CHAPTER FIVE

Learning Through Application: The Maturing of the QIP in the SEL

Victor R. Basili

Empirical studies—formal research that uses respected, validated methods for establishing the truth of an assertion—have started to make headway within software engineering. The good news is that these studies have finally become recognized as an important component of the discipline. One sees more and more empirical studies and experiments in the literature to confirm or reject the effectiveness of some method, technique, or tool.

The bad news is that these studies are not yet used for discovery. The experiment is an add-on, tried after the concept is considered complete. The scientific method, however, is classically based on applying a method, technique, or tool and learning from the results how to evolve the concept. This is how theories are tested and evolved over time. In the software engineering discipline, where the theories and models are still in the formative stages and processes are applied by humans as part of a creative process, observing the application or performing exploratory studies should be an important step in the evolution of the discipline.

What Makes Software Engineering Uniquely Hard to Research

Software engineering has several characteristics that that distinguish it from other disciplines. Software is *developed* in the creative, intellectual sense, rather than *produced* in the manufacturing sense. Software processes are development processes, not production processes. In other words, they are not replicated over and over again. This unique aspect of the discipline

is probably the most important one, and greatly affects how we learn. We always need to be on the lookout for the effect of context variables. Because software engineering is a human-based discipline, there will always be variation in study results and we will never be able to control or even identify all the context variables. The discipline creates a need for continual experimentation, as we explore how to modify and tailor processes for people.

The variation in context springs from more than just the people who create the software; each piece of software itself is different, as are software development environments. One consequence of this is that process is a variable, goals are variables, etc. That is, we need to select the right processes for the right goals for the environment we are analyzing. So, before we decide how to study a technique, we need to know something about the environment and the characteristics of what we are about to build.

A second distinguishing characteristic of the software engineering discipline is software's *intangibility*, or one might say the invisibility of its structure, components, and forms of development. This is compounded by a third characteristic, the field's *immaturity*, in the sense that we haven't developed sufficient models that allow us to reason about processes, products, and their relationships. These difficulties intensify the need to learn from the application of ideas in different situations and the requirement to abstract from what we see.

A final problem is that developing models of our experiences for future use (that is, reuse) requires additional resources in the form of money, organizational support, processes, people, etc. Building models, taking measurements, experimenting to find the most effective technologies, and feeding back information for corporate learning cost both time and money. These activities are not a byproduct of software development. If these activities are not explicitly supported, independent of the product development, they will not occur and we will not make quality improvements in the development process.

All this makes good experimentation difficult and expensive. Experiments can be confirmatory only in the small scale and are subject to problems in understanding scale-up, the integration of one process with another, the understanding of the effect of context variables, etc.

A Realistic Approach to Empirical Research

I believe we need to focus more attention on informal exploratory studies that provide insights into directions software development can take. We should couple these, when appropriate, with more formal empirical studies to test out pieces of the whole that can be added to the tapestry that helps make the discipline clear. I believe that the study of the software engineering discipline is exploratory and evolutionary. It follows the scientific method, but because of its nature, real experiments are not always possible or useful.

I like to say that the study of software engineering is a laboratory science, and it is a big science. So the laboratory is quite grand and we need methods that support the exploratory nature of this big science. The discipline cannot be understood only by analysis. We need to learn from

applying the discipline whether relationships hold, how they vary, and what the limits of various technologies are, so we can know how to configure process to develop software better.

We need to take advantage of all opportunities we can find to explore various ideas in practice—e.g., test their feasibility, find out whether humans can apply them, understand what skills are required to apply them, and test their interactions with other concepts. Based upon that knowledge, we need to refine and tailor each idea to the application environment in which we are studying it so it can be transferred easily into practice. Even before we build models, we need to try out our ideas in practice and evolve them. We are, in short, an exploratory science: we are more dependent on empirical application of methods and techniques than many disciplines.

Variety is an important aspect of applying the scientific method to software engineering. We need many applications of a process, taking place in different environments, each application providing a better understanding of the concepts and their interaction. Over time, all context variables need to be considered. Many of them will not even pop up until we have seen applications of the approach in practice by different people at different sites.

The building of the tapestry is too grand for any one group to perform. This is big science and it requires the collaboration of many groups. Results need to be shared in an effective way, a repository of evolving models and lessons learned that can be added to and used by researchers.

The NASA Software Engineering Laboratory: A Vibrant Testbed for Empirical Research

To support the argument for informal learning I make in this chapter, I'll summarize our experiences in the NASA Software Engineering Laboratory (SEL) over a period of 25 years, where we learned a great deal not just through experiments, but also by trying to understand the problems, applying potential solutions, and learning where they fell short. We did run controlled experiments and performed case studies, but they were done in the context of the larger evolutionary learning process.

The SEL was established in 1976 with the goals of understanding ground-support software development for satellites, and where possible, improving the process and product quality in the Flight Dynamics Division at Goddard using observation, experimentation, learning, and model building [Basili and Zelkowitz 1977]. The laboratory had a team that was supportive of learning, consisting of developers from both NASA and Computer Sciences Corporation (CSC), along with a research group at the University of Maryland (UMD). The three groups formed a consortium. The organization made a decision to support our empirical research and integrate it into the overall activities of the organization. Support came from the project budget, rather than the research budget at NASA.

In 1976, very few empirical studies were being performed in software engineering. The idea of creating a laboratory environment to study software development was perhaps unprecedented. But it provided an excellent learning environment where potential solutions to problems were proposed, applied, examined for their effectiveness and lessons, and evolved into potentially better solutions. Characteristics that made this setup a good place for empirical research included the limited domain of the application, the use of professional developers, firm support from the local organization, the presence of a research team to interact closely with the practical developers, and a mix of developers and managers with different goals, personalities, and responsibilities. The balance created an ideal learning environment with lots of feedback and collaboration.

From 1976 to 2001, we learned a great deal while making a lot of mistakes. Examples of these mistakes include:

- Trying to make assessments before fully understanding the environment
- Being data-driven rather than goal- and model-driven
- Drawing on other people's models derived from other environments to explain our own environment

The learning process was more evolutionary than revolutionary. With each learning experience, we tried to package what we had learned into our models of the processes, products and organizational structure.

The SEL used the University researchers to test high-risk ideas. We built models and tested hypotheses. We developed technologies, methods, and theories as needed to solve a problem, learned what worked and didn't work, applied ideas that we read about or developed on our own when applicable, and all along kept the business going.

The most important thing we learned was how to apply the scientific method to the software domain—i.e., how to evolve the process of software development in a particular environment by learning from informal feedback from the application of the concepts, case studies, and controlled experiments. The informal feedback created the opportunity to understand where to focus our case studies and experiments. Informal feedback also, and perhaps surprisingly, provided the major insights.

What follows in this chapter is a retrospective look at our attempts to instantiate the scientific method and how our approach evolved over time based upon feedback concerning our application of the ideas.

The Quality Improvement Paradigm

I will begin with a distillation of our journey, a process for applying the scientific method to software engineering in an industrial environment. We called our version of the scientific method the Quality Improvement Paradigm (QIP) [Basili 1985], [Basili and Green 1994]. It consists of six basic steps:

1. *Characterize* the current project and its environment with respect to the appropriate models and metrics. (*What does our world look like?*)
2. *Set quantifiable goals* for successful project performance and improvement. (*What do we want to know about our world and what do we want to accomplish?*)
3. *Choose the process model* and supporting methods and tools for this project. (*What processes might work for these goals in this environment?*)
4. *Execute* the processes, construct the products, and collect, validate, and analyze the data to provide real-time feedback for corrective action. (*What happens during the application of the selected processes?*)
5. *Analyze* the data to evaluate the current practices, determine problems, record findings, and make recommendations for future project improvements. (*How well did the proposed solutions work, what was missing, and how should we fix it?*)
6. *Package* the experience in the form of updated and refined models and other forms of structured knowledge gained from this and prior projects, and save it in an experience base to be reused on future projects. (*How do we integrate what we learned into the organization?*)

The Quality Improvement Paradigm is a double-loop process, as shown by Figure 5-1. Research interacts with practice, represented by project learning and corporate learning based upon feedback from application of the ideas.

But that is not where we started. Each of *these* steps evolved over time as we learned from observation of the application of the ideas. In what follows, I will discuss that evolution and the insights it provided, with formal experiments playing only a support role for the main ideas.

The learning covered a period of 25 years, although here I concentrate on what we learned mostly in the first 20. The discussion is organized around the six steps of the QIP, although there is overlap in the role of many of the steps. In each case I say what we learned about applying the scientific method and what we learned about improving software development in the Flight Dynamics Division of NASA/GSFC.

We started in 1976 with the following activities, representing each step of the approach: characterize, set goals, select process, execute process, analyze, and package.

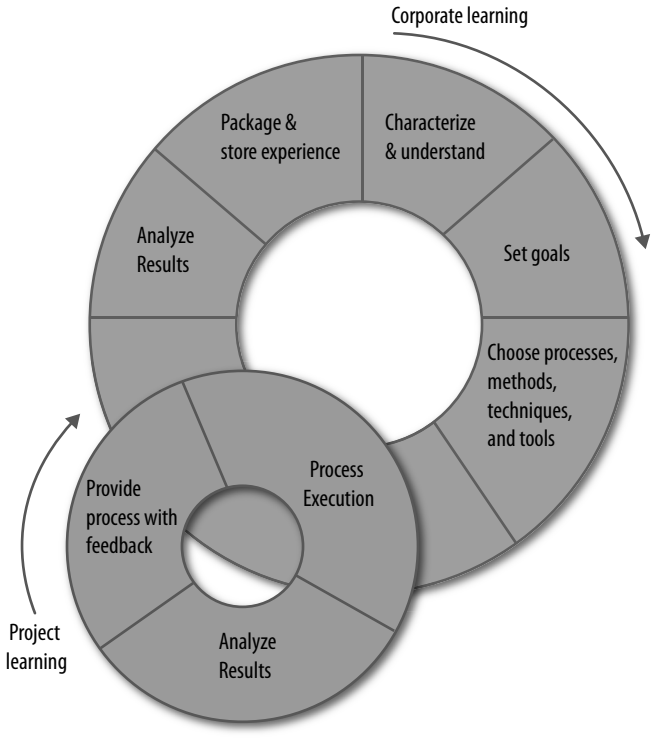

FIGURE 5-1. Quality Improvement Paradigm

Characterize

In the beginning, we looked at various models in the literature and tried to apply them to help us understand our environment (e.g., Raleigh curve or MTTF models). We found they were not necessarily appropriate. Either they were defined for larger systems than what we were building (on the order of 100KSLOC) or they were applied at different points in time than what we needed. This led to the insight that we needed to build our own environment-appropriate models using our own data. We needed to better understand and characterize our own environment, projects, processes, products, etc., because we could not use other people's models that were derived for different environments [Basili and Zelkowitz 1978], [Basili and Freburger 1981], [Basili and Beane 1981].

We needed to understand our own problem areas. So over time, we began to build baselines to help us understand the local environment; see Figure 5-2. Each box represents the ground support software for a particular satellite. We built baselines of cost, defects, percent reuse, classes of defects, effort distribution, source code growth in the library, etc. We used these baselines to help define goals, and as historical data to establish the basis for showing improvement.

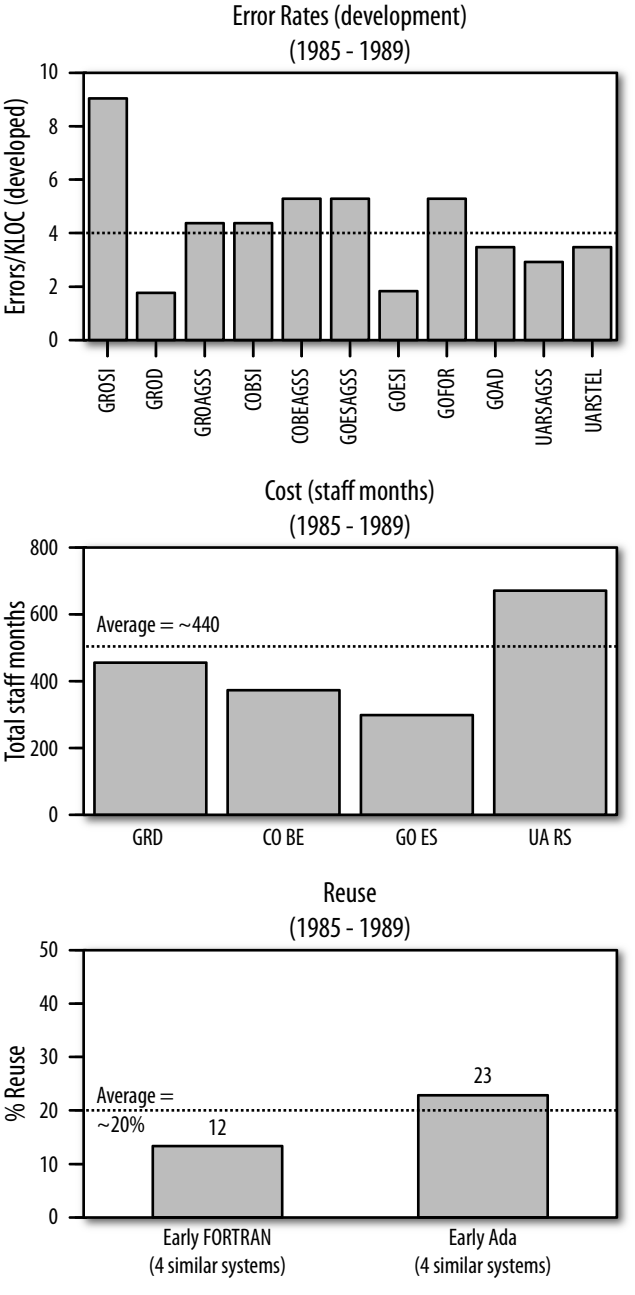

FIGURE 5-2. Example NASA baselines

As we progressed over time, we learned that we needed to better understand the factors that created similarities and differences among projects so we would know the appropriate model to apply and what variables were influencing the effectiveness of the processes. Context, even within the local environment, was important.

Set Goals

From the very beginning, we decided to use measurement as the abstraction process to provide visibility into what was occurring, and developed data collection forms and a measurement tool. We collected data from half a dozen projects in a simple database and tried to interpret that data, discovering that sometimes we did not have the right data or sufficient information to answer our questions. We realized that we couldn't just collect data and then figure out what to do with it; data collection needed to be goal-driven. This led to the development of the Goal Question Metric (GQM) approach to help us organize the data around a particular study [Basili and Weiss 1984]. You can also drown in too much data, especially if you don't have goals. We have continued to evolve the GQM, for example, by defining goal templates [Basili and Rombach 1988].

We also understood the important role that nominal and ordinal data played in capturing information that was hard to measure in other ways. We moved from a database to a model-based experience base as our models evolved, based upon our experiences with over 100 projects in that environment.

Select Process

We began by minimally impacting the processes, using heuristically defined combinations of existing processes. We began to run controlled experiments at the university, with students, to isolate the effects of small sets of variables at minimal cost [Basili and Reiter 1981]. When we understood the effects of those processes, we began to experiment with well-defined technologies and high-impact technology sets. Many of these technologies were studied first at the university in controlled experiments before being brought to the SEL for use on live projects, e.g., Code Reading by stepwise abstraction [Basili and Selby 1987], Cleanroom [Selby et al. 1987], and Ada and Object oriented design [Basili et al. 1997]. The university studies diminished the risks of applying these techniques in the SEL. Over time we began to understand how to combine controlled experiments and case studies to provide a more formal analysis of isolated activities [Basili 1997].

We began to experiment with new technologies to learn more about the relationships between the application of processes and the resulting product characteristics. But the motivation for choosing these techniques was based upon the insights we gained from observing the problems that arose in the SEL and were aimed at specific goals, e.g., minimizing defects. Based upon recognition of problems with requirements, for example, we developed a set of reading techniques for identifying defects in requirements documents [Basili et al. 1996].

We recognized the obvious fact that we needed to understand how to choose the right processes to create the desired product characteristics and that some form of evaluation and feedback were necessary for project control. Reusing experience in the form of processes, products, and other forms of knowledge is essential for improvement. We learned to tailor and evolve technologies based upon experience.

Execute Process

When we started, data collection was an add-on activity; we expected the developers to perform their normal development processes and fill out the data forms we provided. The process was loosely monitored to see that the data forms were being filled out and the developers understood how to fill out the forms. The lack of consistent terminal use and support tools forced the data collection to be manual. Sharing the intermediate results with the developers allowed them to provide feedback, identify misunderstandings, and suggest better ways to collect data. Over time, using GQM, we collected less data, and embedded data collection into the development processes so the data were more accurate, required less overhead, and allowed us to evaluate process conformance. We captured the details of developer experiences via interaction between developers and experimenters, providing effective feedback about local needs and goals. We combined controlled experiments and case studies with our general feedback process to provide more formal analysis of specific methods and techniques.

Analyze

We began by building and analyzing the baselines to characterize the environment. Baselines were built of many variables, including where effort was spent, what kinds of faults were made, and even source code growth over time. These provided insights into the environment, showed us where to focus process improvement, and offered a better understanding of the commonality and differences among projects. We began to view the study of software development as following an experimental paradigm—i.e., design of experiments, evaluation, and feedback are necessary for learning. Our evolution of analysis methods went from correlations among the variables [Basili et al. 1983] to building regression models [Bailey and Basili 1983] and more sophisticated quantitative analyses [Briand et al. 1992], to including all forms of qualitative analysis [Seaman and Basili 1998]. Qualitative analysis played a major role in our learning process, as it allowed us to gain insights into the causes of effects. Little by little, we recognized the importance of simple application of the ideas followed by observation and feedback to evolve our understanding, which we incorporated into our models and guided where and when to use the more formal analytic approaches.

We realized it was impossible to run valid experiments on a large scale that covered all context variables. The insights gained from pre-experimental designs and quasi-experiments became critical, and we combined them with what we learned from the controlled experiments and case studies.

Package

Our understanding of the importance, complexity, and subtlety of packaging evolved slowly. In the beginning we recorded our baselines and models. Then, we recognized the need for focused, tailored packages—e.g., generic code components and techniques that could be tailored to the specific project. What we learned had to become usable in the environment, and we needed to constantly change the environment based on what we learned. Technology transfer involved a new organizational structure, experimentation, and evolutionary culture change. We built what we called *experience models*—i.e., models of behavior based upon observation and feedback in a particular organization. We built focused, tailorable models of processes, products, defects, and quality, and packaged our experiences with them in a variety of ways (e.g., equations, histograms, and parameterized process definitions). The hard part was integrating these packaged experiences. All this culminated in the development of the Experience Factory Organization [Basili 1989]; see Figure 5-3.

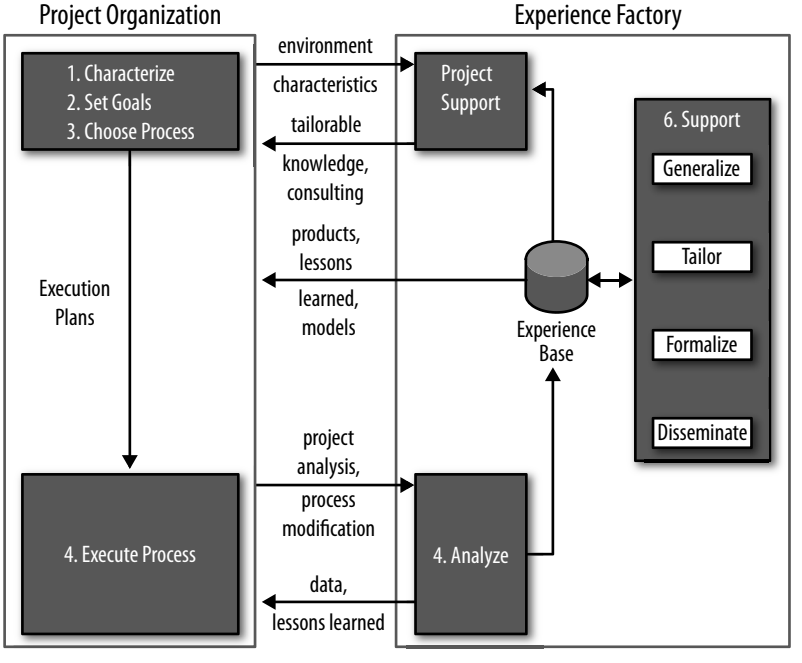

FIGURE 5-3. The Experience Factory Organization

The Experience Factory of processes, products, and other forms of knowledge is essential for improvement [Basili 1989]. It recognizes the need to separate the activities of the project development organization from the building of knowledge based upon experiences within the organization. Sample activities of the project organization are decomposing a problem into simpler ones, instantiation of the solution, design and implementation, validation, and verification. The goal is to deliver a product within cost and schedule. The activities of the Experience Factory are the unification of different solutions and redefinition of the problem, generalization, formalization, and integration of experiences. It does this by analyzing and synthesizing what it observes and experimenting to test out the ideas. Its goal is experience and delivering recommendations to the projects. It is responsible for the evaluation, tailoring, and packaging of experiences for reuse.

The Experience Factory cannot be built solely on the basis of small, validated experiments. It requires the use of insights and intelligent judgment based upon observation for the application of ideas, concepts, processes, etc.

Once we felt we understood how to learn from our observations and how to combine the results of case studies and controlled experiments (which was more publishable than our insights), we continued with our process of application, observation, and learning. We dealt with such topics as commercial off-the-shelf (COTS) development, reading techniques, etc.

Learning in an organization is time-consuming and sequential, so we need to provide projects with short-term results to keep development team interested. We need to find ways to speed up the learning process and feed interim results back into the project faster. If we are successful in evolving, our baselines and the environment are always changing, so we must continue to reanalyze the environment. We need to be conscious of the trade-off between making improvements and the reuse of experience.

Conclusion

This tells the story of over 25 years of evolution and tailoring the goals and processes for a particular environment. The continuous improvement was measured by taking values of three data points: the development defect rates, the reduced cost for development, and the improvement of reuse of code at three points in time: 1987, 1991, and 1995. Each data point represents the average of the three years around it [Basili et al. 1995]; see Table 5-1.

TABLE 5-1. The results of the QIP approach in the SEL

Continuous improvement in the SEL	1987–1991	1991–1995
Development defect rates	75%	37%
Reduced cost	55%	42%
Improved reuse	300%	8%

During this period there was a continual increase in the functionality of the systems being built. An independent study estimated it to be a five-fold increase from 1976 to 1992.

The cost of this activity was about 10% of the development costs. However, the data shows an order of magnitude improvement for that 10% cost, which is a pretty good ROI.

But the data in Table 5-1 are not the results of a controlled experiment or even a well-defined case study. One can argue that these results might have occurred anyway, even if we did nothing. There is no control group. But the people involved believe differently. We believe the learning process worked and delivered these results. No controlled experiment could have been developed for this 25-year study, as we did not know when we started what we would do and how it would evolve.

During the 25 years of the SEL, we learned a great deal about software improvement. Our learning process was continuous and evolutionary, like the evolution of the software development process itself. We packaged what we learned into our process, product, and organizational structure. The evolution was supported by the symbiotic relationship between research and practice. This relationship requires patience and understanding on both sides, but when nurtured, really pays dividends.

The insights gained from learning by application, supplemented where appropriate by pre-experimental designs, quasi-experiments, controlled experiments, and case studies, provided a wealth of practical and theoretical results that could not have been gathered from formal experiments alone. It was the large-scale learning that allowed for the development of such things as the GQM approach, the QIP, and the Experience Factory, as well as the development and evaluation of various techniques and methods.

References

[Bailey and Basili 1983] Bailey, J. and V. Basili. 1983. A Meta-Model for Software Development Resource Expenditures. *Proceedings of the Fifth International Conference on Software Engineering*: 107–116.

[Basili 1985] Basili, V. 1985. Quantitative Evaluation of Software Methodology. Keynote Address, *Proceedings of the First Pan Pacific Computer Conference* 1:379–398.

[Basili 1989] Basili, V. 1989. Software Development: A Paradigm for the Future. *Proceedings of COMPSAC '89*:471–485.

[Basili 1997] Basili, V. 1997. Evolving and Packaging Reading Technologies. *Journal of Systems and Software* 38(1):3–12.

[Basili et al. 1983] Basili, V., R. Selby, and T. Phillips. 1983. Metric Analysis and Data Validation Across FORTRAN Projects. *IEEE Transactions on Software Engineering* 9(6):652–663.

[Basili et al. 1995] Basili, V., M. Zelkowitz, F. McGarry, J. Page, S. Waligora, and R. Pajerski. 1995. Special Report: SEL's Software Process-Improvement Program. *IEEE Software* 12(6):83–87.

[Basili et al. 1996] Basili, V., S. Green, O. Laitenberger, F. Shull, S. Sorumgaard, and M. Zelkowitz. 1986. The Empirical Investigation of Perspective-Based Reading. *Empirical Software Engineering: An International Journal* 1(2):133–164.

[Basili et al. 1997] Basili, V., S. Condon, K. Emam, R. Hendrick, and W. Melo. 1997. Characterizing and Modeling the Cost of Rework in a Library of Reusable Software Components. *Proceedings of the Nineteenth International Conference on Software Engineering (ICSE)*: 282–291.

[Basili and Beane 1981] Basili, V. and J. Beane. 1981. Can the Parr Curve Help with the Manpower Distribution and Resource Estimation Problems? *Journal of Systems and Software* 2(1): 59–69.

[Basili and Freburger 1981] Basili, V. and K. Freburger. 1981. Programming Measurement and Estimation in the Software Engineering Laboratory. *Journal of Systems and Software* 2(1):47–57.

[Basili and Green 1994] Basili, V. and S. Green. 1994. Software Process Evolution at the SEL. *IEEE Software* 11(4):58–66.

[Basili and Hutchens 1983] Basili, V. and D. Hutchens. 1983. An Empirical Study of a Syntactic Complexity Family. *IEEE Transactions on Software Engineering* 9(6):664–672.

[Basili and Reiter 1981] Basili, V. and R. Reiter, Jr. 1981. A Controlled Experiment Quantitatively Comparing Software Development Approaches. *IEEE Transactions on Software Engineering* 7(3):299–320 (IEEE Computer Society Outstanding Paper Award).

[Basili and Rombach 1988] Basili, V. and H.D. Rombach. 1988. The TAME Project: Towards Improvement-Oriented Software Environments. *IEEE Transactions on Software Engineering* 14(6):758–773.

[Basili and Selby 1987] Basili, V. and R. Selby. 1987. Comparing the Effectiveness of Software Testing Strategies. *IEEE Transactions on Software Engineering* 13(12):1278–1296.

[Basili and Weiss 1984] Basili, V. and D. Weiss. 1984. A Methodology for Collecting Valid Software Engineering Data. *IEEE Transactions on Software Engineering* 10(3):728–738.

[Basili and Zelkowitz 1977] Basili, V. and M. Zelkowitz. 1977. The Software Engineering Laboratory: Objectives. *Proceedings of the Fifteenth Annual Conference on Computer Personnel Research*:256–269.

[Basili and Zelkowitz 1978] Basili, V. and M. Zelkowitz. 1978. Analyzing Medium-Scale Software Development. *Proceedings of the Third International Conference on Software Engineering*: 116–123.

[Briand et al. 1992] Briand, L., V. Basili, and W. Thomas. 1992. A Pattern Recognition Approach for Software Engineering Data Analysis. *IEEE Transactions on Software Engineering* 18(11):931–942.

[Seaman and Basili 1998] Seaman, C. and V. Basili. 1998. Communication and Organization: An Empirical Study of Discussion in Inspection Meetings. *IEEE Transactions on Software Engineering* 24(7):559–572.

[Selby et al. 1987] Selby, R., V. Basili, and T. Baker. 1987. Cleanroom Software Development: An Empirical Evaluation. *IEEE Transactions on Software Engineering* 13(9):1027–1037.