SOFTWARE ENGINEERING METHOD AND THEORY – A VISION STATEMENT

By Ivar, Bertrand and Richard

1 Purpose and scope

The original Semat Call for Action gave a broad definition of the problem that the Semat initiative is poised to address:

Call for Action

Software engineering is gravely hampered today by immature practices. Specific problems include:

- The prevalence of fads more typical of fashion industry than of an engineering discipline.
- The lack of a sound, widely accepted theoretical basis.
- The huge number of methods and method variants, with differences little understood and artificially magnified.
- The lack of credible experimental evaluation and validation.
- The split between industry practice and academic research.

We support a process to refound software engineering based on a solid theory, proven principles and best practices that:

- Include a kernel of widely-agreed elements, extensible for specific uses
- Addresses both technology and people issues
- Are supported by industry, academia, researchers and users
- Support extension in the face of changing requirements and technology

To develop appropriate solutions, we need a more precise framework. The present vision statement is this framework, similar to a "requirements statement" and "road map" in software, including a list of expected goals for the first year.

The participants in the effort are coming from many sides of the profession; they include practicing programmers, project managers, consultants and computer scientists. We expect intense discussions, as the intent of the vision statement is not to impose a particular solution. To function properly, however, the group needs to share a vision, agree on common goals, define initial milestones, and accept principles to be observed in reaching for these goals. The present document is an attempt to define the vision (section 2), the goals (section 3), the rules (section 4), the principles (section 5) and the one-year milestones (section 6). As a full understanding of the five major goals of section 3 requires more detail, an appendix is specifically devoted to each of them.

These long-term goals and the one-year milestones of section 6 are indispensable complements to each other: achieving fundamental change, the goal of Semat, will take several years; but creating momentum requires reaching visible initial results promptly.

2 The vision

The Semat vision is twofold:

- Achieve all the goals, as described below and in the appendices.
- Create a platform (the kernel) allowing people to describe their current and future practices, patterns and methods so that they can be composed, simulated, applied, compared, evaluated, measured, taught and researched.

3 The kernel

The Call for Action mandates us to agree on "a kernel of widely-agreed elements, extensible for specific uses". To be effective the kernel must be kept concrete, focused and small.

The scope of the kernel is limited to the elements described in this section. In particular, the kernel is not an attempt at a unified methodology.

The focus of the kernel effort is to identify and describe the elements that are essential to all software engineering efforts.

Typical examples of the kernel's coverage are teamwork, project management and process improvement. The kernel will also integrate concepts from other engineering disciplines.

The kernel must accommodate change. The intention in the first twelve months is to

identify a kernel that both captures all interesting *methods*, *practices* and *patterns* (Fig. 1) used in software production today and can be adapted for future evolutions of the discipline.

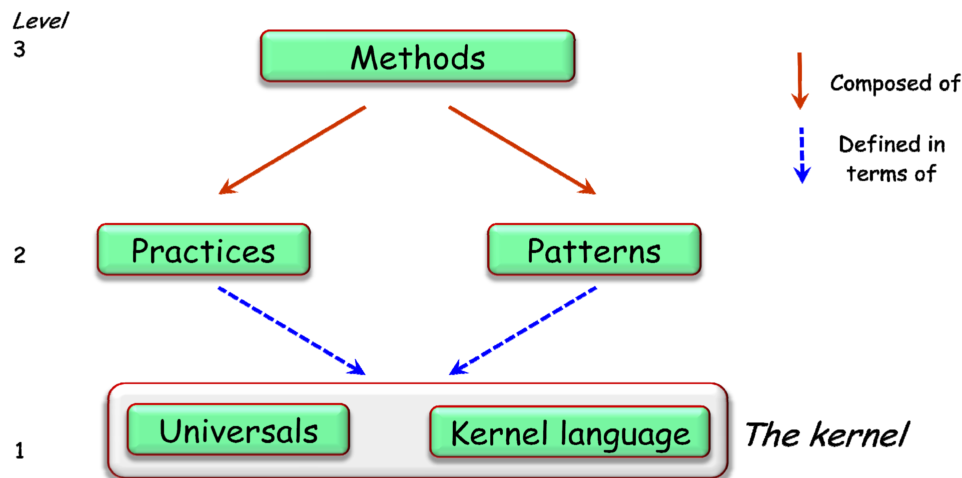

Figure 1: The Semat Diamond

The kernel must include a concrete representation of the acts and artifacts of software development, applicable to a wide range of software projects. It must also provide an extension language for adaptation to specific methods, practices and patterns.

To satisfy these needs, the kernel involves three major kinds of element:

- The kernel: universals and kernel language (level 1 in fig. 1).
- Practices and patterns (level 2), defined in terms of the kernel.
- Methods, defined as combinations of practices and patterns (level 3).

The term "*pattern*" deserves a specific definition. It is used in the present document to denote a general mode of operation that applies across practices. For example, many practices involve organizing workshops, or relying on self-organizing teams. By describing such concepts as patterns, we avoid repeating their details in the description of specific practices.

4 The goals

The work will proceed along five different tracks:

1. **Definitions**: defining software engineering and the other essential concepts of the discipline.

2. **Theory**: identifying the theories (in particular from mathematics) that provide essential help.
3. **Universals**: identifying the universal elements of software engineering, which must be integrated into the Semat kernel.
4. **Kernel language**: defining the language for describing the universals, practices and patterns.
5. **Assessment**: techniques for evaluating software engineering practices and theories, including the results of Semat.

All tracks should produce some visible results within twelve months, but the extent of these results may vary widely. Most in need of short-term results are track 3 and 4, as we must identify kernel elements to serve as the basis for the entire work, help the software industry reduce the jumble of methods, and improve the teaching of software engineering.

The appendices propose some starting ideas for each of the tracks.

5 The principles

While the precise modus operandi of the Semat effort will be determined at the beginning of the effort, a number of general principles are essential to its success.

Principles 1 to 9 apply to the end result of Semat, referred to as "the kernel". Principles 10 to 13 apply to the process for achieving this result.

1. **Quality**. The principal goal shall be the improvement of software products and processes.
2. **Simplicity**. The kernel shall only include essential concepts.
3. **Theory**. The kernel shall rest on a solid, rigorous theoretical basis.
4. **Realism** and **scalability**. The kernel shall be applicable by practical projects, including large projects, and based where possible on proven techniques.
5. **Justification**. Every recommendation shall be justified by a clear rationale.
6. **Falsifiability**. Every claim shall be subject to experimental evaluation and refutation.
7. **Forward-looking perspective**. While taking into account the methodological choices of the previous generation, the kernel shall not be bound to total

compatibility.

8. **Modularity**. Practices and patterns are defined by using the kernel, and they can be combined and adapted to meet the needs of individual organizations.
9. **Self-improvement**. The kernel shall be accompanied by mechanisms enabling its own evolution.
10. **Openness**. In the development of the kernel, every suggestion provided in an appropriate form by members of the Semat effort shall be considered.
11. **Fairness**. All ideas contributed shall be evaluated on merit. No aspect shall be designed to favor the interests of particular stakeholders or communities.
12. **Objectivity**. Ideas shall be evaluated on the basis of objective criteria, clearly defined in advance.
13. **Timeliness.** Deadlines shall be set and observed to allow the effort to progress and deliver results.

6 One-year milestones

The results expected one year after the start of the project are the following. Each corresponds to one of the five directions identified under "goals" and represents initial, objectively assessable progress towards the corresponding goal.

6.1 A set of definitions including a definition of the term software engineering and definitions of the fundamental concepts needed by practices and patterns.

6.2 The identification of specific theories or theoretical areas that hold potential for Semat, backed by examples of their successful application to specific software engineering practices.

6.3 A set of important universals, and a demonstration that they can be validated against a few specific practices used by important methods, including at least one not used in the development of the universals.

6.4 The definition of a kernel language and its successful use to describe the universals of 6.3 and their composition as applied to the methods selected in 6.3.

6.5 A set of metrics, sufficient to assess software practices, products and people, and backed by evidence of successful application to some projects.

Appendices

The five appendices detail the five goals of the Vision Statement: Definitions, Theory, Universals, Kernel language and Assessment.

Appendix 1: Definitions

Many technical debates involve terminology as much as actual disagreement on the substance. The Definitions track is intended to achieve agreement on definitions required early in the process, and tracking and categorizing definitions that arise from the other tracks over time.

A1.1 What is Software Engineering?

A good starting point, and a representative example, will be the definition of software engineering itself. The obvious definition ("software engineering is the application of engineering methods and discipline to the field of software") is insufficient to some people, either because they feel "engineering" in general is not defined well enough, or because they miss a description of how the application to software affects the basic definition. At the other extreme, Wikipedia has a long definition, taken from SWEBOK, the Software Engineering Body of Knowledge ("*Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software*"). This definition is disappointing: the list of adjectives is redundant ("*disciplined*" adds little to "*systematic*"); the list of activities is arbitrary (why "*operation*" but not, for example, documentation?); and the final "*that is*..." puts in question the usefulness of everything that precedes it.

Much discussion has already taken place on the Semat blog on the topic of defining software engineering. A summary of the positions:

- Software engineering is a discipline that requires formal study, experience, respect, creativity and discipline (see the tongue-in-cheek piece at http://parijatmishra.wordpress.com/2010/01/08/188/)
- The SWEBOK definition as quoted above, used by Wikipedia.
- The definition from the American Engineers' Council for Professional Development, which defines *engineering* as the creative application of scientific principles to design or develop structures, machines, apparatus, or manufacturing processes, or works utilizing them singly or in combination; or to construct or operate the same with full cognizance of their design; or to forecast their behavior under specific operating conditions; all as respects an intended

function, economics of operation and safety to life and property (see http://www.sciencedaily.com/articles/e/engineering.htm, and clearly we could substitute “software works” for “works” to define *software engineering*).
- One signatory criticized the above definitions, on the grounds that software engineering *should* comprise of craft, cooperative gaming, lean processes and design as knowledge acquisition (see http://alistair.cockburn.us/The+end+of+software+engineering+and+the+start+of+economic-cooperative+gaming).

Perhaps these definitions are not that far apart; all these views share the general idea of applying scientific knowledge and principles to software. It is among the goals of the Semat initiative to bring together many experts with different training, expertise and experience to agree on a common core.

A1.2 Beyond the basic definition

Together with a few other fundamental concepts, the definition of software engineering, or a first version, is part of the twelve-month initial goals (section 6).

These terms are just a starting point, and it is not realistic to require the rest of the Semat work, in particular the definition of the kernel, to wait for the adoption of final definitions for all the concepts involved in the Semat components: methods, practices, patterns, universals and the kernel language. The definition process will be continuous and iterative, extending and refining the Semat repository of definitions.

Appendix 2: Theory

One of the definitions of engineering is that it is the construction of artifacts based on scientific principles, and ultimately on mathematics. Software engineering should be no exception. A strong mathematical basis is essential to the Semat effort.

A considerable theoretical base is available for software engineering. Some of it simply consists of existing mathematical theories, applied largely unchanged; for example many areas of software engineering make good use of probability, statistics and queuing theories (and prominent researchers have repeatedly stated that we should use these theories much more widely). Another example is category theory, which served as an inspiration for abstract data types and object-oriented programming (although few O-O developers know it). In other cases, software applications prompted the development of new theories or major development to existing ones; examples include logic, where much of the impetus in the past decades has come from the need of programming languages, automata theory, and new developments such as model checking and abstract interpretation.

Software engineering is not just about technology, but also about organization, management, communication, collaboration and other human-oriented facets of software where insights may come from such disciplines as economics, sociology and psychology. Software engineering needs theories from these areas as well, to support the development of methods, practices, patterns and universals.

The chasm between practitioners and academics is more pronounced in software than in most other engineering disciplines. Whereas it is hard to imagine an electrical engineer stating that Maxwell's equations are a pure academic pursuit irrelevant for practitioners, such comments are often heard from software engineers and managers about theoretical contributions (such as formal verification techniques) that are just as important to their field. Even more often, practitioners have not even heard about such techniques.

Academics are not entirely blameless: some research has concentrated on problems that pose scientific challenges but are hard to relate to the concerns of industry.

The interaction between the two camps has been improving in recent years. Formal methods, for example, are increasingly used (often in disguised form) in many areas of information technology. It is one of the goals of Semat to make sure that software

engineering grows up to the level of other disciplines and, without entirely removing the chasm (a goal that is not desirable, as research and application do not have identical roles), establishes a healthy relationship between the theory and practice of software.

The mission of the Theory working group is to further this goal. It includes in particular, in the first step, the following tasks:

1. Identifying the areas of software engineering that are in direst need of theory (and are not already based on sound theory in the profession's dominant modes of operation).
2. Among theories that have already been applied to software but only by a minority of projects, identify those which show the highest potential of widespread beneficial application.
3. Among the areas identified in task 1, identify those for which no theory appears available as yet.
4. Identify the components of the Semat effort that will require specific theoretical support. In particular, define the kind of theory required to provide the kernel language (appendix 4) with a sound theory.

These four goals appear achievable within the one-year period set for the initial milestones of the Semat project.

The second step involves:

5. Developing a Roster Of Applicable Software Engineering Theories (ROAST) providing a list of useful existing theories (as identified under point 2 above) and, for each of them, a detailed guide to its application to actual projects.
6. Developing the theory for the kernel language (see point 4).
7. Picking a small number (most likely no more than three, and possibly as small as one) of theories that need to be developed, and setting the conditions for their development as well as, if possible, basic elements of these theories.

Appendix 3: Universals[1]

To keep the kernel concrete, focused and small requires identifying the truly universal elements of software engineering. While there seems to be general agreement on the existence of such elements, essential to all software engineering efforts, there seems to remain much confusion about what they are and how we assess that they are truly essential (a condition for being included in the kernel). The role of the Universals task is to identify and define these elements.

A3.1 Kernel properties

Here are a few candidate features we consider integral to any successful kernel:

- *Concise.* The kernel must only focus on the small set of elements that are truly essential.
- *Scalable.* The kernel's scope must extend from the smallest projects to large systems and systems-of-systems.
- *Extensible.* The kernel must provide the ability to add practices, patterns, levels of detail and lifecycle models. It must support tailoring to specific domains of application and to projects involving more than software.
- *Measurable.* The kernel must provide mechanisms to assess quantitatively all the relevant artifacts of software processes and products.
- *Formally specified.* The kernel must be mathematically defined. Such a definition should be developed along with the kernel itself, not postponed to hypothetical later efforts. This goal should be pursued in cooperation with the Theory track.
- *Broad practice coverage.* The kernel must support many different practices as long as they are recognized as beneficial for significant segments of the industry.
- *Broad lifecycle coverage.* The kernel must accommodate various lifecycle models recognized as beneficial for significant segments of the industry.
- *Broad technology coverage.* The kernel must be adaptable to a wide range of software technologies (programming languages, specification languages, graphical notations, software tools) as long as they are recognized as beneficial for significant segments of the industry.

---

[1] Written with the collaboration of Ian Spence

A3.2 Role

The kernel is the crucible of the Semat initiative because it determines the applicability of all other Semat efforts to the practical work of software engineers. It will provide a concrete framework that they can use on every project, allowing them to identify and apply the practices that they need to be successful in their particular context.

A3.3 Criteria for inclusion

The basic rules characterizing elements that deserve to be include the following:

- *Universal*: potentially applicable to all software engineering efforts.
- *Significant*: have the ability to contribute in a positive and perceptible way to the quality of software processes or products (or both).
- *Relevant*: available for application by all software engineers, regardless of background, and methodological camp (if any).
- *Defined* precisely.
- *Actionable*: not just words and concepts, but precise guidelines that projects can apply.
- *Assessable*: suitable for quantitative evaluation of their application.
- *Comprehensive*. This criterion applies to the collection of the kernel elements; together, they must capture the essence of software engineering, providing a map that supports the crucial practices, patterns and methods of software engineering teams.

A3.4 Examples and questions

In the various blogs and discussions on the kernel that appear on the Semat Web site, a number of candidate universals have been identified. Here is a brief discussion of a few examples.

Some contributors have suggested that the one true universal of any software engineering endeavor is working programs. What exactly does this mean: programs that pass the defined set of tests? Programs that meet the defined requirements? Programs that address the customer's real needs?

What about requirements? Are they universals, or just one of the practices used to capture the customer's intent?

These are the sorts of questions this working group will have to answer to create an initial kernel that can be used to examine some of our most popular practices and challenge our definition of software engineering.

Other contributors have suggested longer lists of universals including such elements as:

- Project
- Team
- User Experience
- System
- Quality
- Intent

Are all of these really universals? Are they all really essential to software engineering? Are they at the same conceptual level ("the same types of things")? Do they relate to each other and if so how?

The goal of the Kernel task is to produce a concrete model of software engineering answering all these questions, and providing a foundation for the definition, evaluation and improvement of the practices, patterns and methods that will allow us to professionalize the software engineering community.

Appendix 4: Kernel Language

As the reflection on the kernel language is at an early stage, this appendix does not define final requirements for the language, but gives an idea of what needs it will address.

Developing the kernel language will require two kinds of contributors, with distinct expertise:

- Some will bring extensive experience with method elements (a term used in this appendix to denote methods, patterns and practices), to ensure that the kernel language can cover users' needs.
- Others will bring expertise in language design.

The two sections of this appendix correspondingly address the kernel language's usage and its design.

A4.1 Kernel language usage

The role of the kernel language is to describe practices and patterns, and their composition into methods. The word "description" as used in this section denotes such a description of a method element (in the above sense), expressed in the kernel language.

The principal intended users of the kernel language are projects looking for appropriate method elements and applying method elements they have chosen.

The kernel language and the resulting descriptions must satisfy the following properties to deliver the expected value to these users:

- The language can cover all relevant practices and patterns, and their composition, in *today's* methods.
- It supports *composing* them in different ways to describe new method elements.
- It is *extendible*, allowing the description of yet-to-be-invented method elements and their elements (such as individual practices).
- The descriptions are *easy to understand*; the language should be designed for the developer community (not just process engineers and academics).
- The language supports *comparing* method elements (a very hard task today).
- The descriptions are built in terms of the *universals* identified in the Semat effort

(see appendix A.3), helping one of the principal goals of Semat: to avoid reinventing practices, patterns and methods.

- The language and resulting description support *simulating* the application of method elements.
- They support *applying* these method elements in real projects.
- They provide *validation* mechanisms, so that it is possible to assess whether a project that claims to apply a given method element (as described through the kernel language) actually does, and is not just paying lip service to it. This problem is sometimes referred to as "closing the gap" (between what project teams say they do and what they actually do).

The last requirement has implications on team organization: every team should provide appropriate resources to assess the project's actual performance against the chosen method elements, as described in the kernel language.

A4.2 Design of the kernel language

Although no preliminary design exists for the kernel language, the following requirements on its design have been identified.

As any other precisely defined language, the kernel language shall have an abstract syntax, validity rules ("static semantics") and a semantics. It shall have at least one concrete syntax, but may have several, for example a textual form and a graphical form.

The kernel language shall support four principal applications:

- *Describing* universals: practices and patterns, which are the building blocks, and the composition mechanisms for building methods out of these elements.
- *Simulating* the application to a project of the resulting descriptions.
- *Applying* the descriptions for real, "closing the gap".
- *Assessing* a simulation or application against chosen method elements.

The concept of *state* is likely to play an important role in the kernel language, to represent work progress. For example, describing a practice that involves iterative development requires describing the starting and ending states of every iteration.

The kernel language shall be *extensible*; in particular, it shall allow addition or modifications of practices, patterns and possibly composition techniques.

The following rules apply to practices:

- A practice is a separate concern of a method. Examples are "development from requirements to test", "iterative development", "component-based development".
- Every practice shall be described on its own, separately from any other practice.
- Every practice, unless explicitly defined as a continuous activity, has a clear beginning and an end.
- Every practice brings defined value to its stakeholders.
- Every practice must be assessable; its description must include criteria for its assessment.
- Whenever applicable, the assessment criteria must include quantitative elements; correspondingly, the description must include suitable metrics.
- Composition mechanisms for practices include merging and extension.

The following rules apply to patterns:

- Every pattern (per the definition in section 3) shall be applicable to several different practices.
- Every pattern shall be described on its own, separately from any other pattern and any practice to which it applies.

Appendix 5: Assessment[2]

Establishing a firm foundation for software engineering requires a systematic way to assess practices, patterns, methods (together called "method elements" in this appendix) and the supporting tools on the basis of objective criteria. As in other engineering disciplines, these criteria must be quantitative; they must apply sound theory (see appendix 2); and they must draw on statistically sound data.

The fundamental questions, addressed in the three sections of the present appendix, are:

- What are the objectives of the assessed method elements?
- What are the criteria for judging their quality?
- What should the measurement process be?

A5.1 Assessment objectives

Once we have identified the kernel, in particular the relevant method elements (in the above sense), we must give users objective criteria for assessing whether these elements help them with their tasks. This observation raises two questions: who are the users, and what are their intended tasks?

The *user* categories of most direct relevance are:

- Software practitioners (engineers and managers), who are pragmatic and will use whatever method elements they find to be available and easy to use.
- Their customers, who are generally concerned about getting quality software, fast, at low cost, and in a predictable way.
- Academics, who teach or research software engineering.

All these user categories care about all the traditional process and product qualities, including functionality, availability, cost-effectiveness, support, usability, reliability, efficiency, extendibility and reusability.

The *tasks* that will be subject to assessment include all important tasks of software engineering: both technical tasks (from analysis to design, implementation, documentation, verification and validation, deployment, operation, and others) and human-centered tasks such as management, training, support and communication.

---

[2] Written with the collaboration of Watts Humphrey

A5.2 Assessing quality

The assessment method must be applicable to any practice and consider all its important properties:

- *Functionality*: Does the description of the method element make it possible to measure coverage of the intended functionality?
- *Usability*: How is usability measured? Possible techniques involve a usability laboratory if available, otherwise user surveys. Such surveys should be precisely defined and repeatable so that a variety of user situations can be measured and statistically useful and objective data obtained from multiple sources.
- *Reliability*: Does the method element always produce the intended results or are project variables involved and, if so, what are these variables, their likelihoods, and their impacts?
- *Efficiency*: This productivity measure requires data on the time spent by one or more software engineers in applying the method element and the volume of work products produced.
- *Extensibility*: This is a system design principle where the implementation takes into consideration future growth. It is a systemic measure of the ability to extend a system and the level of effort required to implement the extension. Extensions can be through the addition of new functionality or through modification of existing functionality. Does the method element support the central theme to provide for change at minimal impact to existing system functions?
- *Reusability*: Quantitative criteria here are of four types: reuse cost-benefits models include economic cost-benefit analysis as well as quality and productivity payoff; amount of reuse metrics serve to assess and monitor a reuse improvement effort by tracking percentages of reuse for life cycle objects; reusability metrics indicate the likelihood that an artifact is reusable; reuse library metrics are used to manage and track usage of a reuse repository.

A5.3 Assessment process

Defining an assessment process involves three aspects: how to choose the projects for evaluating method elements; what properties to measure; and organizing the process. The following subsections examine these issues in turn.

A5.3.1 Choosing sample projects

To assess practices and other method elements, one can choose either an industrial software project or controlled tests in an experimental environment, for example with students. Neither technique is entirely satisfactory.

Industry projects have two major advantages: size (hard to match in an academic context) and realism (since they are intended for actual usage). They also, however, face two principal limitations. First, they typically do not support reproducibility: the very size and economic impact of an industry project implies that it will generally be a one-off effort, yielding a set of isolated data points with questionable statistical value. This problem can be alleviated by collecting assessment data from many projects in a consistent context, for example projects from a single company, all in the same general area. The other limitation is that in an industry project many parameters are set by the context and beyond the control of the assessment procedure:

- New product vs modification of an existing product.
- Self-contained vs part of larger system.
- Internal vs commercial use.
- Custom development vs mass product.
- Team expertise, experience, and training for both the application domain and the method elements being assessed.
- Cost model: fixed-price, cost plus or other.
- Impact of the assessment procedure itself ("Heisenberg" effect).

Since controlled experiments are difficult to fund in an industrial context, they usually take place in academia, using students. The advantage is that the experimenter has much finer control on the parameters listed above and, most importantly, that it is possible to achieve some degree of reproducibility and statistical significance. The disadvantage is that in general the experiments will only apply the assessed method elements in the small, raising the issue of generalization to realistic situations (scalability), and that students are not necessarily representative of professional software engineers.

Since each technique — assessment in actual projects and in an academic environment — has its advantages and limitations, a fully convincing assessment should involve both: controlled experiments by researchers, to obtain basic insights; and validation through some industrial project, to confirm that these insights scale up to actual industry usage.

A5.3.2 What to measure

The assessment procedure requires a precise definition of what needs to be measured: critical project and developer variables, activities, time spent for each activity, defects found, work products produced.

A5.3.3 Establishing an assessment system

To establish an effective and useful assessment system requires obtaining the support of the two principal communities involved:

- We must convince the software engineering community and its practitioners of the value of objective and quantitative evaluations of software engineering method elements and artifacts (practices, patterns, methods, tools).
- Because of the importance of academic studies (5.3.1), we must convince the academic community of the value of this task and help academics teach to their students the techniques of rigorous data collection. We must also make sure that the resulting work, which falls in the general area known as "experimental computer science", is given proper academic recognition according to the criteria that matter to academics, in particular publication. (As an example, the natural sciences have a tradition of accepting reproducibility experiments, where a researcher confirms or invalidates a previously published result, as worthy of publication, and even potentially prestigious. For the most part computer science lacks such a tradition.)