A Unifying Theoretical Foundation for Software Engineering

Dewayne E Perry
Center for Advanced Research in Software Engineering (ARiSE)
The University of Texas at Austin
perry@mail.utexas.edu

Abstract

The goal of this keynote paper is to argue for a unifying theoretical foundation for software engineering. I believe that one of the reasons for our lack of rigor compared to physical and behavioral sciences is that we have not given enough attention to the theories that underpin our work, both as software engineers and as software engineering researchers. I present my general theory about software engineering and then propose two simple theories, D and E as the basis for laying out a unified theoretical foundation for software engineering and software engineering research. Software Engineering consists of two logical parts: design and empirical evaluation (both terms used in their broadest senses). I propose theory D to as the theoretical basis for the design part, and theory E as the theoretical basis for empirical evaluation. These two theories are then composed in various ways to lay out a space (a taxonomy, or ontology if you will) for software engineering. Finally, I claim that software engineering and software engineering research (both fully integrated with empirical evaluations) are models for the atomic and composed theories.

1. Introduction

The motivation for this research is twofold: 1) to establish a unifying foundation for software engineering, and 2) to establish the same rigorous empirical foundations for software engineering that we find in natural and behavioral sciences. In natural sciences, their rigorous basis rests on 1) theories that have to be testable, 2) testing done in the physical world that 3) provides hard constraints on the theories. In behavioral sciences, their rigorous basis rest on 1) theories that have to be testable, 2) testing done in the behavioral world that 3) provides probabilistic constraints.

Currently, we do not have this same rigor in the sciences of the artificial [5]. Indeed, we are woefully inadequate with respect to empirical studies. Granted, as a field we are improving, but we are a long way from achieving the rigor we find in both natural and behavioral sciences. It is certainly easy to see why: in natural sciences education, students are subjected to a stream of experimental work in the laboratory components of their basic courses; in behavioral sciences, students are subjected to experimental design and experimental statistics courses as both undergraduates and graduates.

2. Experimental Science

Let us first take a basic look at science, even though one might argue that it is not necessary since everyone understands it thoroughly. My reason for doing this is to set the stage for the theories and models relevant for empirical software engineering.

Science is basically an iterative process consisting of the following steps (see Figure 1):

- Observations and abstractions are use to create a theory T.
- We test theory T against reality W with an experiment E using one or more instruments I.
- We then reconcile theory with reality.
- When predictions don't agree with reality, we change the theory.

Gooding et al. [1] argue for the critical importance of the instruments we use in experimental work. They are the lens through which we observe the world. To paraphrase Wittgenstein [7], *the limits of my instruments are the limits of my world*. They enhance, limit, and color our view of the world. In natural sciences, instruments are often physical creations; in behavioral sciences they are often intellectual creations. Humans are common instruments in both. Instruments may be active or passive. They may be theory-laden or transparent and neutral. They may be reliable and standardized or not. In any case, they are a critical part of the empirical apparatus and as such will play a critical part in any scientific endeavor.

3. Natural, Behavioral & Artificial Sciences

In remedying our lack of rigor, it is critical to understand how the sciences of the artificial differ from, and are similar to, behavioral and natural sciences. Obviously, we must have theories that are testable just as they do. The differences come in the context of testability and the constraints faced. The sciences of the artificial have some aspects in common with natural and behavioral sciences: testing is done in both physical and behavioral contexts. However, testing is also done in intellectual and technological worlds as well. For the physical and behavioral contexts we have the same hard and probabilistic constraints. For the technological context, we have selectable constraints – i.e., we have constraints we can select among, perhaps arbitrarily. For the intellectual context, we have malleable constraints – i.e., we have constraints that we can change, also arbitrarily.

There are interesting differences between natural and behavioral

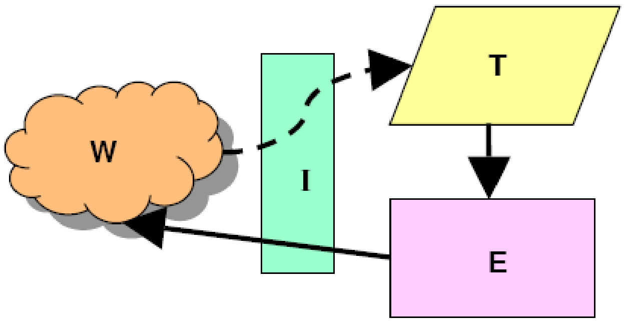

Figure 1: Basic Science

sciences that are relevant to design disciplines. The general goals of natural sciences are to understand natural phenomena and create a theoretical basis for prediction. Further, natural sciences provide a basis for invention and engineering. The general goals of behavioral sciences are to understand human and societal phenomena and provide a theoretical basis for prediction *and interventions*. The inventions and interventions are important because of the need to change the world and which is one of the fundamental goals of software engineering: build systems of practical value in the world [3]), not merely to observe it though, of course, we do need to observe our software systems and make predictions about them as well.

4. My Theoretical Approach

My theory about software engineering (which I claim to provide a unifying foundation) comes from my experience as a practicing software engineer and from my experience as a software engineering researcher.

Software Engineering consists of two logical parts: design and empirical evaluation (both terms used in their broadest senses). I propose two simple theories, D and E as the basis for laying out a unified theoretical foundation for software engineering and software engineering research. I propose theory D to as the theoretical basis for the design part, and theory E as the theoretical basis for empirical evaluation. These two theories are then composed in various ways to lay out a space (a taxonomy, or ontology if you will) for software engineering. Finally, I claim that software engineering and software engineering research (both fully integrated with empirical evaluations) are models for the atomic and composed theories.

4.1 Theories and Models

The terms "theory" and "model" are used and misused in a variety of ways, often informally and interchangeably. I want to use them in a very specific way: a theory (a more or less abstract entity) is reified, represented, satisfied, etc by a model (a concrete entity).

This view of theories and models is derived in part from Turski and Maibaum [6] where they state *"A specification is rather like a natural science theory of the application domain, but seen as a theory of the corresponding program it enjoys an unmatched status: it is truly a postulative theory, the program is nothing more than an exact embodiment of the specification"*. I note, however, that I want a theory to be broader than a specification and, more than likely, less formal.

We often use models as a representation of a theory. In natural sciences, the model is often a set of mathematical formulas. In logic, a model is an interpretation of a theory and has certain logical properties. Here again, I want to broaden the notion of a model to be a representation (indeed, a reification) of the theory. The model is of paramount importance in design disciplines as it is the visible manifestation of the theory. Of fundamental importance is the fact that a theory can have an arbitrary number of models.

4.2 More About Theories

My claim is that the key to a unifying, and a rigorous and systematic foundation for software engineering, software engineering research, and empirical studies in software engineering and software engineering research is to be found in a focus on theory.

So what is it that I consider to be important in theories: 1) the source of the theories; 2) the structure of the theories; and 3) the use of the theories.

Source of Theories. In terms of sources of theories relevant to software engineering, three different types of theories are important:

1. Scientific theory – Scientific theory is based on *observations of* the world. They change on the basis of new observations, or new interpretations of observations.
2. Legal theory – Legal theory is quite different: it is based on *decisions about* the world, and is changed on the basis of new decisions or new interpretations of decisions.
3. Normative theory – Normative theory is different yet and is based on a system of philosophical tenets about what is good and bad, and judgments are changed on the basis of new inferences from those tenets or new interpretations of them.

Theories in design disciplines are a combination of all three of the above. They are based on observations, decisions, and judgments about the world. They change on the basis of new observations, decisions, and judgments or on the basis of new interpretations of those observations, decisions and judgments.

Structure of Theories. Markus and Robey [4] distinguish two different theory structures:

1. Variance – In the case of variance, the theoretical structure is a set of laws about interactions or relationships. For example, given a variation in A, what other units can be linked to A such that they account for the variance in A.
2. Process – In the case of process, the theoretical structure is a temporal ordering of activities, steps, or events.

We find both kinds of theoretical structures in design discipline theories depending on what kind, and at what level, we are theorizing about design issues.

Use of Theories. The taxonomy of uses I describe here is derived from Gregor [2]. I distinguish five distinct uses of theories that may be used also in combinations:

1. Description – A theory is used to describe phenomena in terms of its constructs, properties, and relationships, and the boundaries within which those properties and relationships hold. Descriptions are intended to be complete.
2. Prescription – A theory is used to provide a set of constraints on its constructs, properties, and relationships, and the boundaries within which those properties and relationships hold. Prescriptions are intended to emphasize the crucial aspects of the theory.
3. Explanation – A theory is used to explain how, why and when things happen based on causality and methods of demonstration (that is, argumentation). The intent is to provide deeper understanding and insight into the subject phenomena.
4. Prediction – A theory is used to predict what will happen on the basis of necessary and sufficient conditions for the theorized phenomena. The phenomena *will not* happen if the necessary conditions are withheld; nor *will* they happen if the sufficient conditions are withheld.
5. Action – A theory provides principles, techniques, and methods for enabling the desired phenomena (for example, achieving a desired goal, or designing or constructing an artifact).

Depending on the context in software engineering, we make use of theory in all these different ways. Theories, of course, influence their models: the source of a theory will affect its model; the structure of a theory will influence the structure of its model; and, the use of a theory will also influence the structure of its model.

4.3 Model Calculus

Since theories as used here are informal entities, their composition is also informal and the resulting integration is done informally.

My theory about models, however, has a more formal definition and a set of rules for the model operators. My theory about models is as follows:

1. A model is a tuple consisting of two sets: a set of objects, and a set of transformations (or mappings) from an object in one set of objects to another object in a (usually different) set of objects, written as $A \rightarrow B$.
2. There are one to one transformations of mappings and there are many to one transformations. One to one mappings are indicated by $A \rightarrow B$ and many to one mappings are indicated by $A \times B \rightarrow C$, where A **x** B denotes a combination of objects in the Cartesian space of A and B.
3. Models can be composed to yield further models. How that is done depends on the intent specified by the composition. A model can be arbitrarily considered to be atomic – that is, its structure remains hidden – or open-structured. For example, composing and open structured model with an atomic model results in a model:

   OSM : AM =
   <{O},{T}> : AM =
   <{O}:AM, {T}:AM> =
   $<\{o_1:AM \ldots o_n:AM\}, \{t_1:AM \ldots t_n:AM\}>$

   On the other hand , composing an atomic model with an open structure model yields a number of models (depending on the number of objects and transformations).

   AM : OSM =
   AM : <{O},{T}> =
   $AM:o_1, \ldots, AM:o_n, AM:t_1, \ldots, AM:t_n$

   Each of these is a model *restricted* to that particular object or transformation.
4. There are rules that govern compositions and their effects on objects and transformations. A complete discussion of those rules and deeper aspects of models is beyond the scope of this paper and can be found in [8].

5. Design Theory & Model *D*

Design theory D has two parts: a theory about D and a theory about the model that reifies D.

5.1 Theory of *D*

Theory D is meant to capture the typical cycle of creating a theory that is then reified into a model where the model is then injected into the world and changes the world (see Figure 2). I summarize it as follows:

- We observe and abstract some specific part of the world and create a theory.
- From that theory we create a usable model to reify or represent that theory.
- We iteratively adjust both the theory and the model as our understanding of the theory and its model evolves, both iteratively and interactively.

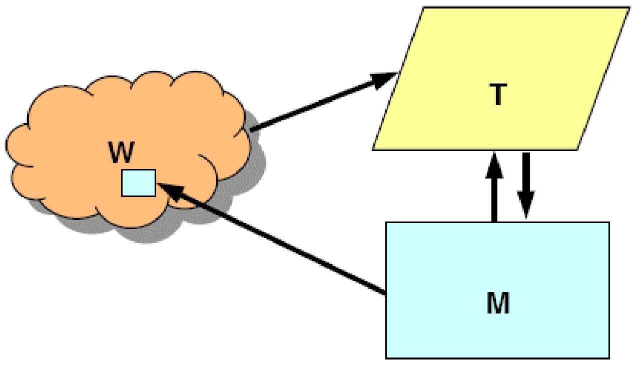

Figure 2 – Theory and Model *D*

- When satisfied that the model adequately represents the theory, we inject the model into the world.
- Injecting the model into the world changes the world.
- The changes brought about by these changes as well as other changes often lead to adjustments and extensions to the original theory.
- Changes to the theory in turn lead to further changes in the model and the world.

This abstract theory is then reified into a concrete model as described below.

5.2 Model of *D*

The model of D consists of three elements (objects) and six transformations (mappings, or, if you will, processes). The elements are as follows:

- **W** – The world, but more specifically, the part of the world relevant to the theory
- **T** – The theory initiated by observations and abstractions
- **M** – A model that reifies, represents or satisfies the theory T

The transformations involving these elements of the model are as follows:

- **W → T –** Generate a theory: observe and abstract from the world W to create a theory T
- **T → M –** From the theory T create/evolve a model M
- **T → T** – Evolve theory T until satisfied
- **M → M** – Evolve the model M until satisfied
- **M → T** – Change the theory T to better conform to model M
- **M x W → W** – Inject model M into the world W thereby changing it (which depends on both the model and the world before the injection of the model into it).

It should be clear that this model represents the theory of D above.

5.3 SE Design as a Model of *D*

I claim that the design part of software engineering at a suitable level of abstraction is a model of D. For example, W in theory D contains what Jackson [9] calls the *problem space*. It is that part of the world that represents the problem that we want to address with our software system. We observe and abstract from this problem space to create a theory T (i.e., theory T in D) of the problem we want to solve. We refer to T as requirements. W also contains what Jackson calls the *solution space*. It is in this space that we find the elements that we put together to create the model M (the software system itself) that reifies and represents those requirements T.

W→T is the process of deriving the requirements from the chosen problem space by observing and abstracting what is considered to be critical and central to the problem to be solved. It is also the process of understanding the effects of a changing world on the requirements that exist as the basis for an existing system M. T→M is the process of creating and evolving the model/system from the theory/requirements, while M→T is concerned about adjusting the theory/requirements to better conform to an existing model/system. This latter happens regularly as we find that some requirements may be too costly, too complex, or that time is too short, etc. And as the entire enterprise of design is an iterative venture, T→T and M→M are those processes of evolving both the theory/requirements and the model/system from its initial incomplete state eventually to its sufficiently detailed state. And, finally, M→W releases/injects the model/system into the world to be used in solving the intended problem, and, in doing so, often radically changes the world. This is often referred to as *technology transfer*

6. Empirical Theory & Model *E*

As I did with D, I here propose a theory about a theory and model for E – a theory about empirical evaluation. For purposes of explanation and illustration I use a very simple theory for E. A more elaborate theory for E will be introduced in future work to illustrate more fully empirical evaluations. It is sufficient at this point to indicate that empirical evaluations can range from very informal (as indicated by this formulations of E) to formal and controlled experimentation (as will be indicated by a more complete model of E).

6.1 Theory of *E*

Not surprisingly, the theory E is essentially a simplification of basic empirical science discussed above (see Figure 1).

- Given a theory T, generate an hypothesis H to test some part of the theory
- From the hypothesis H, generate an evaluation E.
- On the basis of the evaluation results, revise theory T.

I note that this is a very basic theory, but it still is sufficiently rich to cover the entire range of studies from exploratory through to rigorously explanatory studies. Of course, theory T may be vague and ill-formed (as it would be for exploratory work) or well-formed and mature (as it should be when doing explanatory work). Similarly the hypothesis may be generic and open-ended or focused and specific. Evaluations E may be human and opportunistic (for exploratory work) or specifically and well-designed. Further, the theory of *E* supports both theory generation (in the case of exploratory work) and focused evaluation of existing theory.

6.2 Model of *E*

The basic elements in the model and their interrelationships are: theory **T**, hypothesis **H**, and evaluation **E**.

The following transformations represent the processes of conducting an empirical study.

- **T → H** – derive an hypothesis H from theory T
- **H → E** – create an appropriate evaluation based on H
- **E x T → T** – reconcile theory and reality – i.e., on the basis of the evaluation and the current theory T, revise T.

7. Evaluating the Design Theory *D – ED*

It is here in the evaluation of the design part that we find the other half of the software engineering enterprise. It is here we determine the adequacy and utility of our theories and models, the efficacy of our processes in deriving these theories and models.

To evaluate the design theory D, we compose an atomic model of E with an open structured model of D giving us the following models:

- evaluation of the world of D, **E:W**; evaluation of the theory of D, **E:T**; and evaluation of the model of D, **E:M;**
- the evaluation of the processes of
    - creating a theory T from the world W – **E:(W → T)**;
    - creating a model M from theory T **– E:(T → M);**
    - evolving theory T **– E:(T → T);**
    - evolving model M **– E:(M → M);**
    - adjusting theory T to be consistent with model M **– E:(M → T)**; and
    - evolving he world as a result of injecting model M into it – **E:(M x W → W).**

Among the kinds of questions the evaluations must address are the following: the adequacy of D.T representing some part of W; the adequacy of D.M representing D.T; the utility of D.M in the world D.W; the effectiveness of such transformations as creating D.M from D.T, evolving D.T and D.M, or of creating D.T from D.W.

8. Designing Design – Theory *DD*

Theory DD (the composition of D with itself) is meant to capture the typical cycle of creating a theory of D (i.e., a theory of producing a design product) that is then reified into a model of D and the model is injected into the world and changes the world (see Figure 2). I summarize it as follows:

- We observe and abstract some specific part of the world and create a theory of
    - What the world of D is like
    - What form a theory in D should take
    - What form a model in D should take
    - What form the processes of creating the theory and its model of D should take
    - How the resulting model of D should be injected into the world
- From that theory we create a usable model to reify or represent that theory of
    - What the world of D is like
    - What form the theory in D should take
    - What form the model in D should take
    - What form the processes of creating the theory and model D should take
    - How the resulting model of D should be injected into the world.
- We iteratively adjust both the theory and the model as our understanding of the theory and its model evolves, both iteratively and interactively.
- When satisfied that the model adequately represents the theory we inject the model into the world.
- Injecting the model into the world changes the world
- The changes brought about by this injection as well as other changes often lead to adjustments and extensions to the original theory.
- Changes to the theory in turn lead to further changes in the model and the world.

The composition of model D (as an open structured model) with itself (as an atomic model) results in a new model with the following elements: the world of D, **W:D**; the theory of D, **T:D**; and the model of D, **M:D**.

The transformations involving these elements of the model generate the following:

- **W:D → T:D –** Generate a theory: observe and abstract from the world of D to create a theory D
- **T:D → M:D –** From the theory of D create a model of D
- **T:D → T:D** – Evolve theory of D until satisfied
- **M:D → M:D** – Evolve the model of D until satisfied
- **M:D → T:D** – Change the theory of D to better conform to the model of D
- **M:D x W:D → W:D** – Inject model of D into the world of D thereby changing it (which depends on both the model and the world before the injection of the model into it).

It should be clear that this model represents the theory DD.

The composition of model D (as an atomic model) with itself (as an open structured model) yields nine models: **D:W**, **D:T**, **D:M, D:(W → T)**, **D:(T → M), D:(T → T), D:(M → M), D:(M → T)**, **D:(M x W → W)** – i.e., the design of each of the elements in the design model D, and the various transformations that take place in the design model D.

To illustrate the richness of compositional results, consider D:T where we now view D as an open structured model. We first get the objects **WT**, **TT** and **MT** (the world of T, the theory of T, and the model of T respectively). We also get the following transformations: **WT → TT)**, **TT → MT, TT → TT, MT → MT, MT → TT**, and **MT x WT → WT,** exactly analogous to the transformations of D. The same follows for each of the remaining composed models above.

Analogous to my claim that the design aspect of software engineering is a model of D, I also claim that the design aspect of software engineering research is a model of DD, and it is here that things get really interesting.

DD.TW, DD.MW – world of software development. The world of software systems is a varied and multi-faceted world. It is a world of problems and solutions [9]. It is a world where some problems are not solvable at all by automation as well as a world where some problems are just too hard to solve at all [10]. For the problems that are solvable, there are those that are solvable by what Vincenti [11] calls normal design and those that are solvable only by radical design. We may or may not be successful in solving problems that require radical design, but when we are successful we almost always need several iterations before we achieve that success.

It is a world of rapid technological change where software-intensive systems are increasingly invading our lives, where computation is constantly getting faster and cheaper, and where electronic storage is getting larger, faster and cheaper as well. It is a world where the bases for design decisions are constantly changing, where the tradeoffs we previously made must be re-examined in the light of the current state of the world.

DD.TT, DD.MT – theories/models of requirements. Frustratingly, there is little theory that is explicit in DD.TT or DD.TM; it is by-and-large implicit. Or, more specifically it is often stated normatively rather than descriptively (as one would find in natural sciences, for example). In one way, this is not surprising as our theories in D are largely normative: the system ought to do …; it ought to respond within …; it must provide …. Indeed, this normative approach is a feature of the sciences of the artificial [5]. And, of course, it is seen all too easily in every new *salvation du jour*.

However, as my goal in this paper is to lay a foundation for empirical software engineering, I claim that to make progress towards the kind of rigor we find in natural and behavioral sciences, that for this level of discourse we need to be more descriptive – that is, we need to be more explicit about our theories in such a way as to be easily testable. Ignoring those issues for the time being, let's consider some of the relevant theories found in DD. Please note that I am not trying to be in any way complete, or even representative. The intent here is merely to be illustrative.

Nuseibeh, Kramer and Finkelstein's multiple viewpoints [12] approach implicitly embodies theoretical implications about D.W, D.T and D.W→T: there are different stakeholders with respect to the problem to be solved; these stakeholders have different views on what is important in the software solution; these different views need to be captured in the requirements; and eventually any and all apparent and real conflicts need to be resolved to provide a consistent set of requirements (i.e., a consistent theory).

There are a wide variety of models we use for various aspects of D.T. For example, we often use scenarios to provide examples of behavior in T. We often provide checklists, templates, style guides, etc for both requirements documents (as well as system architecture, design and code) to represent the models for our theories of requirements and systems.

DD.TM, DD.MM – theories/models of software systems.

Common theories in DD about the form that a model D.M (or parts of the model) should take include structured programming, object oriented programming, aspect-oriented programming, etc. Looking at D.M in a different way, there are the theories about creating systems bottom up or top down, or about structuring them for future change, or about organizing them hierarchically, as networks of cooperating processes, or to reflect the shape of the problem. There are those who theorize that the components in software systems should be orthogonal and each component do one thing well, while others such as Jackson indicate we should be mindful of the fact that the world where we find our problem space has been implemented with the full exploitation of the Shanley Principle [9] of efficient design where each element serves multiple purposes.

There are a variety of theories in DD about how we do the transformation from requirements to the system (D.T→M). The more or less standard ones include waterfall development, Boehm's spiral development, refinement, etc. A more radical departure from these standard approaches is that of Extreme Programming. An interesting variation of refinement can be found in Batory's algebraic compositional approach [13].

9. Evaluating the Theory *DD – EDD*

To evaluate the design theory of DD, we compose an atomic model of E with an open structured model of DD giving us the following:

- the evaluation of the world of D, **E:(W:D)**; the evaluation of the theory of D, **E:(T:D)**; and the evaluation of the model of D **E:(M:D);**
- the evaluation of the processes of
  - creating a theory T:D from the world W:D – **E:(W:D → T:D)**;

  - creating a model M:D from theory T:D – **E:(T:D → M:D);**
  - evolving theory T:D – **E:(T:D → T:D);**
  - evolving model M:D – **E:(M:D → M:D);**
  - adjusting theory T:D to be consistent with model M:D – **E:(M:D → T:D)**; and
  - evolving he world as a result of injecting model M into it – **E:(M:D x W:D→ W:D).**

As DD was significantly more complex than D, so EDD is significantly more complex than ED. Despite this increased complexity, the aims are still the same as with ED. Its just that there are many more elements and processes to evaluate. The space is much larger. But of course that is to be expected when we are concerned with theories and models about theories and models as we are in software engineering research.

However, just as the software engineering of software systems is composed of design and evaluation, so to is research about the design and evaluation of software systems composed of both design and evaluation.

10. Evaluating Evaluations – *EE*

To evaluate the empirical theory E, we compose an atomic model of E with an open structured model of E giving us the following:

- the evaluation of the theory of E, **E:T**; the evaluation of the hypothesis H of E, **E:H**; and the evaluation of the evaluation of E, **E:E;**
- the evaluation of the processes of
  - creating a hypothesis H from theory T – **E:(T → H)**;
  - creating an evaluation from hypothesis H – **E:(H → E);**
  - evolving theory T as a result of the evaluation E – **E:(E x T → T).**

The issues that need to be considered here are those concerning the adequacy of the evaluations and the effectiveness of the evaluation processes. Among the critical issues are those such as the relevance of the hypothesis to the theory, the relevance of the empirical evaluation to the hypothesis, and the standard problems of construct, internal and external validity.

11. Designing Evaluations – *DE & DEE*

To design the empirical evaluation theory E, we compose an atomic model of D with an open structured model of E giving us the following:

- the design of the theory of E and its evaluation, **D:E** & **D:(E:T)**; the design of the hypothesis H of E and its evaluation, **D:E** & **D:(E:H)**; and the design of E and its evaluation, **D:E** & **D:(E:E);**
- the evaluation of the processes, and the evaluations, of
  - creating a hypothesis H from theory T – **D:(T → H) & D:(E:(T → H))**;
  - creating an evaluation from hypothesis H – **D:(H → E)** & **D:(E:(H → E));**
  - evolving theory T as a result of the evaluation E – **D:(E x T → T)** & **D:(E:(E x T → T)).**

The design of empirical evaluations and the design of evaluating empirical evaluations (DE and DEE) is analogous to DD: it is part of the software engineering research enterprise.

12. Conclusions

I propose theory D as the theoretical basis for the design part of software engineering, and theory E as the theoretical basis for the empirical evaluation part (which is the composition ED). These two theories and the composed theories then lay out a space (a taxonomy, or ontology if you will) for all of software engineering and software engineering research.

From these two theories I have created a set of various composed theories that focus on various aspects of design and evaluation. The first composition is that of ED in which we actually realize the empirical evaluation part of software engineering. The composition of D with itself, DD, gives us the design portion of software engineering research, while EDD provides us with the empirical evaluation of our research. The empirical evaluations themselves need to be empirically evaluated and E composed with itself, EE, provides that. Of course, there is then the design of the empirical evaluations that we represent with the compositions DE and DEE.

These two initial theories and their compositions lay out a very rich space for our field. And, it is on this basis that I claim to have provided a unifying theoretical basis for a rigorous software engineering and software engineering research discipline.

Moreover, this approach is even more general than that. I also claim that such design disciplines as project management, instrument creation and evolution, empirical studies themselves are also models of these atomic and composed theories – that is, they are design disciplines and hence models for the theories of design disciplines that I have presented and alluded to.

13. References

[1] David Gooding, Trevor Pinch, and Simon Schaffer, Editors. *The Uses of Experiment: Studies in the Natural Sciences.* Cambridge: Cambridge University Press, 1989.

[2] Shirley Gregor. "The Nature of Theory in Information Systems", MIS Quarterly 30 (2006), pp 611-642

[3] Michael Jackson. The World and the Machine, 17th International Conference on Software Engineering, 1995, Seattle WA, 283-292

[4] M. L. Markus and D Robey. "Information Technology and Organizational Change: Causal Structure in theory and Research", *Management Science* 34:5 (1988), pp 583-598

[5] Herbert A. Simon. *The Sciences of the Artificial.* Cambridge: MIT Press, 1969.

[6] Turski, Wladyslaw M. and Maibaum, Thomas S. E. *The Specification of Computer Programs*. Reading, Mass: Addison-Wesley, 1987.

[7] Wittgenstein, Ludwig. *Tractatus Logico-Philosophicus*, ed. A. J. Ayer, London: Routledge & Kegan Paul, 1961. Section 5.6

[8] www.ece.utexas.edu/~perry/work/tm/modelcalculus/

[9] Michael Jackson. The World and the Machine, 17th International Conference on Software Engineering, 1995, Seattle WA, 283-292

[10] L. Fortnow, Steve Homer. A Short History of Computational Complexity. In D. van Dalen, J. Dawson, and A. Kanamori, editors, *The History of Mathematical Logic*. North-Holland, 2002.

[11] Walter G. Vincenti. What Engineers Know and How They Know It. Baltimore: The Johns Hopkins University Press, 1990.

[12] Bashar Nuseibeh, Jeff Kramer, Anthony Finkelstein. Expressing the Relationships Between Multiple Views in Requirements Specification, ICSE, 1993.

[13] D. Batory, J.N. Sarvela, and A. Rauschmayer. "Scaling Step-Wise Refinement", IEEE Trans. on Software Engineering, June 2004.