not guarantee a good product
Mahler: it is helpful to communicate experiences, both good and bad. One can learn from both success and failure stories
Gentelman: there is rigidity in the prescriptive approach; it is hard to make changes in the rules when that becomes necessary because of experience gained during execution of process; easier with descriptive approach than with prescriptive approach
answer: yes, being ready for changing the rules is very important
several: must have tools to support management and must have tools that measure management effectiveness so that necessary improvements can be tracked
Erhard: programmers have great fear for 'big brother' phenomenon: they see management as a personnel evaluation procedure by management.
Notkin: Japanese are much more ready to accept evaluation of their work and gladly use critique to improve personal performance
Schaefer: automate as much as possible for the sake of consistency and accuracy, and also for taking away a burden from the people involved.
Dieter: the power of process modeling is in integrating the various 'roles' that programmers, managers and others play in the development process.
Gentelman: interaction needed between team and future users (or contracting agency). What about tools for this task?
answer: not much hope now or in the future
Erhard: the process has general characteristics also found in other organizations, but we are in a favorable position with regard to the supporting software we have (or create)
Gentelman: yes, we are in a similar situation, but we do far worse in meeting delivery deadlines, estimating code size and cost, etc. (people raise eyebrows, and don't find a good explanation)
Dewayne: when you extend software, you don't just add more of the same (like doors or windows) but usually entirely different functionality.

David Notkin, Software Properties
Observation: 1) software systems are bound to need evolution because things such as new technology and new user requirements are inevitable.
2) proportionality idea of Ossher: effort of change must be proportional to size(change), not to size(system)
Hoffman: like to propose refinement of Parnas: when you make a change, you must check whether your users like it, whether it did indeed improve performance (if that was intended), etc. Ask yourself: was its goal met?
Ossher: be careful: you cannot foresee all changes ahead of time; you must be prepared to handle changes when the (unforeseen) need arises.
Giles: what is 'small'? the user may come to me and ask for a small change and I may tell him/her this is a huge change, because it affects my architecture. Should the implementor decide what is 'small'? (not always)
3) we can talk about lower bounds such as proportional to size(change) and about upper bounds, such as info hiding scope.
Wile: with your proportional rule you left out DATA; it is likely that a change that affects data is proportional to that data, not to the change
Notkin: yes, I did not take that into account, and I have to think about it
Young: although you have this upper bound, it may never be achievable; so, even if it exists, it may not be very useful.
4) a change should not cause a disproportionate effort in one of the various levels of system design.
Garlan: but there are cases where tools can make disproportionate changes tolerable, e.g. global change of a name with the 'replace' command

Dewayne Perry, Evolving Large Systems
subtitle: lessons from an oversimplified software development process.
General simplistic presentation: Spec -> implementation
Observations 1) a system always has more than one (orthogonal) organization, e.g. you may want to link all functions with a given signature, etc.
2) system consists of descriptive components which represent: requirements architecture, design and implementation (often put together in waterfall)
3) things that system builders typically are involved in: discover, explore, decide and validate.
4) waterfall model does not allow for 'release' information. If this is added, dimension is added and whole thing gets terribly unwieldy. trying this gets you into parallel variants and versions of variants.
5) possible solutions to control explosion: semantically rich descriptions, codification & classification.
Nico: I don't see why you could not do better when you organize by product component (seen as a tree-structured design of specializations of an overall abstract architecture in Dave Garlan's sense)
response: you won't reduce the information
others: you may lose reusability or component integration gets more difficult
Schaefer: the manufacturer often does not know the architecture.

Dieter Rombach, How to apply science paradigm to Software Engineering

1) most important is the experience of knowing what to apply when,
particularly for new problems (not just repetitive application)
2) process modeling research is only possible in a laboratory context
and with a close interaction with an industrial partner.
3) product model = requirement + architecture + design + code
process model = integration of various (project people) roles
4) Software engineering = planning the process with the support of
a source of techniques, tools and EXPERIENCE

Nico: I miss the software substance in that definition; here is Mary Shaw's
definition of software engineering (research):
the activity of reducing to routine practice the application of
well-understood techniques, tools and software artifacts for the
purpose of constructing and maintaining software systems

Phyllis: don't forget to distinguish between SE research and SE practice

Mahler: where do you draw the line between 'technique' and 'process' ?

opinion: software engineering = the creation of a process model.
(many in the audience think this definition is too narrow)

Gail: why is 'process' not part of the collection techniques, tools and
experience?

Notkin: what kind of objects do you put in your database that represent this
collection? (answer escaped me, I was getting tired)

5) do experiment in univ lab and then repeat and test in industry

Notkin: how often did an example work in univ and fail in industry? (50%)

[I (Nico) apologize to the speakers of this afternoon,
but I ran out of steam.]