Dagstuhl Seminar 99071

Software Engineering Research and Education: Seeking a new Agenda

Taking Stock of Software Engineering Research and Education
What do we know? What should we know?

February 14 – 19, 1999

organized by

Ernst Denert, sd&m and Technical University München, Germany
Daniel Hoffman, University of Victoria, Canada
Jochen Ludewig, University Stuttgart, Germany
David L. Parnas, McMaster University, Canada

Preface

Software Engineering should address, and solve, existing problems.

Software Engineering as a branch of computer science emerged from discussions and conferences in the late sixties. Its goal was to apply knowledge and techniques from traditional engineering to the construction and maintenance of software.

Now, as the end of the century draws near, many people apply concepts, techniques, and notations created in, and around, the field of software engineering. But we are far away from a common understanding of what the important problems are and which approaches are appropriate. We have many conferences and magazines of questionable value, and little agreement about what should be taught in universities and which topics should be further investigated.

This workshop attempts to reach an agreement about these questions, for those who participate, but hopefully also with some effect on our colleagues who don't. By discussing our ability to solve those problems which actually occur in software engineering, we hope to identify what should be in the curriculum and in the research agenda.

Jochen Ludewig, 1999-02-16

Introduction

This report is the result of a very intensive five-day workshop at Dagstuhl in February 1999. As part of the announcement and invitation, a list of suggested tasks was distributed; those who intended to participate were asked to submit a position paper on at least one of the subjects, and to judge the current state of all the subjects. Here is the initial list:

Analyze intended application, write requirements document

... to determine the requirements that must be satisfied. Record those requirements in a precise, well-organized and easily-used document.

Select the basic hardware and software components.

Analyze the performance of a proposed design,

... either analytically or by simulation, to ensure that the proposed system can meet the application's requirements.

Produce an estimate of the cost (effort and time) of the proposed system.

Design the basic structure of the software,

... i.e., its division into modules, the interfaces between those modules, and the structure of individual programs while precisely documenting all software design decisions.

Analyze the software structure for its quality,

... i.e. for completeness, consistency, and suitability for the intended application.

Implement the software as a set of well-structured and documented programs.

Integrate new software with existing or off-the-shelf software.

Perform systematic and statistical testing

... of the software and the integrated computer system.

Revise and enhance software systems,

... maintaining (or improving) their conceptual integrity and keeping documents complete and accurate.

Demonstrate that the resulting software meets the expectations.

At the beginning of the workshop, participants reduced and modified this list for various reasons, until eight topics for discussion and elaboration were identified:

- Requirements
- Design, Structures
- Implementation
- COTS (commercial-off-the-shelf software)
- Software Families
- Test

- Maintenance
- Measurement

Later on, the topic „Software Configuration Management“ was added.

Out of 23 people who planned to participate, three (including Dave Parnas, who had initiated this workshop) were not able to attend, mainly due to illness. Here is the list of those who actually arrived at Dagstuhl. They all stayed for at least four days, most of them for the full workshop. See the complete list, including addresses, in the appendix.

Joanne Atlee	Motoei Azuma	Wolfram Bartussek
Jan Bredereke	Ernst Denert	Karol Frühauf
Martin Glinz	Daniel M. Hoffman	Heinrich Hußmann
Pankaj Jalote	Ridha Khedri	Peter Knoke
Stefan Krauß	Jochen Ludewig	Lutz Prechelt
Johannes Siedersleben	Paul Strooper	Walter F. Tichy
David Weiss	Andreas Zeller	

Everybody participated in two (or three) groups:

	chairing	Design, Structures **Sieders-leben**	Software Families **Weiss**	Mainte-nance **Zeller**	Config. Mgmt. **Tichy**	Measure-ment **Azuma**
Requirements	**Atlee**	Atlee Glinz Prechelt	Bartussek Weiss	Bredereke Khedri		
Implementation	**Knoke**	Denert Siedersleben				Knoke Ludewig
COTS	**Strooper**	Hußmann	Strooper	Krauß Zeller	Tichy	Jalote
Test	**Hoffman**	Hoffman		Frühauf		Azuma

Prechelt: also in Implementation

This organization allowed for two meetings every day, plus two plenary meetings where the intermediate results were presented and discussed. The content of the report was complete by the end of the workshop; it was revised and finished after the workshop. Dan Hoffman and Stefan Krauß did this work.

The results, though presented and discussed in plenary meetings, are certainly not agreed upon by all participants in any detail; we do believe, however, that they in general express our consensus, and can be used as a starting point for further discussions. Several participants have expressed their interest in a permanent activity along the lines initiated at Dagstuhl; we will try to keep the spirit alive by presenting our results in conferences and magazines, hopefully stimulating responses from those who were missing. Software Engineering needs definitely more work like this.

Readers of this report, who are interested to contribute, or just keep in touch, are invited to contact any of the organizers.

All participants enjoyed the excellent working conditions provided in Dagstuhl, and the quiet, but efficient support from the staff there.

Topics

Lead authors are shown below in parentheses.

1. Requirements (Joanne Atlee) 9

Analyze the intended application to determine the requirements that must be satisfied. Record those requirements in a precise, well-organized and easily used document.

2. Structures (Johannes Siedersleben) 17

Design the basic structure of the software, i.e., its division into modules, the interfaces between those modules, and the structure of individual programs while precisely documenting all design decisions.

3. Implementation (Peter Knoke) 18

Implement the software as a set of well-structured and documented programs.

4. COTS (Paul Strooper) 23

Integrate new software with existing or off-the-shelf software.

5. Test (Dan Hoffman) 29

Perform systematic and statistical testing of the software and the integrated computer system.

6. Families (David Weiss) 35

Design a set of similar software products as a family exploiting the similarities between the products.

7. Maintenance (Andreas Zeller) 41

Revise and enhance software systems, maintaining (or improving) their conceptual integrity, and keeping documents complete and accurate.

8. Measurement (Motei Azuma) 44

Maintain product and process metrics and measurements, and use them to evaluate existing and future products and processes.

9. Configuration management (Walter Tichy) 54

Keep order in long-lived, multi-person software projects.

Tabular Evaluation Format

In this report, tables are used to provide a standardized evaluation of the existing means for each task, i.e., to solutions of the problem posed by performing the task. Each table row corresponds to a means of performing a task. There is one table column for each of the following attributes:

Effectiveness. How well the solution works, considering factors such as how much of the task it covers and how good a solution it is to the problem posed by accomplishing the task. Ratings are High (the solution is very effective), Medium (the solution is somewhat effective), and Low (the solution is not very effective).

Affordability. The extent to which a typical software development organization can afford to perform the solution. Note that it may be that a solution is high cost, but that an organization cannot afford *not* to use it. Ratings are High (the solution is very affordable), Medium (the solution is somewhat affordable), and Low (the solution requires relatively high investment).

Teachability. The extent to which the solution can be taught in a University, including the body of knowledge that must be conveyed to students and how well we understand how to convey that body of knowledge. Ratings are High (we know how to teach the solution very well), Medium (we know how to teach the solution to some extent), and Low (we do not really know how to teach the solution).

Use in Practice. The extent to which the solution has been adopted by industry. Ratings are High (the solution is widely used), Medium (the solution is somewhat used), and Low (the solution is not used very much). For use in practice we also provide an alternative view of the evaluation, namely the class of users who have adopted the solution, where class is one of the following: laboratory users (LU), innovators (IN - those who are willing to use early prototypes of the solution), early adopters (EA - those who are willing to use advanced prototypes of the solution), early majority (EM - those who are willing to be the first users of industrial-quality versions of the solution), and late majority (LM - those who will not use the solution until there is considerable industrial experience with it). Note that these categories are taken from *Diffusion of Innovations* by E. M. Rogers.

Research Potential - The extent to which the set of existing solutions to a problem could be improved. Ratings are High (better solutions would greatly improve effectiveness, affordability and/or teachability), Medium (better solutions would provide some improvement), and Low (new solutions would not be substantially better).

Requirements

Joanne Atlee, Wolfram Bartussek, Jan Bredereke, Martin Glinz, Ridha Khedri, Lutz Prechelt, David Weiss

The Question

How can we analyze the intended application to determine the requirements that must be satisfied? How should we record those requirements in a precise, well-organized and easily-used document?

Requirements Engineering is the understanding, describing and managing of what users desire, need and can afford in a system to be developed. The goal of requirements engineering is a complete, correct, and unambiguous understanding of the users' requirements. The product is a precise description of the requirements in a well-organized document that can be read and reviewed by both users and software developers.

Short answer – Partially Solved

In practice, this goal is rarely achieved. In most projects, a significant number of software development errors can be traced to incomplete or misunderstood requirements. Worse, requirements errors are often not detected until later phases of the software project, when it is much more difficult and expensive to make significant changes. There is also evidence that requirements errors are more likely to be safety-critical than design or implementation errors.

We need to improve the state of requirements engineering by improving our application of existing practices and techniques, evaluating the strengths and weaknesses of the existing practices and techniques, and developing new practices and techniques where the existing ones do not suffice.

Long answer

The above short answer is unsatisfying because it doesn't convey the different aspects of the question. The answer depends on

- the task to be performed (e.g., elicitation, documentation, validation)
- the application domain (e.g., reactive system, information system, scientific applications)
- the degree of familiarity (i.e., innovative vs. routine applications)
- the degree of perfection desired (e.g., 100% perfection or "good enough to keep the customer satisfied")

Rather than provide a complete answer, we choose to answer the question on the basis of the different requirements engineering tasks. With respect to the other aspects of the problem, our answers are domain-independent, they apply to innovative applications rather than routine applications, and they apply to the development of high-quality software. If we had considered a different slice of the problem, we would have arrived at different answers.

Substructure of the problem

We divide requirements engineering into five tasks:

Elicitation - extracting from the users an understanding of what they desire and need in a software system, and what they can afford.

Description/Representation - recording the users' requirements in a precise, well-organized and easily-used document.

Validation - evaluating the requirements document with respect to the users' understanding of their requirements. This sub-task also involves checking that the requirements document is internally consistent, complete, and unambiguous.

Management - monitoring and controlling the process of developing and evaluating the requirements document to ease its maintenance and to track the accountability of faults.

Cost/Value Estimation - analyzing the costs and benefits of both the product and the requirements engineering activities. This sub-task also includes estimating the feasibility of the product from the requirements.

Table 1. Structure of the topics of Requirements engineering

1	**Elicitation**	
	1.1	Gathering Information (interviews, questionnaires, joint meetings...)
	1.2	Requirements analysis methods (SA, OOA, scenarios,...)
	1.3	Prototyping
	1.4	Consensus building and view integration
2	**Description/representation**	
	2.1	Natural language description
	2.2	Semiformal modeling of functional requirements
	2.3	Formal modeling of functional requirements
	2.4	Documentation of non-functional requirements
	2.5	Documentation of expected changes
3	**Validation**	
	3.1	Reviews (all kinds: inspection, walkthrough, ...)
	3.2	Prototyping (direct validation by using prototype / testing the prototype)
	3.3	Simulation of requirements models
	3.4	Automated checking (consistency, model checking)
	3.5	Proof
4	**Management**	
	4.1	Baselining requirements and simple change management
	4.2	Evolution of requirements
	4.3	Pre-tracing (information source(s) ↔ requirement)
	4.4	Post-tracing (requirement ↔ design decision(s) & implementation)
	4.5	Requirements phase planning (cost, resources,...)
5	**Cost/value**	
	5.1	Estimating requirements costs
	5.2	Determining costs and benefits of RE activities
	5.3	Determining costs and benefits of a system (from the requirements)
	5.4	Estimating feasibility of a system

For each task, we determine a selection of techniques that have been proposed as solutions to that task (see Table 1). This list should neither be considered complete, nor should it be interpreted as our opinion of the best techniques; it is simply a sampling of the solution space of the task.

Also, we do not consider any specific techniques for any task (e.g., UML collaboration diagrams). Instead, we consider how well classes of techniques solve a particular task. Answers for specific techniques would be more interesting and more useful than answers for classes of techniques, but would have greatly lengthened this report.

Ranking of the Different Aspects

The tables in this section provide an evaluation of how well classes of techniques solve the problems posed by performing the task.

Problem: Elicitation

Ranking of solutions

Solution	Effective-ness	Afford-ability	Teach-ability	Use in practice	Research potential	Comments
Gathering information (interviews, questionnaires, joint meetings...)	medium	high	medium	ad hoc: high sound: low	medium	
Requirements analysis methods and languages (SA, OOA,...)	medium	medium	high?	low	medium	1
Prototyping	high	low	medium	ad hoc: high sound: low	low	
Consensus building & view integration	medium	low	medium?	low	high	

1. Analysis itself is hard to teach, but some concrete languages and methods are easy.

Problem: Description

Ranking of solutions

Solution	Effective-ness	Afford-ability	Teach-ability	Use in practice	Research potential	Comments
Natural language description	medium?	high	medium	high	medium	
Semi-formal modeling of functional requirements	medium - high	high	high	medium?	medium - high	
Formal modeling of functional requirements	medium	low	medium	low	medium	1
Documentation of non-functional requirements	high	low	low - medium?	low	high	2
Documentation of expected changes	high	high	medium	low	medium	

1. Affordability is high in specific situations when checking important or safety-critical properties
2. Rankings are for those techniques we know (however, we do not know enough)

Problem: Validation

Ranking of solutions

Solution	Effective-ness	Afford-ability	Teach-ability	Use in practice	Research potential	Comments
Reviews (all kinds)	high	high	high	high	medium	
Prototyping	high	medium	medium	medium	medium	
Simulation	high, if feasible	low - medium	medium	low	high?	
Automated checking (consistency, model checking)	high, if feasible	medium	high	low	high	
Proof	high, if feasible	low (except safety-critical systems)	low	low	high	1

1. Research potential high especially concerning feasibility and developing new methods

Problem: Management

Ranking of solutions

Solution	Effective-ness	Afford-ability	Teach-ability	Use in practice	Research potential	Comments
Baselining requirements, simple change management	high	high	high	medium	low	
Evolution of requirements	high	medium - high	low - medium?	low	medium - high?	
Pre-tracing (info sources <->rqmts)	medium	medium?	medium	very low	medium?	
Post-tracing (rqmts <-> design&code)	medium - high	low - medium?	low - medium?	very low	medium?	
Requirements phase planning	high	high	high?	medium	low	

Problem: Cost/Value

Ranking of solutions

Solution	Effective-ness	Afford-ability	Teach-ability	Use in practice	Research potential	Comments
Estimating requirements cost	medium	medium	medium	low	high	1
Determining cost/benefit of RE activities	low	low	low	low	high	2
Estimating costs/benefits of a system (from the requirements)	medium	medium	low	low	high	3
Estimating feasibility	medium	low	low	medium	high	

1. Experience-based techniques dominate in practice
2. Only ad hoc techniques, motivated by fear of not doing them
3. Requires marketing techniques as well as technical ones

What should be taught

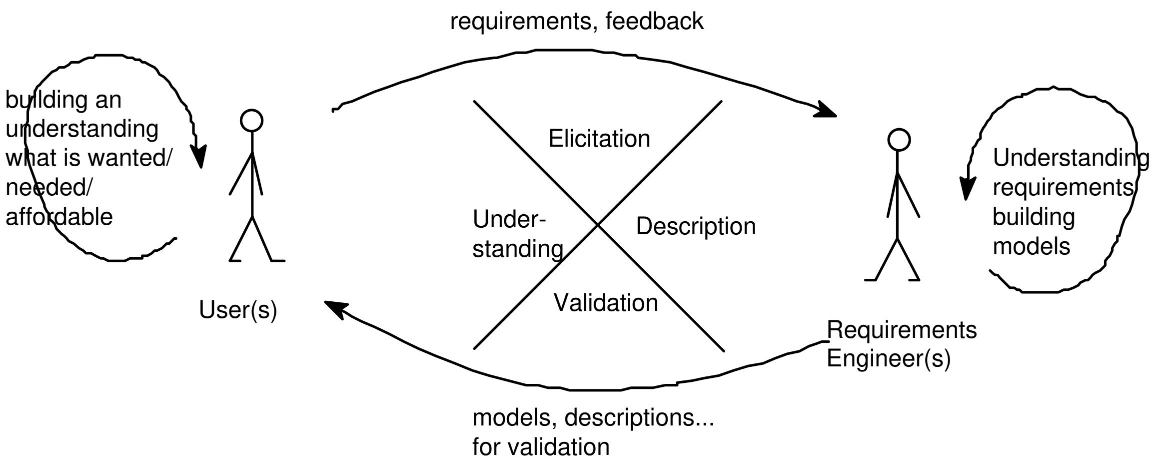
We identified five major points that teaching of requirements engineering should be centered around:

1. The basic RE-process
2. The problems and obstacles encountered
3. The principles of good RE
4. A selection of techniques
5. Practice, practice, practice

The basic RE-process

The requirements engineering process iterates over four main tasks:

- The users develop an understanding of what behavior is desired/needed.
- The requirements engineers elicit requirements from the users.
- The requirements engineers build a model of required behavior based on the elicited information.
- The users evaluate the engineers' model against their understanding of their requirements.



With each iteration, the users form a more concrete understanding of their needs, and the requirements engineers elicit and document more accurate and more precise behaviors. The process terminates when the users decide that the engineers' descriptions match the users' conceptual models.

Problems and obstacles

Requirements engineers need to be aware of the problems and obstacles to eliciting and specifying a complete and correct set of requirements. Several good techniques have been developed to help solve different problems and overcome certain obstacles. However, these techniques are not foolproof, they do not cover all of the cases, and many are incompatible with one another. Below is a sample list of problems; the list is neither ordered nor complete.

- Users do not know what they want

- Users specify requirements in terms of solutions
- Different stakeholders have different, possibly inconsistent views and needs
- The "right" users are unknown or unavailable
- Requirements evolve (shooting at moving targets)

Principles of good RE

Requirements engineers need to understand and be able to apply good engineering principles, as they pertain to eliciting and documenting requirements. Several good techniques have been developed that codify the application of certain principles. However, the principles have become lost in the presentation of the techniques. This is a problem because the techniques themselves are not complete; they do not cover every possible situation. Requirements engineers need to be able to fall back on the fundamental principles when the techniques let them down. Below is a sample list of RE principles; the list is neither ordered nor complete.

- Separation of concerns
- Abstraction
- Precision
- Planning for change
- Reviewability; (automated) analyzability
- Continuous validation
- Support for testing
- Support for achieving consistency and completeness
- Variation of depth and precision to accommodate the cost and risk of the problem

A selection of techniques

Requirements engineers need a collection of effective techniques for eliciting, describing, validating, and managing requirements. The techniques are simply heuristics for solving the problems, overcoming the obstacles, and adhering to the principles; they are not complete solutions. Therefore, the specific techniques that are taught should be chosen for their ability to solve key problems and for their coverage and support of key principles. Also, it is important to teach not only the bare techniques, but also where they can be applied effectively and why they work.

Practice, Practice, Practice

It is essential that the students apply in practice what they have been taught. The course must have a project component where the students are exposed to the problems and obstacles and can practice the application of some of the principles and techniques. A good project would include design, implementation, and enhancement of the requirements, so that the consequences of good or bad requirements engineering are revealed and can be experienced.

Research agenda

In general

The following research problems are not specific to requirement engineering. They arise from the structure of the table format used to evaluate problems and techniques, and apply to all of the SE tasks:

- Evaluate the effectiveness/cost of existing techniques
- Develop new techniques that
  - adhere more closely to more principles / neglect fewer ones
  - do a better job of solving problems completely/overcoming obstacles
  - have a lower cost

In particular

The following research problems are those sub-tasks and techniques which we rank as having high research potential - either because existing techniques are not very effective, are too expensive, or are deemed too difficult-to-use for widespread practice.

- Elicitation:
  - Consensus building & view integration
- Description:
  - Semi-formal modeling of functional requirements
  - Documentation of non-functional requirements
- Validation:
  - Simulation
  - Automated checking
  - Proof (for high-risk-systems)
- Management:
  - Requirements evolution
- Cost/value:
  - Estimating requirements costs
  - Determining costs and benefits of RE activities
  - Determining costs and benefits of a system (from the requirements)
  - Estimating feasibility of a system

Software Design

Joanne Atlee, Ernst Denert, Martin Glinz, Dan Hoffman, Heinrich Hußmann, Lutz Prechelt, Johannes Siedersleben

The Question

Design the basic structures of the software, evaluate competing design alternatives, and reuse existing designs whenever possible.

Short Answer

The design task is solved at the level of individual modules, but not solved at the level of system architecture.

Directions for Research and Teaching

1. We believe most PRINCIPLES of successful design are known:
    - describe parts by their interfaces
    - hide details behind interfaces
    - separate concerns
    - design for change
2. All design techniques are mostly heuristics for applying these principles. Their success depends on the degree to which they enforce or encourage the use of the principles.
3. For the design of one or a few modules, sufficient techniques are available, e.g. object oriented methods, structured design, design patterns.
4. Teaching has to make sure that the principles of separation of concerns and information hiding are not forgotten, as they often are, when teaching object oriented methods.
5. For the design of system architectures, the known techniques are not sufficient. Substantial research effort will be needed in order to develop sufficient techniques. The emerging proposals must be validated.
6. We do not know how to relate different levels of abstraction in a large system. Since this capability is required for producing a consistent design, more research is needed in this area.
7. We should collect architectural patterns, that is, proven and concrete architectural building blocks aimed at particular problems. These may or may not be domain-specific. The biggest such patterns would be reference architectures for a particular class of applications. This approach will yield useful results faster than attempts to define new methods based on first principles.
8. We should study successful projects. This will contribute to both, (5) and (7): Studying the structure of the systems helps identifying architectural patterns and studying the design process may tell us which activities lead to a successful design. The study of project failures would also help.

Software Implementation

Ernst Denert, Peter Knoke, Jochen Ludewig, Lutz Prechelt, Johannes Siedersleben

Topic Description

Implement the software as a set of well-structured and documented programs.

Assumptions

The following assumptions apply to this section. They were made to allow rapid focus with associated results in a short time.

- Good requirements and design documents exist, and the implementer has access to these documents.
- The implementer's primary job is to implement the software design using a programming language of some kind. He creates source code which realizes the intent of the design. This programming language can range from assembly language through conventional 3rd generation languages, object oriented languages, and visual programming languages to 4th Generation Languages.
- The implementer's job includes unit testing, the results of which serve as evidence of satisfactory implementation. It also includes good documentation of the source code. However, the implementer's job does not usually include integration testing or full system testing.
- The implementation phase typically includes such quality-enhancing processes as walkthroughs and code inspections in which the implementer must participate. The inspections may range from personal inspections to formal inspections by programmer peers.
- The emphasis of this section is primarily on implementation as an individual activity, as opposed to implementation as a team activity.
- The implementer does detail level design, e.g. design at a module level. He has to deal with ambiguous, incomplete or possibly incorrect design documents, and he has the capability to deal with such phenomena. If the implementer needs to repair such design faults or weaknesses, he is aware of and complies with suitable procedures set up for such cases.
- Implementation for or with code reuse is outside the scope of this section.
- The primary focus of this section is on the implementation of standard information systems software. Other areas such as real-time software, embedded system software, and software for parallel systems are outside the scope. Most emphasis is on generic issues rather than domain or application-specific matters.

The Problem

The problem is that the quality of the software implementation is generally not as good as it could and should be. Quality source code is important, because poor source code increases software test time and cost, decreases software performance, and reduces software maintainability.

The Question

How can this problem situation be improved by various means, including SE teaching, SE research, or other means?

Short Answers for Solutions

SOL 1: Programming as an activity should receive more emphasis and credit by management. Specifically, programming positions should be filled with skilled and experienced people, i.e., quality people.

Ranking: SOLVED, but not STANDARD OPERATING PROCEDURE.

SOL 2: SE teaching should be improved so that more highly skilled and professional programmers are educated and trained.

Ranking: SOLVED, but not STANDARD OPERATING PROCEDURE.

SOL 3: Software engineers and their management should seek to improve their management of change (i.e., reduce chaos) in the programming language and programming environment technology areas.

Ranking: PARTIALLY SOLVED.

Rationale for Ranking

SOL 1: SOLVED because management can presently decide to commit resources to this area, but it is not STANDARD OPERATING PROCEDURE for management to apply the best and most experienced people to programmer (coder) positions.

SOL 2: SOLVED because for almost any programming area there are text books and sources of expert opinion. However, in some places where programming is taught the best teaching practices are not used. There is a substantial gap between the known best teaching practices and the widely used teaching practices. The problem is aggravated by rapidly changing implementation technologies. The use of the best practices here is not STANDARD OPERATING PROCEDURE.

SOL 3: PARTIALLY SOLVED. Change management practices of some sort are inevitably used in all areas of rapid technological change. However, these practices tend to be "ad hoc": they are usually neither general nor well-defined. Change management practices must be used by all software developers, but the use of a good set of these practices is not STANDARD OPERATING PROCEDURE. The SEI CMM (Capability Maturity Model) includes Technology Change Management as a Key Process Area at Level 5, but most software development organizations today are operating at CMM level 2 or 3 at the most. Many such organizations have little or no knowledge of the SEI CMM.

Examples, Limits

SOL 1 Examples

Often, young people are fast coders, rapid learners of new programming languages, paid a relatively low salary, and willing to work long hours of unpaid overtime in developing programs. This situation provides an incentive for managers, needing to get a lot of programming done fast, to hire young, inexperienced, and maybe poorly trained programmers. The damaging effects of such decisions might not be seen for some time. Microsoft is well known for routinely hiring young, bright, hardworking programmers (however, usually such programmers serve initially in software test positions).

It is understandable that managers in software development companies, faced with the choice of possibly bad code discovered and regretted later versus the near certainty of bankruptcy tomorrow, might opt for the former. This particular tradeoff (quality vs. cost and time) is definitely domain specific. A Space Shuttle which blows up due to a software failure in the full view of millions is a thing to be avoided in spite of high costs of its avoidance. A word processing program which occasionally crashes when an esoteric feature is used is another matter.

One contributor to this section mentioned a possible management policy of not hiring an implementer who has not written at least one big program (say, 5000 LOC). Such a policy is domain specific. There is now a considerable demand for people who can quickly produce Web Pages. At minimum, such a task may require a little knowledge of HTML. It is easy to learn the use of HTML, but experience and creativity are required to produce consistently good web pages.

One contributor to this section noted that an evaluation of good programming skills is needed at some time (by industry if not by the university). Microsoft routinely does this evaluation at new hire interview time, where the recruiters are themselves programmers.

SOL 2 Examples

The teaching of implementation (programming) is often done by instructors with no interest and no experience.

More emphasis should be placed on reading good code. The question is, where to look for examples of such good code.

A good programmer (coder) should know when to use REPEAT, when to use WHILE, etc.

The use of assertions and preconditions, as recommended by Dijkstra, may be desirable.

Coding can be regarded as a craft (similar to carpentry). If so, perhaps teaching techniques used for teaching other crafts are applicable to teaching coding.

Walkthroughs, which may be important for good quality code, may be difficult for code in languages like as Smalltalk or C++.

Books like "Elements of Programming Style" can be helpful in reducing the amount of bad code being produced. However, "good style" may be programming language and computer platform dependent. Perhaps there exists a core of "Good Programming Style" which is independent of program language and computer platform.

Good documentation is recognized as a solved problem, but it is not SOP.

In implementation, it may be desirable to separate interface issues from other implementation issues, especially if portability and flexibility are desired source code features.

SOL 3 Examples

The SEI Capability Maturity Model level 5 includes a Key Process Area called Technology Change Management (see "The Capability Maturity Model: Guideline for Improving the Software Process", CMU SEI, Addison-Wesley, 1995)

What Can Be Taught? How?

Everything in this topic area can usually be taught without great difficulty. However, academic policy issues determine what will be taught and by whom. Some of the teaching/training can be delivered outside the universities. If programming is indeed a craft, then perhaps it can be taught with the aid of apprenticeships and internships in industry.

Research Topics, Goals

Not much need is seen for research in the implementation area. However, some potentially promising research areas examples are noted below.

- Research might be desirable into what teaching methods and programming styles work best and why, or what is important and what is not. This could be described as "empirical" software engineering research. There could be research into the impact of language limitations, such as a limitation on the allowed number of characters for an identifier. There could be research into the costs of bad or non-existent program documentation, such as in the case of the Y2K problem. What are the advantages and disadvantages of the use of multiple-level inheritance as opposed to single level inheritance in object oriented programming languages? What were the true quantitative costs of the use of the GOTO statement and perhaps resultant "spaghetti code"?
- Research could be done to find quantitative evidence of the importance and criticality of high quality code. Such evidence, if available, could be used to convince management to apply better (more expensive) people to coding asks (cf. solution#1). Such research might use methods similar to those now being used to determine the adverse impacts of Global Warming.
- What are the trends in the evolution of programming languages, and what will be their shorter and longer term impacts on software implementation. One seminar participant remarked that a possible goal of software engineering was to take the fun out of implementation. What is the likely impact of ever-higher-level languages on the implementer?

- Some reports on the impact of possible SE advances assert that ROI (Return On Investment) is much higher for area of SOFTWARE REUSE than for the areas of BETTER TOOLS and BETTER PROCESS. Perhaps research into matters like this would be worthwhile.

The Standard Table

Problem: software implementation (code) quality is lower than desired.

Solution	Effective-ness	Afford-ability	Teach-ability	Use in Practice	Research Needs	Remarks
more implementation emphasis	high	high	--	low - medium	see above	partly mgmnt policy problem, use better people for coding.
better programming techniques	high	high	N/A	low - medium	seek better teach methods, better curricula	partly university policy problem
PSP training	high	high	high	low - medium		rigid training disc., diff or std
code reading training	high	high	medium	low		may lack suitable books, language dep.
better style training	high	high	high	low - medium		language dep.? separate handling of interface
more prog. experience for students	high	high	high	low - medium		each student writes big prog. (5000 LOC)
better mngmt. of prog. lang. and env. change	high	high	high	low - medium		hard to do with changing tech. env.

Integrating New Software with Off-The-Shelf Software

Heinrich Hussman, Pankaj Jalote, Stefan Krauß, Paul Strooper, Walter Tichy, Andreas Zeller

The question

Commercial-Off-The-Shelf (COTS) products have the potential to reduce the cost and time-to-market for software applications. This is one of the main reasons that COTS is now being mandated in Request for Tenders by certain organizations and that some software procurers are asked to justify why they are not using COTS.

To focus our discussion, we first define what we mean by COTS and the type of COTS products that we want to consider. We will use the following definition of COTS:

"A COTS software product provides a core functionality that is potentially useful in many systems. It is provided by a third party organization which is different from the developer and user of the system. This third party is responsible for the quality and evolution of the COTS product and supplies it to system developers on a commercial basis."

The range of COTS products is extreme, including small-scale components such as GUI or component libraries, standard software for commercial applications such as the R/3 software from SAP, and systems software such as operating systems.

Some of the features or properties that are typical of COTS products are:

- the user of the product has no or little control over the evolution of the product
- the product is accessible through a well-defined interface, often an application programmer interface (API)
- a specification of the functionality of the product must exist
- the product is intended for multiple use, and as such it often includes more features than necessary for any particular application
- some customization by the user may be required; COTS products range from products that can be used without change to products that require significant customization

The use of COTS we will focus on is where a portion of an application is not custom-developed, but instead provided by a COTS product. Of course, this only makes sense if the benefits gained by using the COTS product outweigh the costs of finding and integrating the COTS product. Here benefits and costs include issues such as financial cost, time-to-market, etc. In some sense, the question to be answered is "Buy or Build?".

However, before we can decide whether to buy or build, we must know the consequences of using COTS for the various aspects of the development process. This is the problem we will address here: Do we know how we can integrate COTS products into software systems?

Short answer

As we will show below, there are significant problems with the integration of COTS software and there is a lack of documented solutions and experience reports dealing with these problems. We therefore conclude that the problem of integrating COTS products into software products is mainly unsolved.

It should be noted that small-scale COTS products, such as GUI and component libraries, have been used successfully for a long time now. In fact, even though the problems listed below exist with these products to a certain extent, these problems are addressed by fairly standard software engineering practices and no special solutions or techniques are needed in this case.

Tasks and problems

Below we identify the tasks associated with developing an application by integrating COTS software.

1. Analysis and Design
   1.1 Evaluate COTS vendors and products
   1.2 Requirements analysis
   1.3 Determine design/architecture
2. Development/integration
3. Verification and validation
4. Maintenance
5. Management

These tasks are similar to the tasks encountered in the traditional lifecycle, except that we have added tasks for finding suitable COTS products and we have merged the analysis and design tasks. The reason for this is that the traditional lifecycle, where we consider requirements before design, does not work due to potential integration problems with incompatible COTS products. This is why requirements analysis and design, together with the evaluation of COTS vendors and products, have been merged as three overlapping subtasks in a single task. We have also included management as a separate task, because there are several management problems associated with the use of COTS, as explained below.

Clearly the problems associated with the development of software where COTS products are not integrated are also present when we include COTS products. However, there are a number of additional problems:

1. Some traditional process models do not work. As explained above, we do not want to ignore design during requirements analysis. This may lead to architecture mismatch and other interoperability problems during design.
2. Similarly, we cannot expect to complete requirements analysis and then find suitable COTS products to suit those requirements. This will either unnecessarily restrict the types of products that we can use, or increase the cost when implementing the requirements.
3. COTS product evolution, which is under the control of the COTS vendor, and vendor behaviour can have a significant impact. For example, some features may be altered or even deleted in future versions of the product. Not incorporating these future versions may not be an option if older versions are no longer supported by the vendor.

4. Related to the above problem are potential problems with licensing and redistribution of the COTS product.
5. COTS products are typically developed for a large market and as such incorporate many more features than are necessary for any particular application. This causes problems with identifying the useful features, and potential problems with interference of features that are not really needed with those that are.
6. For many COTS products, no source code is delivered. This restricts the type of V&V activities that can be carried out on these products. For example, it makes it harder to guarantee that unwanted features will not interfere with the ones that are needed.
7. Some COTS products have missing or incomplete documentation.

Maturity of solutions

The following tables summarize our opinion of the maturity of the solutions for the tasks and problems identified above. Note that all the problems, except the first, are related to one or more of the identified tasks. We therefore consider only this first problem and then each of the identified tasks. For each task, we list the related problems and the solutions that we are aware of. Many of the entries contain a question mark, indicating that we did not feel that we knew enough about the proposed solution to judge its maturity.

Problem: Traditional lifecycle models do not work

Solution	Effective-ness	Affordability	Teachability	Use in Practice	Research Potential
risk based process model (e.g. spiral model)	?	medium	medium	low	?
IIDA [FLM98, FML98]	?	?	?	low	?

Task 1.1: Evaluate COTS vendors and products

related problems: 3, 4, 5, and 7

Solution	Effective-ness	Affordability	Teachability	Use in Practice	Research Potential
OTSO [Kontio96, Kontio96a]	?	?	?	low	?
[Voas98]	?	medium	medium	low	medium

There is also a proposed acceptance process for COTS software in reactor applications [PS95]. We have not included this in the table above, because it was developed for a particular application domain (the nuclear industry), which has more stringent requirements than typical applications.

Task 1.2: Requirements analysis

related problems: 2, 5, and 7

Solution	Effective-ness	Affordability	Teachability	Use in Practice	Research Potential
Prototyping	medium	medium	medium	low	medium
unify features and requirements (joint appl. development)	medium	low	low	low	low

Task 1.3: Determine design / architecture

related problems: 1, 5, and 7

Solution	Effective-ness	Affordability	Teachability	Use in Practice	Research Potential
life cycle architecture milestone [Boehm96]	?	?	?	low	?
middleware (e.g. CORBA, DCOM)	medium	medium	high	medium	high
wrappers, glue, bridges	low	high	high	low	medium

The Simplex architecture [SGP98] is a proposal for an architecture for incorporating COTS products into trusted systems that attempts to limit the impact of a COTS component that fails on the remainder of the system.

Task 2: Development / integration

related problems: 3, 5, 6, 7

Solution	Effective-ness	Affordability	Teachability	Use in Practice	Research Potential
middleware (e.g. CORBA, DCOM)	medium	medium	high	medium	high
wrappers, glue, bridges	medium	low	high	medium	low

Task 3: Verification & validation

related problems: 5, 6, 7

Solution	Effective-ness	Affordability	Teachability	Use in Practice	Research Potential
use operational experience (for V&V)	low	low	medium	low	medium
black-box unit and system testing	medium	medium	high	medium	low

Kohl [Kohl96] has proposed a method for dealing with dormant code, that is, code that is not required or used in the application in which the COTS product is integrated.

There is also a set of guidelines for the testing of existing software (such as COTS components) for safety-related applications [SL].

Task 4: Maintenance

related problems: 3, 4, 6, 7

Solution	Effective-ness	Affordability	Teachability	Use in Practice	Research Potential
use open comm. standards	low	high	high	medium	low

Task 5: Management

related problems: 1, 3, 4

Solution	Effective-ness	Affordability	Teachability	Use in Practice	Research Potential
COCOTS [COCOTS98]	?	?	high	low	medium

Note that COCOTS, a version of COCOMO tailored for applications that include COTS products, only addresses the cost estimation aspect of the management of software projects that include COTS.

What to teach

Clearly there are many potential problems and very few mature solutions, which poses a problem as far as teaching is concerned. It is also clear that students should be exposed to the potential payoffs of using existing software, and the issues involved in choosing between integrating existing software or writing it from scratch. They should also be aware of existing standards for middleware and interconnection languages.

Perhaps the best way to do this is to emphasize the areas where COTS has been successful: small-scale COTS products such as GUI and component libraries. This could be done by incorporating such products into one or more projects that would typically exist in a Software Engineering curriculum.

Research topics and goals

In this case, the fact that there are many problems and very few mature solutions could indicate that there are significant research opportunities. However, it is not clear to us that we know exactly what all the problems are and how significant they are. This suggests that the most important research that we can carry out is experimental research to assess this.

The Call for Papers for a 1999 ICSE workshop on Ensuring Successful COTS Development lists the following topics: architecture for COTS integration, evaluation techniques for COTS candidates, development process changes, business case development and/or examples, COTS-based system management, maintenance of COTS-based systems, requirements engineering of COTS-based systems, security aspects of COTS-based systems, and tools to support COTS-based development. The call for papers also invites experience reports that provide insight into the advantages and disadvantages of COTS adoption and integration. We feel that the latter will provide much more insight and progress than more proposals for potential solutions.

References

[Boehm96] B. Boehm, Anchoring the Software Process, IEEE Software, July 1996, pp. 73-26.

[COCOTS98] COCOTS - Constructive COTS, http://sunset.usc.edu/COCOTS/cocots.html, August 1998.

[FLM98] G. Fox, K. Lantner, and S. Marcom, A Software Development Process for COTS-based Information System Infrastructure: Part 1. Crosstalk (http://www.stsc.hill.af.mil/CrossTalk/crosstalk.html), March 1998.

[Kohl98] R. Kohl, V & V of COTS Dormant Code: Challenges and Issues. SES'98 Presentation (http://www.rstcorp.com/ots), 1998.

[Kontio96] J. Kontio, OTSO: A Method for Selecting COTS, http://www.cs.umd.edu/~jkontio/reuse.html, August 1996.

[Kontio96a] J. Kontio, A Case Study in Applying a Systematic Method for COTS Selection. Proceedings ICSE-18 Berlin, IEEE Computer Society, 1996.

[PS95] G.G. Preckshot and J.A. Scott, A Proposed Acceptance Process for Commercial Off-the-Shelf (COTS) Software in Reactor Applications. Lawrence Livermore National Laboratory, Report UCRL-ID-122526, September 1995.

[SPG98] L. Sha, J.B. Goodenough, and B. Pollak, Simplex Architecture: Meeting the Challenges of Using COTS in High-Reliability Systems. Crosstalk (http://www.stsc.hill.af.mil/CrossTalk/crosstalk.html), April 1998.

[SL] J.A. Scott and J.D. Lawrence, Testing Existing Software for Safety-Related Applications. Lawrence Livermore National Laboratory, Report UCRL-ID-117224 (revision 7.1).

Test

Motoei Azuma, Karol Fruehauf, Dan Hoffman

What is testing?

The repeatable execution of software with the intention of revealing deviations from requirements, including functional, performance, and reliability requirements.

What is the state of the practice and the state of the art (briefly)?

State of the practice

- Many companies do no systematic testing. Some companies do perform systematic testing; it is often reasonably effective but very expensive in effort and calendar time. Unit testing is usually ad hoc, if performed at all. Test automation is primitive.
- Test education, especially in universities, is poor.

State of the art

- Sophisticated automation is feasible and can be very effective, as shown by Microsoft. Statistical testing can be used effectively, as shown by AT&T. Given the importance of testing to industry, there is relatively little research done. What is done focuses primarily on automated input generation from source code or formal specifications.
- Some good training materials do exist, mostly in industry.

What are the most important dimensions?

The topic can be investigated along a number of dimensions. Each dimension would most likely lead to a different answer to the basic questions. Therefore we will restrict our considerations to the life cycle dimension (but mention the other dimensions which could have been investigated).

lifecycle

unit, integration, system, field

lifecycle model applied

waterfall, iterative, incremental, spiral

focus

correctness, performance, usability, reliability

development paradigm

procedural, OO, AI

domain

information systems, real-time/embedded systems, OS/network, DBMS, telecom

mode of operation

batch vs. interactive (GUI)

Task: system testing

Subtask	Effectiveness	Affordability	Teachability	Penetration	Research potential
Design for testability	low	medium	low	innovators	high
Test planning	high	medium	medium	early majority	low
Selection: deterministic	medium	medium	medium	early majority	medium
Selection: statistical	high	low	medium	innovators	high
Oracle	medium	low	high	early majority	high
Test environment	high	medium	low	late majority	low
Test report	high	high	high	early majority	none

Notes

General

Systematic testing only practiced by early majority, limiting the penetration in many cases above.

Design for testability

An important topic in testing that, arguably, belongs under the “design” topic.

Test planning

Careful planning is essential for systematic testing

Selection: deterministic

The focus to date has been automated input generation.

While some interesting results have been obtained over the past 20+ years, there has been little industrial impact.

Selection: statistical

Seems promising but lack of experience makes all entries suspect.

Will probably be very effective in some domains; ineffective in others.

Oracle

At present, usually derived manually from the requirements specification, at high cost.

Test environment

Some participants felt that automation is the main route to improvement here, of input generation, execution, and oracle; others disagreed.

Test report

Usually a manual task, though automated clerical support is sometimes present.

Task: integration testing

Subtask	Effectiveness	Affordability	Teachability	Penetration	Research potential
Design for testability	high	medium	low	innovators	high
Test planning	high	low	low	innovators	medium
Selection: deterministic	medium +	medium +	medium +	early adopters	low
Selection: statistical	low	low	medium	none	low
Oracle	low	low	low	innovators	?
Test environment	high	low	low	innovators	high
Test report	high	high	high	innovators	none

Notes

General

Integration testing is poorly understood by researchers and practitioners. There is little agreement on the basic issues, terminology, and principles. Consequently, confidence in this table is low.

Nonetheless, the topic is important. There is lots of experience but it is not packaged for teaching or use. There is a big potential payoff in getting a firmer understanding of the area.

Design for testability

Really belongs under Structures.

Rarely explicitly practiced but potential is high.

Test planning

The key issue is planning the integration order, especially in large projects.

Selection: deterministic

Usually based on design specifications.

Selection: statistical

May help in some large subsystems.

Input generation is not hard but it makes the oracle very expensive.

Oracle

Automation has great potential here.

Test environment

Configuration management support for subsystem builds is important.

Support for many test configuration items is needed

Test report

A manual task.

Task: unit testing

Subtask	Effectiveness	Affordability	Teachability	Penetration	Research potential
Design for testability	low	?	low	innovators	medium
Test planning	high	medium	medium	innovators	low
Selection: deterministic	medium +	medium +	medium +	early adopters	low
Selection: statistical	low	low	medium	none	low
Oracle	low	low	high	late majority	high
Test environment	high	medium	medium	innovators.	low
Test report	high	high	high	innovators	none

Notes

General

Systematic unit testing rarely done.

Design for testability

Again, really belongs under "Design".

Test planning

Systematic unit testing is done rarely because it is not explicitly planned as an activity. The specification of the unit test cases rarely appears in a project plan.

Selection: deterministic

Tool support can be improved.

Selection: statistical

Input generation is not hard but it makes the oracle very expensive.

Oracle

Automation has great potential here.

Test environment

Automation is the main route to improvement here, of input generation, execution, and oracle.

Test report

A manual task.

What should be taught?

- What testing is and is not.
- How testing differs from "playing with the program".
- The fundamental principles and limitations of testing.
- What to test, when, and how.
- Test planning and reporting.
- The key payoff of unit test: catch errors before integration test.
- The key payoff of integration test: catch errors before system test.
- Test approaches.
- Test tools.
- Statement coverage, test management.
- *Overall*: reduce methods not proven in practice to "a mention".

Where should research be focused?

- Design for testability.
- Work needed to understand and express "testability".
- Big payoff in careful thought, experimentation, and refinement.
- Oracle automation (and automation of other areas).
- Little work so far, especially in practical situations.
- Lots of potential for improvement.
- Statistical test case selection.

Defining Software Families

Wolfram Bartussek, Paul Strooper, David Weiss

Introduction

Family-oriented software development was suggested as early as 1968 0. More recently there have been several suggestions for engineering families based on the idea of identifying abstractions that are common to a family and using them as the basis for designing a specification language for describing family members and for creating a design common to all family members 0, 0, 0, 0, 0, 0, 0. A variety of technologies may then be used to implement a translator for such a language or to use the design to create family members rapidly. The general goal of all of these approaches is to make software development more efficient by specializing for a particular domain (where we take a domain to be a family) the facilities, tools, and processes that you use to produce software for the domain. You make an investment in specialized facilities, tools, and processes that you might otherwise not, but your investment is repaid many times over in increased efficiency in your software development process.

Software Families

The original definition of a program family as given by Parnas [1] is “We consider a set of programs to constitute a family whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members.”

This definition is utilitarian since the use of the word "worthwhile" suggests that one must do a value analysis before deciding whether or not to consider a set of programs to be a family, but otherwise says little about the commonality among family members. In current practice, when software families are identified, the identification is likely to be based on experience and intuition, rather than by a systematic procedure. This approach corresponds more to defining a family as a set of items that have common aspects and predicted variabilities. Emphasis here is on having made the commonalities explicit and on predicting the variabilities among family members.

We will distinguish among several stages in defining a family.

Stage 1: Potential family. A set of software for which one suspects that there is sufficient commonality to be worth studying the common aspects of the software.

Stage 2: Semi-family. A set of software for which common and variable aspects have been identified.

Stage 3: Defined family. A semi-family for which an economic analysis has been performed in order to decide how much of an investment should be made in the family, i.e., an investment to exploit the commonalities and variabilities for the purpose of efficiently creating family members.

For purposes of investment it is often worth considering a special case: a product line, which is a family of software systems where individual members of the family are sold as products (instead of being incorporated into products or used to create products).

It is rare that you get the chance to create a family afresh. Most families are the result of success: a company develops a successful product and discovers that customers want changes to the product. To continue to sell instances of the product and to keep up with the market for it, the company finds that it is producing many variations on it. The company realizes that it has a product line, i.e., a semi-family of software systems, or systems that need software to operate, where individual members of the family are sold as products. (Alternatively, a software development organization may discover that it has a set of software assets that are used in building members of its product line(s) but that themselves are not packaged to be sold as products. An example is a database system used in a telephone switch. The same system may be used in many switches, but is not sold as a separate product.)

Most software families, then, evolve from existing systems. Even when an organization realizes that it has a product line that is a semi-family, it still rarely has methods that allow it to guide the evolution of the product line. It lacks reasonable methods for deciding what the short and long term evolution should be, for identifying information critical to the software developers for creating software families, for informing the developers of that information, and for managing their decisions over time. From a business viewpoint, this is strategic software planning. From a technical viewpoint, it is strategic software design.

In many engineering disciplines the idea of investing in a product line is a natural one and is part of the product line development process. Automobile manufacturers make strategic decisions about investing in assembly lines and factories as a standard way of conducting business. They plan investments in technology and process, and know the expected return from those investments. Manufacturing engineers are involved in such decisions and design the manufacturing process accordingly. Such infrastructure investment planning is missing in most software development processes, which often focus on the near term production of a single system.

The Question: How To Define A Family

For software families, there are few techniques available for deciding what the members of a defined family should be, i.e., for moving from a potential family to a defined family. Without performing such an analysis, it is difficult to decide how much investment to make in the family, and it is difficult to create the resources needed for creating family members, such as design, code, and tools for creating family members. Note that these issues involve both economic and technical issues concerning the feasibility of creating the resources. This section is concerned with identifying the tasks involved in identifying (and documenting) defined families.

The Short Answer

In general, few techniques exist for defining families, i.e., for performing the analysis needed to identify a defined family. There are a few techniques that have become commercially available in the last 2-3 years, but they are just starting to be tried by the early adopters in industry. As a result, we rate the problem as partially solved. The following sections provide more detail on what types of techniques are available, and the reasons we have ranked them as they appear .

Aspects Of Defining A Family

There are two key aspects of identifying a defined family: the tasks to be performed and the means for performing those tasks. The tasks involve the actions to be taken, and the means involve the methods for performing those tasks and the artifacts that result from applying those methods.

Task Aspect

1. Elicitation of common and variable requirements
2. Description of common and variable requirements
3. Validation of common and variable requirements
4. Investment analysis - cost benefit of including family members, and cost benefit of infrastructure investment
5. Management

Means Aspect

1. Languages
2. Processes
3. Tools
4. Heuristics

Evaluation of Means for Individual Tasks

Problem: Elicitation of common and variable requirements

Solution	Effective-ness	Affordability	Teachability	Use in Practice	Research Potential	Confidence
Commonality Analysis	High	Medium	High (teach!)	Low (EA)	Medium	High
Scenarios	Medium	Medium	High	Low (EA)	Medium	Medium
Model Building	Low	Low	Medium	Low (IN)	Medium	Medium
Organization Domain Modeling	Medium	?	?	Low (LU)	?	Low
Summary	High	Medium	High	Low	High	Medium

Research might focus on investigating combinations of the identified solutions

Problem: Description of common and variable requirements

Solution	Effectiveness	Afford-ability	Teachability	Use in Practice	Research Potential	Confidence
Precise Use of Natural Language	Medium (highly effective for communicating ideas, less effective for further formal treatment)	High	High (teach!) (much time needed for disciplined use of nat. lang. for engineering purposes)	Low (EA)	High	Medium
Table of Parameters	Medium	Medium	High (teach!)	Low (EA)	Medium	Medium
Domain Models (state trans., context models, object diagrams,...)	Low	Low	Medium	Low (EA)	High	High
Pattern Language	?	?	Medium (dare to teach)	Low (IN)	High	Low
Semantic Networks	?	?	Low	Low (LU)	?	Low
Summary	Medium	High	High	Low	High	Medium

Combination of the first three items may yield a solution of very high pay off

Problem: Validation of common and variable requirements

Solution	Effective-ness	Afford-ability	Teachability	Use in Practice	Research Potential	Confidence
Prototyping	Medium	Low	?	Low (IN)	Medium	Medium
Formal reviews	Medium	Medium	High (teach!)	Low (EA)	Low	High
Examples from history	Medium	High	High (teach and train!)	Low (EA)	High	High
Market research	Medium	Low	Low	Low (EA)	High	Low
Summary	Medium	Medium	High	Low	High	Medium

Formal reviews: understandability (experts from the field, not involved in the project), active reviews, inspections, desk reading

Examples from history: More effective than other techniques but in a narrow range

Problem: Investment Analysis

Solution	Effective-ness	Affordability	Teachability	Use in Practice	Research Potential	Confidence
PULSE-ECO Product line SE	Medium	Medium	Medium (teach and train)	Low (IN)	High	Low
Value Proposition	Medium	Medium	Medium (hire economist)	Low (EA)	High	Low
FAST cost analysis	Medium	High	High (train!)	Low (EA)	Medium	Medium
Summary	Medium	Medium	Medium	Low	High	Low

High potential, little work done, but may able to use standard marketing and economic techniques

Problem: Management Analysis (managing the process to find the family)

Solution	Effectiveness	Affordability	Teachability	Use in Practice	Research Potential	Confidence
Summary	Medium	High	Low (include in management training)	Low (EA)	?	Low

Standard management techniques used; identification of roles and resources

Summary of Research Issues

The most promising areas for research are those ranked as high:

- Precise use of natural language for description of common and variable requirements,
- Domain models for description of common and variable requirements,
- Pattern languages for description of common and variable requirements,
- Use of historical examples for validation of common and variable requirements,
- Market research for validation of common and variable requirements,
- Use of PULSE-ECO for investment analysis, and
- Creation of value propositions for investment analysis.

In addition, research into integrating several different means of accomplishing tasks may lead to more effective, more affordable, and more appealing versions of tasks. In particular, precise use of natural language combined with tables of parameters and model building may lead to a very effective and affordable technique for description of family requirements.

Summary of Teaching Issues

The most promising areas for teaching are those ranked as high:

- Commonality analysis for elicitation of common and variable requirements,
- Precise use of natural language for description of common and variable requirements,
- Use of formal reviews for validation of common and variable requirements, and
- FAST cost analysis for investment analysis.

References

[1] Campbell, Grady H. Jr., Faulk, Stuart R., Weiss, David M.; Introduction To Synthesis, INTRO_SYNTHESIS_PROCESS-90019-N, 1990, Software Productivity Consortium, Herndon, Dijkstra, E. W., Notes on Structured Programming. Structured Programming, O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare, eds., Academic Press, London, 1972

[2] Coglianese, L, Tracz, W.; An Adaptable Software Architecture for Integrated Avionics, Proceedings of the IEEE 1993 National Aerospace and Electronics Conference-NAECON 1993, Jun, 1993

[3] Cuka, D., Weiss, D.; Engineering Domains: Executable Commands As An Example, Proc. International Conference On Software Reuse, June, 1998

[4] Dijkstra, E. W., Co-operating Sequential Processes, Programming Languages, ed. F. Genuys, New York: Academic Press, pp. 43-112, 1968

[5] Gupta, N., Jagadeesan, L., Koutsofios, E., Weiss, D.; Auditdraw: Generating Audits the FAST Way, IEEE International Symposium on Requirements Engineering, pp. 188-197, January, 1997

[6] Kang, K., Cohen, S., et al., Feature Oriented Domain Analysis (FODA) Feasibility Study, Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Pittsburgh, PA, November, 1990

[7] Moore, Geoffrey, Crossing the Chasm

[8] Neighbors, J., The Draco Approach to Constructing Software from Reusable Components, IEEE Transactions on Software Engineering, SE-10, 1984

[9] Parnas, D.L., On the Design and Development of Program Families, IEEE Transactions on Software Engineering, SE-2:1-9, March 1976

[10] Simos, Mark, Anthony, Jon, Weaving the Model Web: A Multi-Modeling Approach to Concepts and Features in Domain Engineering, Proc. Fifth Int. Conf. Software Reuse, June, 1998

[11] Weiss, David M., Lai, Robert Chi Tau, Engineering Software Domains: A Family-Based Software Development Process, Addison-Wesley-Longman, July 1999

Maintenance

Jan Bredereke, Karol Frühauf, Ridha Khedri, Stefan Krauß, Andreas Zeller

The question

How can we maintain a product's integrity during its evolution?

The answer

This problem is partially solved.

Reasons for the ranking

There is a number of well-understood solutions that help in software maintenance. However, there is a need and a potential for better solutions, especially in the areas of reverse engineering and reengineering.

Details

The *aim* of maintenance is to change a software product after it has been released to the user; the *problem* is to ensure the integrity of the product even after a change. As the number of old software systems increases, so does the need for maintenance.

Designing for change is the best way to easy maintenance.

Prior to any change comes the problem the structure oft the product needs to be understood and the potential impact of the intended change determined. This is typically supported by the product documentation. Where the documentation is incomplete or inaccurate, program comprehension can be supported by *reverse engineering* tools.

If the required change is sufficiently large and has an impact on the overall structure of the program then the product need to be *reengineered.*

After a change to the product has been accomplished, *regression testing* helps in detecting unwanted side effects. For the whole maintenance activity, *software configuration management* is a necessity to keep track of the product evolution.

The following table lists a number of proven and proposed solutions that can help reducing maintenance costs.

Solution	Effective-ness	Affordability	Teachability	Use in Practice	Research Potential
1. Configuration Management (see also extra working group)	high	high	high	early majority	low
2. Regression Testing	high	medium	high	early adopters	medium
3. Reverse Engineering					
3.1 Peopleware					
3.1.1 Talk to knowledgeable People	high	medium	low	late majority	high
3.2 Static Analysis					
3.2.1 Lexical Analysis (i.e. cross reference)	high	high	high	early majority	medium
3.2.2 Syntactic Analysis	?	medium	medium	pioneer	high
3.2.3 Semantic Analysis	?	low	low	lab use	high
3.3 Dynamic Analysis (i.e. program spectra)	?	?	?	?	?
3.4 Complexity Analysis	medium	high	high	pioneer	low
3.5 Reconstructing Abstractions	medium	?	?	?	?
4. Reengineering					
4.1 Restructuring Code	low	medium	high	innovators	low
4.2 Restructuring Modules	medium	low	low	innovators	high
4.3 Wrapping Legacy Systems	high	medium	high	early adopters	low
4.4 Software Migration	medium	high	low	late majority	high
5. Designing for Change (see also extra working group)	high	medium	high	early adopters	medium
6. Documentation Maintenance	medium	high	low	late majority	high

As the expertise of the authors mainly covers configuration management and regression testing, the confidence level of the judgements for the areas 2-6 is significantly lower. Several of these judgements need to be supported by experience data; gathering such evidence is as useful a research topic as gathering more knowledge about the individual solutions.

What should be taught?

As “best practices”, the solutions with high effectiveness and high teachability should be part of any software engineering course:

- Software Configuration Management
- Regression Testing
- Lexical Analysis
- Wrapping Legacy Systems
- Designing for Change (the little we know)

What research should be done?

First of all, we should evaluate existing solutions for effectiveness, if this has not already been done. Besides developing alternate solutions, the following tasks have high research potential:

- Extracting and Evaluating People's Knowledge
- Syntactic and Semantic Analysis
- Restructuring Modules
- Designing for Change (all that we don't know)
- Software Migration
- Documentation Maintenance

The potential of the following tasks was not judged due to lack of knowledge:

- Dynamic Analysis
- Reconstructing Abstraction

Suggested Readings

[1] Reidar Conradi and Bernhard Westfechtel. Version Models for Software Configuration Management. ACM Computing Surveys 30(2), June 1998, pp. 232-282.

A survey article on the current state of the art in configuration management.

[2] Akira K. Onoma and Wei-Tek Tsai and Mustafa H. Poonawala and Hiroshi Suganuma. Regression Testing in an Industrial Environment. Communications of the ACM 41(5), May 1998, pp. 81-86.

A recent survey on regression testing covering several practical aspects.

[3] Helmut Balzert. Lehrbuch der Software-Technik, Band 2. Spektrum-Verlag, Heidelberg, 1997.

A recent software engineering textbook that discusses recent maintenance and reengineering issues.

[4] Hausi Müller and Thomas Reps and Gregor Snelting (eds.). Program Comprehension and Software Reengineering. Dagstuhl Seminar-Report #204. http://www.dagstuhl.de/DATA/Seminars/98/#98101

A survey on current research topics in Reverse Engineering and Reengineering.

Measurement

Motoei Azuma, Pankaj Jalote, Peter Knoke, Jochen Ludewig

Introduction

Why are metrics important? Because:

1. In order to do anything scientifically the target product as well as process should be measurable.
2. In order to improve anything, the effects must be observed, i.e. measured.
3. In order to measure something, metrics should be well defined, validated and standardized.

Basic Questions

Specifically, when dealing with software using metrics, we can assess metrics with the following basic questions:

- Are we able to describe and forecast process and product characteristics by metrics?
- Are we able to control process and product using metrics, possibly continuously?

Overall Answer

Metrics can be categorized into process metrics and product metrics based on the target attributes to be measured. There are some books and papers on process and product metrics. Some metrics are widely known and sometimes, used in practice. ISO/IEC JTC1/SC7 is keen to develop international standards for measurement. JTC1/SC7/WG6 is developing a series of international standards for software product metrics and evaluation. Examples are shown in the bibliography. Yet there are many metrics to be developed, validated and taught for practical use.

Therefore the overall answer to the above questions is:

Partially Solved.

Concept of Metrics and Measurement

Definitions from ISO/IEC 14598-1 (Information Technology - Software product evaluation - Part 1: General overview)

Metric: the defined measurement method and the measurement scale.

Measurement: the use of a metric to assign a value (which may be a number or category) from a scale to an attribute of an entity.

Measure (Noun): The number or category assigned to an attribute of an entity by making a measurement.

External measure: an indirect measure of a product derived from measures of the behavior of the system of which it is a part.

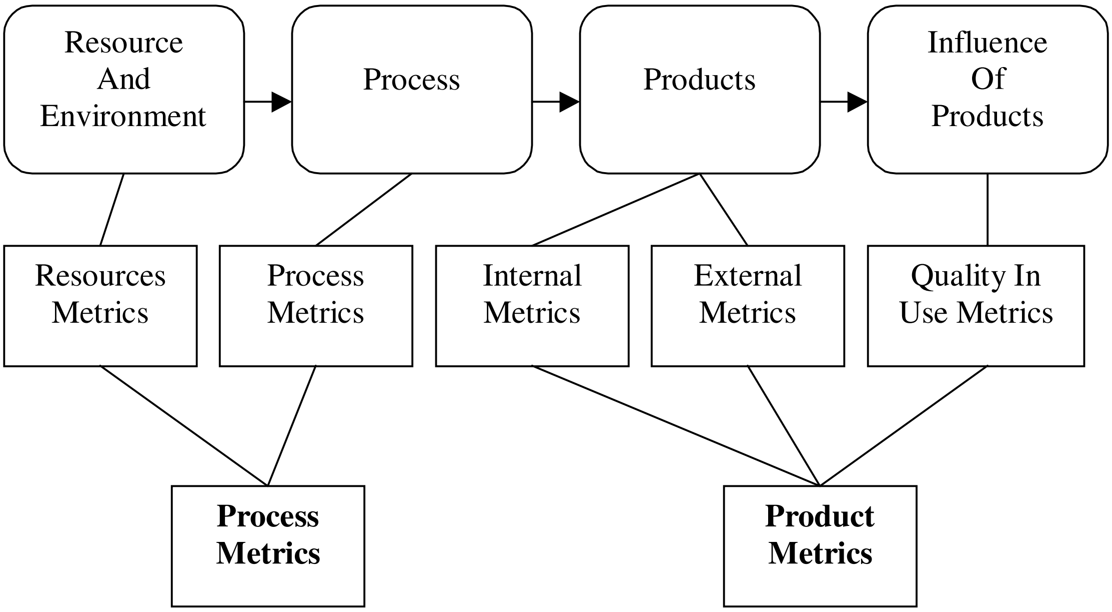
Internal measure: a measure of the product itself, either direct or indirect.

Taxonomy (1): Categories based on attributes to be measured

Metrics can be categorized by target attributes to be measured.

The highest level of categories is process metrics and product metrics.

They are refined into detailed levels of categories. Process metrics includes those that measure process as well as resources. Product metrics are categorized into internal metrics, external metrics and quality-in-use metrics. (See Reference Model)



Taxonomy (2): Usage and timing of use

Metrics can be also categorized by their usage.

Initial Stage: At this stage metrics are used for description of plan (or requirement) for both process and product.

Current Stage: At this stage metrics are used for description of current (at a milestone) process or product. Resulting value (measure) is used for control or as input for forecasting process or product at goal stage.

Future Stage (Goal): Metrics for future stage are those which forecast (or predict) Estimated Process or Product at Goal Stage.

Framework of Metrics by Target Attributes and Usage

Target attribute and usage are used for categorizing measurement technologies. Measurement technologies include metrics as well as process for measurement and their use.

Table 1: Categories of Metrics and Assessment Results Clauses

	Planning	Current	Estimate Forecast
Process	Process-Planning	Process-Current State	Process-Forecast
Product	Product-Planning	Product-Current State	Product-Forecast

Technology Assessment Framework

The framework

Solution indicates examples of metrics in the category or associated information.

Metrics and associated method and tools were assessed by the following criteria:

- Technical Maturity (Effectiveness).
- Practical Maturity (Use in practice).
- Affordability.
- Teachability (Education needs).
- Research potential (Research needs).

Technical Maturity

How accurate or correct it is?

High: Scientifically proved
Med-High: Statistically validated
Medium: Accessed by experts
Med-Low: Proved to be useful by experimental use
Low: Proposed

Practical Maturity

How widely it is known and used?

High: Widely used in practice
Med-High: Widely accepted, but used by limited users
Medium: Widely known, but used by limited users
Med-Low: Experimentally used by some users
Low: Proposed, but known by small

Affordability

How reasonable are the cost and effort required for using it?

Teachability (Education needs)

How easy is it to teach it in university or industry? Is it necessary to teach it?

Research potential (Research needs)

Is research for new technologies necessary?

Assessment Results

Process / Planning

Metrics and measurement technologies for planning a process.

(1) Sub-category: Resources estimation

Solution (Metric or Measurement Process): Cost models such as COCOMO

Technical Maturity (Effectiveness):	Useful but limited
Practical Maturity (Use in practice):	Medium
Affordability:	Medium
Teachability (Education needs):	High
Research potential (Research needs):	(1) Model with easy inputs that is applicable in starting stages.

(2) Sub-category: Schedule estimation

Solution (Metric or Measurement Process): Schedule estimation model

Technical Maturity (Effectiveness):	High
Practical Maturity (Use in practice):	Med-High
Affordability:	High
Teachability (Education needs):	High
Research potential (Research needs):	(1) Minimum schedule for a given project and effort estimate, (2) Relationship between schedule compression and effort escalation.

(3) Sub-category: Measurement process

Solution (Metric or Measurement Process): JTC1/SC7/WG13, ESPRIT project AMI

Technical Maturity (Effectiveness):	High
Practical Maturity (Use in practice):	Med-High
Affordability:	High

Teachability (Education needs): High
Research potential (Research needs): Not research topic

Product / Planning

Metrics and measurement technologies for planning products.

(4) Sub-category: Functionality

Solution (Metric or Measurement Process): Validated Function

Technical Maturity (Effectiveness):
Practical Maturity (Use in practice):
Affordability:
Teachability (Education needs):
Research potential (Research needs):

(5) Sub-category: Reliability

Solution (Metric or Measurement Process): MTTF (Mean Time To Failure), Mean time for recovery, etc.

Technical Maturity (Effectiveness): High
Practical Maturity (Use in practice): Low
Affordability: Low
Teachability (Education needs): High
Research potential (Research needs): (1) Relationship between defect density and MTTF.

(6) Sub-category: Usability

Solution (Metric or Measurement Process): Time to learn, Mistakes made in using the system, User satisfaction

Technical Maturity (Effectiveness): Low
Practical Maturity (Use in practice): Low
Affordability: Low
Teachability (Education needs): Medium
Research potential (Research needs): High. Need simple, cheap, and usable metrics (e.g. which can have tool support)

(7) Sub-category: Efficiency

Solution (Metric or Measurement Process): Response time, Throughput, Memory requirement.

Technical Maturity (Effectiveness): High
Practical Maturity (Use in practice): Medium-high
Affordability: Medium
Teachability (Education needs): High
Research potential (Research needs):

(8) Sub-category: Maintainability

Solution (Metric or Measurement Process): MTTR, effort required to make changes, regression testing effort.

Technical Maturity (Effectiveness): Medium
Practical Maturity (Use in practice): Low
Affordability:
Teachability (Education needs):
Research potential (Research needs): Medium (relationship between user level metrics and internal/process metrics needs to be understood)

(9) Sub-category: Portability

Solution (Metric or Measurement Process): Effort needed to port to a new hardware/software platform.

Technical Maturity (Effectiveness): Low
Practical Maturity (Use in practice): Medium
Affordability:
Teachability (Education needs):
Research potential (Research needs):

Process / Current State

Metrics and measurement technologies which describe the current state of the process.

(10) Sub-category:

Solution (Metric or Measurement Process): Effort expended, time consumed, tasks completed, defects found, no. of changes (revisions) made, no. of compilations, etc.

Technical Maturity (Effectiveness): Medium
Practical Maturity (Use in practice): Medium
Affordability: Medium
Teachability (Education needs):
Research potential (Research needs): (1) Models for forecasting user-level properties from these process measures,
(2) Rules for interpreting the data.

Product / Current State

Metrics and measurement technologies which describe the current state of the products.

(11) Sub-category: Functionality

Solution (Metric or Measurement Process): Amount of functionality built

Technical Maturity (Effectiveness): Low
Practical Maturity (Use in practice): Medium
Affordability:

Teachability (Education needs):
Research potential (Research needs): (1)Tools support for tracing current system to requirements/design.

(12) Sub-category: Reliability

Solution (Metric or Measurement Process): No of errors found, timing of errors, No of errors removed, code coverage in testing, percentage of test cases that succeeded.

Technical Maturity (Effectiveness): High
Practical Maturity (Use in practice): Medium
Affordability: High
Teachability (Education needs): High
Research potential (Research needs):

(13) Sub-category: Usability

Solution (Metric or Measurement Process): Clarity of messages

Technical Maturity (Effectiveness): Low
Practical Maturity (Use in practice): Low
Affordability:
Teachability (Education needs):
Research potential (Research needs): Need intermediate metrics for controlling usability.

(14) Sub-category: Efficiency

Solution (Metric or Measurement Process): Measure response time, analyze design, benchmarking.

Technical Maturity (Effectiveness): Measurement is OK, analysis is not.
Practical Maturity (Use in practice): Medium
Affordability:
Teachability (Education needs):
Research potential (Research needs): Analyze a design for runtime properties.

(15) Sub-category: Maintainability

Solution (Metric or Measurement Process): Complexity of design, code, etc.; modularity; structure; OO metrics.

Technical Maturity (Effectiveness): Low
Practical Maturity (Use in practice): Low
Affordability:
Teachability (Education needs):
Research potential (Research needs): High

(16) Sub-category: Portability

Solution (Metric or Measurement Process): Amount of hw/sw dependent code; percentage of modules that have this code.

Technical Maturity (Effectiveness): Low
Practical Maturity (Use in practice): Low
Affordability:
Teachability (Education needs):
Research potential (Research needs): High

Process / Forecast

Metrics and measurement technologies for forecasting properties of the remaining part of the process.

(17) Sub-category:

Solution (Metric or Measurement Process): Time to complete, effort to complete, defect density on completion, other product quality properties, productivity.

Technical Maturity (Effectiveness): Medium
Practical Maturity (Use in practice): Medium
Affordability:
Teachability (Education needs):
Research potential (Research needs): Medium

Product / Forecast

Metrics and measurement technologies for forecasting a product quality at goal stage.

(18) Sub-category: Functionality

Solution (Metric or Measurement Process): Amount of functionality that will be delivered.

Technical Maturity (Effectiveness): Low
Practical Maturity (Use in practice): Low
Affordability:
Teachability (Education needs):
Research potential (Research needs):

(19) Sub-category: Reliability

Solution (Metric or Measurement Process): MTTF, defect density, recovery time

Technical Maturity (Effectiveness): Medium
Practical Maturity (Use in practice): Low-medium
Affordability:
Teachability (Education needs):
Research potential (Research needs): Tools/models for identifying potential reliability problems; potentially difficult modules; Improved prediction models. Can metrics be used to identify hot-spots?

(20) Sub-category: Usability

Solution (Metric or Measurement Process): ??

Technical Maturity (Effectiveness):
Practical Maturity (Use in practice):
Affordability:
Teachability (Education needs):
Research potential (Research needs): high

(21) Sub-category: Efficiency

Solution (Metric or Measurement Process): Models for predicting resp. time, etc.

Technical Maturity (Effectiveness):
Practical Maturity (Use in practice):
Affordability:
Teachability (Education needs):
Research potential (Research needs): High. Predicting run-time performance of a design, or code, for some environment. Tools to identify possibilities of performance improvement.

(22) Sub-category: Maintainability

Solution (Metric or Measurement Process): ??

Technical Maturity (Effectiveness):
Practical Maturity (Use in practice):
Affordability:
Teachability (Education needs):
Research potential (Research needs): Given process data and product data, can maintainability be predicted. Can we use metrics to identify possibilities for improving maintainability.

(23) Sub-category: Portability

Solution (Metric or Measurement Process):??

Technical Maturity (Effectiveness):
Practical Maturity (Use in practice):
Affordability:
Teachability (Education needs):
Research potential (Research needs): Can portability of design be estimated, Can portability of code be estimated? Can we identify modules/deign elements for improving portability.

Summary and Recommendations

Education Needs

What should be taught in universities? Focus on mature product metrics and how they can be used for improving product development.

What should be taught in industry? Both product and process metrics; Building models and calibrating models; Using metrics for project management (planning and control).

Research Needs

1. Validate as yet unvalidated metrics (using experiments and statistical techniques).
2. Build forecasting models (identify measurable metrics and how they can be used to predict desired characteristics).

Bibliography

[1] ISO/IEC 9126-1: Software product quality - Part 1: Quality model
[2] ISO/IEC 9126-2: Software product quality - Part 2: External metrics
[3] ISO/IEC 9126-3: Software product quality - Part 3: Internal metrics
[4] ISO/IEC 9126-4: Software product quality - Part 4: Quality in use metrics
[5] ISO/IEC 14598-1: Software product evaluation - Part 1: General overview
[6] ISO/IEC 14756: Measurement and rating of performance of computer based software system
[7] ISO/IEC 15504-1: Software process assessment - Part 1: Concepts and introductory guide
[8] ISO/IEC 15504-2: Software process assessment - Part 2: A reference model for process capability
[9] Fenton and Pfleeger, Software Metrics: A rigorous and Practical Approach, International Thomson Computer Press, 1996
[10] Natale, D. Qualita E Quantita Nei Sistemi Software (Italian), FrancoAngeli, 1995
[11] Lorentz and Kidd, Object-Oriented Software Metrics – A practical Guide, Prentice Hall 1994
[12] Azuma, M (Editor), Software Quality Evaluation Guide Book (Japanese), JISA, 1994
[13] Moller, K.H. and Paulish, D.J. Software Metrics, Chapman & Hall (IEEE Press), 1993
[14] Dumke, R. Softwareentwicklung nach Masz (German), Vieweg, 1992
[15] Zuse, Software Complexity - Measures and Methods, Walter de Gruyter, 1991

Software Configuration Management

Karol Fruehauf, Walter Tichy, Andreas Zeller

The question

How can we manage and control the evolution of a product?

The answer

This problem is solved.

Reasons for the ranking

Software configuration management is a well-understood discipline that offers a number of solutions for managing software evolution. Some areas still offer room for improvement.

Details

Software configuration management (SCM) is a sub-discipline of software engineering. Its goal is to keep order in long-lived, multi-person software projects. It does so by controlling and recording the evolution of software products through their entire lifecycles, from requirements and development to test, deployment, operation, all the way through maintenance and upgrades.

Solutions and Tasks	Effective-ness	Affordability	Teach-ability	Use in Practice	Research Potential
Version management for individual items (revisions, branches, variants, checking / checkout, sandboxes, identification)	high	high	high	early majority	low
Version management for structures (renaming, reorganization, retiring of subsystems with whole history)	medium	high	hi	pioneer	medium
Configuration definition (parts lists, baselines plus change sets, generic configurations, version selection)	high	high	high	early majority	low

Solution and Tasks	Effective-ness	Affordability	Teach-ability	Use in Practice	Research Potential
Build management (Make and Make-oids, derivation history)	high	high	high	late majority	low
Automated change management (change requests, change request tracking, task assignment)	medium	high	high	early majority	low
Process support	low	?	?	lab use	?
Traceability (implementation → design → requirements)	high?	?	?	lab use	?
LAN connectivity	high	low	high	late majority	?
Internet connectivity (remote access, replication, caching)	medium	?	high	early adopters	medium
Support for geographically distributed collaboration (parallel work as the rule, automated merging)	low	?	?	lab use	high
Software distribution and installation on the internet, esp. for distributed applications					
for distribution	high	low	high	late majority	low
for updates	low	?	?	pioneer	high
Dynamic reconfiguration	high?	?	?	pioneer	high

What should be taught?

Software Configuration Management should be part of any software engineering course. The functionality of SCM tools like RCS, CVS, or MAKE should cover almost all needs of a mid-size (i.e. undergraduate-level) software project.

What should we research?

Most parts of software configuration management are well-understood and well-automated. Distributed software configuration management still requires more attention. With the advent of loosely connected components over a network, it is expected that static configuration (at build time) will lose importance in favour of dynamic configuration (at run time), which is still a research topic. Also, composition of complex systems from versioned components brings problems with configuration consistency, which must be identified.

Suggested Readings

[1] Reidar Conradi and Bernhard Westfechtel. Version Models for Software Configuration Management. ACM Computing Surveys 30(2), June 1998, pp. 232--282.

A survey article on the current state of the art in configuration management.

[2] Walter F. Tichy (ed.). Configuration Management. John Wiley & Sons, Chichester, UK, 1994.

A collection of articles covering the state of the art and state of the practice in software configuration management.