On the Structure of General Theories of Software Engineering

Dewayne E. Perry and Don Batory
The University of Texas at Austin
perry@mail.utexas.edu, dsb@cs.utexas.edu

Abstract

Taming complexity is a fundamental goal of software engineering. The core techniques that have been successful in practice are those that separate concerns, especially variants of architectural abstractions called components and connectors. We argue that General Theories of Software Engineering (GTSE) should be organized by components and connectors to distinguish conceptually distinct elements and their inter-relationships and interdependencies. Doing so, we argue, separates concerns that should be distinct and not conflated, thereby increasing the value of GTSE efforts.

1 Our Position

Software Engineering (SE) is an extremely broad field. It covers all domains for which software systems are feasible. Not surprisingly, there are core SE techniques and principles that are relevant to all domains as well as more focused techniques and principles relevant to specific domains. This is clearly evident in the typical courses covered in both Computer Science and Software Engineering curriculums.

A quote from the GTSE 2013 workshop report [1] summarizes an important thread of discussion (the quoted references are from the report):

> *Participants agreed that theorizing takes many forms [17] and SE entails myriad phenomena; for instance, Perry [11] distinguishes between software engineers, SE, and software project management. Ralph [12] consequently suggests formulating a multi-level GTSE, i.e., a theory that crosses many units of analysis including individual, team, artifact, process and project. A core question then is: What might the different levels of a GTSE contain?*

We assert that *any* *General Theory of SE (GTSE)* should be constructed from two entities: component theories and connector theories. *Component theories* are about the major components in software engineering: economics, management, software engineers, and SE. *Connector theories* are about the relationships and interdependencies among component theories. Cognition, for example, is a critical element of a component theory about software engineers (people); structural complexity is a critical element of a component theory about software systems (SE). A connector theory would express the relationship between cognition and complexity.

Further, we know from experience that the basic structure of components and connectors is recursive: component theories can be expressed as aggregations of more restricted component and connector theories; connector theories can be decomposed in terms of simpler component and connector theories. We believe that an organizational structure with component and connector theories is fundamental to a GTSE; it provides the means to structurally separate conceptually distinct elements and their inter-relationships and interdependencies.

In what follows, we provide an example of a GTSE structure in terms of component and connector theories (in some cases recursively) to illustrate our approach; it should be a platform to guide and organize future discussions on GTSE. Our illustrations, by themselves, are not intended to be complete theories, but rather suggestive of what is possible.

2 Motivation for a Theory of Architectural Structure for GTSE

A sound and foundational principle in SE is the *separation of concerns* [6]. It is critical to the design and implementation of software: organizing and encapsulating concerns of a system that are related is one of the fundamental intellectual tools we have at our disposal to manage complexity [1], the core problem in building and evolving software systems. The goal is to separate various concerns and encapsulate them in distinct modules.

We assert that we should do no less in the creation and evolution of theories about the entire enterprise of SE. Just as we use modularity and encapsulation as intellectual tools to manage the ever besetting problem of complexity in designing, building, and evolving software, so should we utilize modularity and encapsulation – separating concerns – in theories about SE in general. It is only by doing so that we have *any* chance of managing the inherent complexity we face in theories of SE. Further, it is the only chance we have of eliminating unnecessary or accidental complexity that invades and further obfuscates existing theories [2]. Component and connector theories structurally separate the myriad concerns in precisely the right way.

Ask yourself: was Einstein's Theory of Relativity influenced by the price of bagels? Of course not; it is a ridiculous idea. Was Relativity influenced by the organization or personalities at Los Alamos in the 1940s? Of course not. But did economics, organization, or personalities influence the development of the atomic bomb? Yes, they did. Theories of science/engineering are separable from theories of economics, organization, and human relations. As we will argue, their influences *will* lead to the selection of *particular* technologies, organizational structures, economic choices, and management practices – as dictated by their conceptually separate theories. In this sense, SE is no different than other engineering disciplines where the conceptual separation of technologies, organizational structure, economics, and management are routinely practiced.

We recognize four major component theories in building and evolving software systems:

(1) strategic and tactical concerns about a product – economic concerns,

(2) project management concerns,

(3) people concerns, and

(4) technical concerns.

(1) is primarily of interest to management and entrepreneurs. (2) is primarily of interest to project management. (3) is a focus of business and project management. And, we assert, only (4) is of interest to software engineers.

There should be separate (component) theories for each concern. Management and entrepreneurs should create their theories to be independent of the theories for project management, independent of theories for software technology. These component theories are related by connector theories that define dependencies and interactions – the choice of a particular technology should impact a component economic theory by altering (or supplying) its distinctive parameter values; the actual details of technology are hidden behind these parametrics. The technical details of how a technology works belongs to a component theory about engineering, *not* management.

The history of science is replete with separate, albeit related theories, and this is reflected in the academics. There are courses on Newtonian mechanics; there are separate courses on the application of Newtonian mechanics to Civil Engineering. In the engineering of software, we see similar trends. There is mounting evidence that *Domain-Specific Languages (DSLs)* or domain-specific models in *Model Driven Engineering (MDE)* are at the forefront of research in specifying target domain-specific systems.

Each domain has its own peculiarities; generally two domains have little in common. But it is the deep engineering knowledge and experience that is captured in DSL and/or MDE approaches and their integration (which would correspond to our notion of connector theories) that leads to a coherent whole. The idea of using a complex universal language is eschewed for an integration of "island" theories-or-languages that are honed for domain-specific tasks.

In the following, we explore the idea of component theories and connector theories. One of the core principles of our approach is recursion – the ability to compose component theories and connector theories to describe more abstract component and/or connector theories. To paraphrase a phrase popularized by Stephen Hawking, it is "turtles all the way up" or "turtles all the way down" [7].

3 Component Theories

We identify four core component theories together with their sub-component theories: economics, project management, software engineers, and SE.

3.1 Economics

Businesses worry about strategic and tactical marketing decisions which ultimately come down to economics. While economic decisions may very well impact what software or platform to use (e.g., Windows vs. Linux), the details of operating systems are abstracted away, exposing only parametric models that represent the economic levers and constraints of alternative technologies. The basic problems addressed by a component theory are those of core competencies, market window (e.g., timing), estimated demand, and estimated cost (e.g., resources, effort, and time).

3.2 Project Management

Fundamental to any project, software or otherwise, is the monitoring of progress and the management of resources. It can expose and/or reveal the historical way software has been developed (and will continue to be developed) in a business. Decisions about project management may very well impact what software technology to use, but again, the details of these technologies must be abstracted away for decision making to be practical.

A theory of project management might be broken down into basic sub-component theories: planning, monitoring and metrics, and resource allocation.

3.2.1 Planning

A sub-component theory for planning elaborates the issues of effort estimation, resource costs, and project planning and project constraints (which in turn are potential sub-sub-component theories).

Effort Estimation. Currently there are two approaches generally used for effort estimation: function points and lines-of-code. The former has a firmer foundation in that function points are derived from software system requirements while the latter are based on estimates on how large the various components in the system will be based on past experience. There are established theories for each approach [8].

Resource Costs. Resource costs may be broken up into a variety of categories. For example, people, hardware, various forms of support, COTS, etc. What is critical here depends significantly upon the type of project, the domain of the system, etc. [9].

Project Planning. Whatever cost and effort estimations are used for a project, they are critical for project planning – that is, they lay out the timelines, milestones, and allocations for a project to succeed. All this is pretty much ordinary management in the domain of software system production [10].

Project Constraints. Project constraints result from a combination of project plans, business strategies, and market forces (so there is an interdependence here, in addition, with economics and hence a connector theory that relates economic considerations with project constraints). For example, a decision to focus on Windows and Linux and ignore Apple, would constrain theories (a.k.a. models) for planning, business goals, and marketing.

3.2.2 Resource Allocation

Given a plan, various resources are allocated to execute the plan. A wide variety of resources are needed for a software engineering project: space, staff, technology, etc. A component theory of resource allocation might well have sub-component theories for effective space utilization (for example, the design of IBM's Santa Teresa Labs [25]), staff structure, etc.

3.2.3 Monitoring and Metrics

Given that the project has been adequately planned and resources appropriately allocated, the primarily task of project management is to monitor the progress of the project. In doing this, a variety of metrics are used to chart the progress of the project and the quality of the software system as it progresses. Interestingly, what metrics are used depends on the CMMI level of the project management organization. The failure to meet the established plans results in a very iterative cycle: plan/re-plan, allocate/reallocate resources, and monitor with the appropriate metrics [11].

3.3 Software Engineers

One theoretical approach is to view software engineers both as individuals and as members of teams. In the former, we might develop sub-component theories about desired capabilities, training and education, and experience and judgment. In the latter, we might develop sub-component theories about team formation and team structure. Further, we might develop connector theories that relate the various characteristics of software engineers to various team formations and structures. Other possible connectors are those relating cognitive abilities to training and education, etc.

3.3.1 As Individuals

Cognition. Cognition obviously plays a critical role in the various activities in the life of a software engineer. For example, Sackman, Erickson and Grant in their seminal study of programmers [3] were among the earliest to find an order of magnitude difference in productivity in programmers: "These studies revealed large individual differences between high and low performers, often by an order of magnitude**."** Equally interesting is the finding that the "two studies suggest that such paper-and-pencil tests may work best in predicting the performance of programmer trainees and relatively inexperienced programmers."

A wide variety of studies about cognition can be found in the Empirical Studies of Programmers (ESP) Workshop Series, 1986-1996. We summarize several studies reported there as examples of the wide variety of component cognition theories that are possible in a general theory of software engineering. In addition, they would entail a wide variety of connector theories as well.

Programmers as Knowledge-Based Understanders. Letovsky [4] in ESP 1986 provides "an empirical study of the cognitive processes involved in program comprehension" – obviously a critical issue in the creation and evolution of software systems. They used "Verbal protocols . . . from professional programmers as they were engaged in a program understanding task" from which "several types of interesting cognitive events were identified."

There are three components in knowledge based understanding:

- **a knowledge base**, which encodes the expertise and background knowledge which the programmer brings to the understanding task.

- **a mental model** which encodes the programmer's current understanding of the target program. This model evolves in the course of the understanding process.
- **an assimilation process** which interacts with the stimulus materials (target program code and documentation) and the knowledge base to construct the mental model.

Their results are summarized as follows:

> *We have shown that data from the verbal protocols can be analyzed into fragments of events. We have focused on two types of events, namely questions and conjectures, and described how these are organized into a larger event type we call an inquiry. We developed taxonomies for questions and conjectures, and then analyzed the various categories to develop crude theories of the mental representations and processes that produced them.*

Distributed Cognition in Software Teams. Flor and Hutchins [5] introduce a new approach: *distributed cognition*, for collaborative activities and provide a case study of team performance while doing system improvements (i.e., during perfective software maintenance).

> *Distributed Cognition takes as its unit of analysis a complex cognitive system: collections of individuals and artifacts that participate in the performance of a task. The external structures exchanged by the agents of complex cognitive systems comprise its "mental" state and unlike individual cognition, where mental states are inaccessible, these states are observable and available for direct analysis. Through analysis of these structures, their trajectories through the system, and their transformations, it will be demonstrated that complex cognition systems engaged in software development tasks possess cognitive properties distinct from those of individual programmers.*

The following is a list of these cognitive properties:

> *The Reuse of System Knowledge*
> *The Sharing of Goals and Plans*
> *Efficient Communication*
> *Searching Through Larger Spaces of Alternatives*
> *Joint Productions of Ambiguous Plan Segments*
> *Shared Memory of Old Plans*
> *Divisions of Labor and Collaborative Interaction Systems*

Clearly, a theory of cognition in the context of teams is needed because "Successful software development is viewed as a consequence not of a single programmer's cognition, but of an interaction of programmers and development artifacts in a system of distributed cognition."

Training, Education, and Experience. There exists a vast literature (see, for example, [21]) on education curricula delineating what our education and training specialists recommend as necessary preparation for computer science and software engineering. ACM's SIGSCE (Special Interest Group on Computer Science Education) sponsors regular conferences presenting the latest research approaches for educating and training computer scientists and software engineers. The IEEE Computer Society's Conference on Software Engineering Education and Training, CSEET, offers a similar research venue. Clearly there is a rich set of component and connector theories on achieving educational goals to create professional scientists and engineers.

Judgment. Brooks offers the theory in his *Mythical Man Month* [2] that great designers are born, not made. They can be mentored and developed, but there is an inherent talent – judgment, if you will – that enables them to create a *conceptual integrity* that is not possible otherwise. While the designs of ordinary designers can be improved with training, education and experience, they will never reach the level of quality of those designs from these inherent great designers.

3.3.2 As Members of Teams

Two possible team component theories are those of team formation and team structure. One could also easily consider theories of project structures as well, as is done by Allen in his seminal book, *Managing the Flow of Technology* [12], and Grinter et al., in their paper "The Geography of ' Coordination: Dealing with Distance in R&D Work" [13]. The interesting question is whether these theories would be component or connector theories. One could easily make the case that a project structure is about the relationships and interdependencies among a variety of different teams.

We note that the difference between a component theory and a connector theory may be more intentional than structural for complex connectors.

Team Formation. One possible component theory of team formation is delineated by McGrath [time-matters] and confirmed in a case study by Perry et al. [14]. The team building process of McGrath is delineated by a sequence of stages of team activity:

1. Goal choice – inception and acceptance of project goals
2. Means choice – agreement on the solution of technical issues
3. Policy choice – resolving conflicts and political issues
4. Goal attainment – doing the work achieving the intended goals

We found that critical factors in achieving an effective team formation were time to guild trust among the team members, face to face time to build working relationships, clear agreement on the team goals, and sufficient communication bandwidth to support the level of the needed interfaces among team members.

Team Structure. An interesting component theory of team structure is that of an empowered, interdisciplinary team [15] that works together throughout the entire project. This theory contrasts with the usual functional theory of team structure where the teams all work together doing the same thing: there is a requirements team, an architecture team, a design team, an implementation team, a test team, etc., and a decision structure that relied on management decisions about technical issues. This interdisciplinary structure was motivated by two considerations: error injection that occurred during handoffs between teams, and time blocked waiting for technical decisions to go up the management ladder and back again.

The interdisciplinary structure enabled the team to increase the effectiveness of intra-team communication, to increase the level of parallel development, and to reduce significantly the time lost waiting for technical decisions. The project cycle time was reduced by 25% and defects were removed earlier and very few faults were found after integration, thereby increasing the level of overall quality of the product.

In interesting side effect of this new team structure was that managers were able to manage twice as many teams, enabling them to focus on management issues and leave technical issues to the interdisciplinary teams.

3.4 Software Engineering

The most obvious examples of component and connector ideas stem from the vast field of software design. At an abstract level, virtually all design representations created in the last 30 years express software designs as graphs: nodes are components and edges are connectors. Depending on the representation, edges can represent relationships (i.e., mathematical relations) or they can represent software that permits components to communicate and/or interact with each other. Let's review some basic examples.

3.4.1 Software Architectures

Components and connectors express software architectures [16]. Components are objects (that have methods, an identity, and typically have state); connectors represent (at a minimum, though they can be more complex) a communication structure (typically provided by an operating system or component platform) that enables objects (components) to exchange messages. Classical distributed object platforms like CORBA, COM, and DCOM materialized these ideas; *Service Oriented Architectures (SOA)* are next generation distributed-object technologies.

Separation of concerns is clearly present: computations are distinguished (and modularized differently) from that of communications. A computation could be a large and complex program – which itself could be described by components and connectors (e.g., system of systems). And a connector could be implemented as simply as a method call in a programming language, or it could require a sophisticated infrastructure (remote call procedures or a component platform) that could itself be a large and complex program. It is here that the recursive nature of component and connectors (and their theories) are clearly evident.

3.4.2 UML Diagrams

Class diagrams follow a similar lead: distinguish classes from relationships. Classes represent modules; associations are connectors (which can be of two distinct types: inheritance and associations). State charts distinguish states (tiny components) from transitions (their connector/relationships). Message sequence charts distinguish objects from their communications. Object diagrams distinguish objects from their relationships.

In all, UML asserts that the complexity of software can be reduced to entities (objects or components) and their relationships (connectors).

3.4.3 Model Driven Engineering

A *metamodel* in MDE is a class diagram that captures the fundamental entities and relationships in a domain of applications. Instances of a metamodel, called *models*, are abstract descriptions of particular applications in that domain.

It is common to compose different metamodels to describe applications that integrate different domains. As different domains have different terminologies, entities and relationships, metamodel composition requires a third metamodel to define corresponding ideas (entities, relationships) to relate elements of one metamodel to another. Figure 1 shows the basic idea: to compose metamodels M_1 and M_2, a third metamodel W is needed; the 'composition' of M_1, M_2, W yields a single model C. In the MDE literature, W has several names: a *weaving* metamodel or a *pullback* metamodel; C also has several names: the *woven* metamodel or a *pushout* metamodel.

$$\begin{tikzcd}
& W \arrow[dl] \arrow[dr] & \\
M_1 \arrow[dr] & & M_2 \arrow[dl] \\
& C &
\end{tikzcd}$$

Figure 1 Metamodel Composition

Again we see separation of concerns: metamodels are components – units of reusable design. The connectors between models (which, in the general case, can be quite sophisticated and detailed) express correspondence relationships between common elements in M_1 and M_2 (themselves entities or relationships).[1] All of these models can be decomposed into more primitive components (classes) and their relationships (inheritance or associations).

[1] Readers may note that the "structure" of Figure 1 is isomorphic to the integration of theories for management (read M_1) and programming technology (read M_2) in Section 2.

3.4.4 Feature Interaction

A *Software Product Line (SPL)* is a family of related programs. Members of a family are distinguished by the features they possess. A feature is a modularized increment in program functionality. SPL technology maps a declarative specification of a program (i.e., a set of features) to its implementation. In general, if there are n optional features, there could be up to 2^n distinct programs in an SPL.

Any pair of features, say M_1 and M_2, can interact. That is, M_1 works correctly when M_2 is absent, and M_2 works correctly when M_1 is absent. But when M_1 and M_2 are present in the same program, one or both no longer work properly. A third feature, $W = M_1 \# M_2$, called a *resolution*, is added so that both M_1 and M_2 work correctly together. The composite feature integrates M_1, M_2, W to produce C. This is a 2-way interaction; the ideas generalize to n-way interactions.

We deliberately used names for features and feature interactions to call attention to the structural similarity of feature composition to metamodel composition in MDE and Figure 1. Features are components in SPLs. So too are their interactions. The connectors between them express correspondence relationships.

Readers who are familiar with mediators will recognize the following connection [17]. A *mediator* (W) is a separate component than mediates the interaction of two different components (M_1 and M_2) so that they work together correctly. M_1 and M_2 are features and their mediator W is their interaction resolution. The composition of all three is C.

3.4.5 Software Design in General

Software design, in general, is a satisfiability problem: does there exist a design that satisfies a set of constraints (a specification, possibly with performance constraints). Often there can exist many such designs, at which point, software design morphs into an optimization problem[2]: which legal design maximizes some property or combination of properties (e.g., memory footprint, estimated performance, power usage)?

4 Connector Theories

There are relationships and interdependencies among the various component theories. We believe that the connector theories ought to be about direct relationships – one can always do a transitive closure over the entire architecture to find the indirect relationships and interdependencies. And as we may divide component theories into sub-component theories, so we may divide a connector theory into sub-connector theories to emphasize more finely grained relationships and interdependencies. We present two examples of connector theories: 1) an example of a relationship between software engineers and SE – namely, that of cognition, complexity and software structure, and 2) an example of a relationship between software engineers and project management – namely, that of software engineer experience and effort estimation.

4.1 Cognition, Complexity, and Software Structure

As Brooks [2] points out, complexity is the critical essential characteristic of software systems. He claims that "Software entities are more complex for their size than perhaps any other human construct." Wulf, London, and Shaw [18] hold a similar view: "Large programs, even not so large programs, are among the most complex creations of the human mind." Clearly then, complexity is a fundamental problem for SE. We claim there are two basic forms of complexity: intricacy and wealth of details. The former are often

[2] Relational query optimization is a classic example that illustrates this idea [22].

found in deep complex algorithms. The latter are found in virtually every system: there is a massive amount of generally shallow details.[3]

But complexity is not entirely an SE problem: it is also a software engineer problem. This is where the cognitive abilities of software engineers come into play. Curtis et al., in their study on complexity metrics and psychological complexity [19], note the following (emphasis added):

> *Differences among programs played an important role in these experiments. The Halstead and McCabe metrics provided some information about program differences, but there were other factors unassessed by these metrics which influence the psychological complexity of the programs. The metrics predicted programmer performance better on versions of programs which were unstructured or uncommented. By reducing the cognitive load on a programmer, information available from structured code and comments altered the psychological complexity of a program such that it was no longer accurately reflected by the complexity metrics.*
>
> *A distinguishing characteristic of psychological complexity is the interaction between program characteristics and individual differences, such as programming experience.*
>
> *If the ability of complexity metrics to predict programmer performance is to be improved, then metrics must also incorporate measures of phenomena related by psychological principles to the memory, information processing, and problem solving capacities of programmers.*

The primary issue then is the relationship and interdependency between *cognitive load* and *program structure*. Examples of SE techniques that reduce cognitive load include structured programming, modularity, encapsulation, abstraction, (and the greatest of these is abstraction). This list is not meant to be exhaustive but illustrative of SE techniques usefully related to reducing cognitive load. It would be the goal of this connector theory to cover thoroughly and completely the issues of this relationship and interdependence. Below we discuss basic issues to consider.

4.1.1 Structured Programming

As Curtis et al. mention, "information available from structured code and comments altered the psychological complexity of a program". Of paramount importance is the clarity of program structure that was achieved with structured programming. An additional virtue of structured programming that reduces a software engineer's cognitive load is that the static representation of the program provides useful insights into the dynamic structure of that program – that is, the static representation reflects the possible dynamic structure of the program. This is particularly important when analyzing and debugging a program.

4.1.2 Modularity

A fundamental general problem-solving technique is *divide and conquer*. Modularity is an SE technique that enables this problem solving technique and reduces cognitive load by limiting the amount of information found in an individual component (function, procedure, class, subsystem, etc.). Decomposing a system into components provides a fundamental technique for managing complexity, both structurally and cognitively.

[3] MDE is a case in point. Models capture the essence of a problem. Model-to-text mappings translate models into volumes of customized and boilerplate code. Models are typically much smaller than their code counterparts.

4.1.3 Encapsulation

The cognitive utility of modularity is enhanced significantly when coupled with encapsulation – that is organizing the elements of a module such that those elements are closely related to each other along some dimension of concern. A standard metric for measuring this relatedness is *cohesion*.

4.1.4 Abstraction

There are two fundamental forms of abstraction: parametric abstraction, and interface abstraction with information hiding. Parametric abstraction comes in a variety of forms. The simplest is that or functions and procedures where values are abstracted into parameters and the values supplied as arguments. This reduces cognitive load in two ways: (1) it simplifies a multitude of similar pieces of code and abstracts the differences to make it simpler to understand; and (2) it reduces the amount of code significantly where there is pervasive use of abstraction (i.e., it reduces complexity relative to the amount of details in a system). A further cognitive advantage due to parametric abstraction is that it localizes changes and reduces the effort and cognitive load of evolving the system structure.

There are more complex forms of parametric abstraction that are useful: type abstraction, function abstraction, etc. Interface abstraction with information hiding is one of our most powerful SE techniques in managing complexity and reducing cognitive load. The purpose is to present an abstract interface that is much simpler than the implementation and the concepts needed for that implementation. A well-understand example of this is provided in a typical file system. Its interface provides a very simple abstraction compared to its hidden implementation that must deal with file control blocks, disc allocation and de-allocation, the device complexities of reading and writing data from secondary storage, etc. This is a significant reduction in cognitive load.

Equally important is that the abstract interface provides a conceptual vocabulary – a *little language* – that provides a high level of abstraction and concepts appropriate to a particular small domain that make it much simpler and easily understood. An SE structural technique that exploits this is called a *virtual machine* where the machine is built in successive layers with each layer embodying higher levels of abstraction and richness of expression making each layer easy to understand and easily implemented with the appropriate concepts from the lower layers. A well-understood example of this is the structure provided by operating systems and programming languages: an assembly language provides the first layer of abstraction over the bare machine; the system implementation language and the operating system provide a much more useful layer (which is often split into two layers: the kernel and the OS services); and a programming language with its run-time system to provide an even higher layer of abstraction with which to build applications (which should follow this virtual machine approach as well in its implemented structure).

4.1.5 Summary of This Connector Theory

We provide this as an example (though incomplete) of the relationship between system structure and cognition – that is between software engineers and SE. Our presentation is but a start; there is much more to the rich connector theory that should be considered to fully understand the relationship between cognition and SE techniques and structures.

A useful way of doing this would be to decompose cognition into various sub-elements and develop a focused connector theory for each of these elements in their relationship and interdependence on SE techniques and structures.

4.2 Modularity, Encapsulation, Abstraction, and Object Oriented Technology

While logically modularity, encapsulation and abstraction are distinct concepts, they work most effectively as mechanisms for managing complexity when integrated together. OO technology does precisely that: integrate them together – thereby forming a succinct connector theory about their effective use. The notion of an object as state (data) together with its operations (on that data) provides the criterion for encapsulation. The class structure provides a mechanism for modularization and abstraction for this form of encapsulation, providing also a variety of choices as to how much and what is open or hidden (i.e., abstracted).

Abstract data types (as supported, for example, by Ada Packages [26]) provide a similar, but stricter, combination of these three mechanisms

4.3 Project Planning and Software Engineer Time Estimates

A critical part of project management's planning is the input from software engineers about their estimates of the size and time. For the purposes of illustrating an interesting connector theory, we focus on time estimates. Time estimates were a critical part of a series of time studies by Perry, Staudenmeyer, and Votta at Bell Labs [20]. The focus of these studies was on the effectiveness of software engineers in the context of a large system evolutionary development. One of the more interesting results was the fact that developers were only 40% effective – that is, they were blocked from work on the development 60% of the time; they were waiting for a needed resource before they could continue.

This effectiveness factor of 40% has a significant effect on time estimation and the importance of the distinction between *lapse time* and *race time*. Lapse time is the time a task takes from start to finish; race time is the actual time spent working on the task. Ideally lapse time and race time are the same. However, ideal conditions rarely happen. In these studies of how developers spent their time, the difference between race time and lapse time was a factor of 2.5 [20]. In the projects studied, software developers tended to give their time estimates in terms of race time. To derive a useful time plan of the project, the project manager had to factor in the projected difference between these race time estimates and expected lapse time estimates. In this case, the managed and derived, through project manager experience in this context, the estimated factor of 2.5. These studies validated his conversion factor. Thus, while a relatively small connector theory between project management's project planning and software engineer's time estimates (as a part of the larger issue of effort estimates), the distinction between race and lapse time is critical and needs to be correlated between software engineer experience and project management experience for accurate project planning.

5 Conclusions

We believe that a full general theory of software engineering is akin to a very large complex software system and as such needs to be decomposed appropriately as is done with our software systems. To this end, we propose structuring our theories using two types of elements: component theories and connector theories, to enable us to adequately and succinctly treat separate concerns. As an example, we propose 4 high level components to represent the entire software enterprise: business strategies and economics, project management, software engineers, and software engineering.

We illustrate our approach giving examples of component decompositions and connector theories that relate the elements in one component theory with those in another. Each of these component theories has its own domain specific properties and structure just as a software system architecture has its sub-architectures.

The simple elegance of this approach provides two basic elements that can be used recursively to expand the full space of general theories of software engineering. We can then modularize our theories and encapsulate related aspects together thus managing the complexity of our general theory.

6 References

[1] Johnson et al., "Report on the Second SEMAT Workshop on General Theory of Software Engineering (GTSE 2013)", ACM SIGSOFT Software Engineering Notes, September 2013.

[2] F.P. Brooks. *The Mythical Man Month – Silver Anniversary Edition*, Addison-Wesley, 1995.

[3] H. Sackman, W. J. Erikson, and E. E. Grant. "Exploratory Experimental Studies Comparing Onine and Offline Programming Performance", *Communications of the ACM*, 11:1, January 1968, pp 3-11.

[4] S. Litovsky. "Cognitive Processes in Program Comprehension", *Empirical Studies of Programmers*, Ablex Publishing Corp, 1986, pp 58-79.

[5] N.V. Flor and E.L. Hutchins. "Analyzing Distributed Cognition in Software Teams: A Case Study of Team Programming During Perfective Software Maintenance", Empirical Studies of Programmers: 4th Workshop, Ablex Publishing Corp, 1991, pp 36-64.

[6] E.W. Dijkstra. "On the role of scientific thought". *Selected writings on Computing: A Personal Perspective*. New York, NY, USA (1982) Springer-Verlag. pp. 60–66.

[7] S. Hawking, *A Brief History of Time,* Bantam Dell Publishing Group, 1988.

[8] K. Molokken and M. Jorgensen, "A review of software surveys on software effort estimation", Empirical Software Engineering, 2003. ISESE 2003, pp 223-230.

[9] Barry Boehm, *Software Engineering Economics*, Prentice Hall, 1981.

[10] M. Chemuturi, T.M. Cagley Jr., *Software Project Management: Best Practices, Tools and Techniques*. J.Ross Publishing, 2010.

[11] N.E. Fenton and S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, 2nd Edition, Course Technology, 1998.

[12] T.J. Allen, *Managing the Flow of Technology: Technology Transfer and the Dissemination of Technological Information Within the R&D Organization*, MIT Press, 1984.

[13] R.E. Grinter, J.D. Herbsleb, and D.E. Perry. "The Geography of Coordination: Dealing with Distance in R&D Work", *Proc. GROUP '99*, Phoenix, AZ, November 14-17, 1999.

[14] G.S. Gil, D.E. Perry and L.G. Votta. "A Case Study of Successful Geographically Separated Teamwork". Software Process Improvement Conference 1998 (SPI98), December 1998, Monaco.

[15] D.E. Perry and L.G. Votta. "The Tale of Two Projects – Abstract", *European Software Engineering Conference/Foundations of Software Engineering Conference 1997*, Zurich CH, September 1997.

[16] Dewayne E. Perry and Alexander L. Wolf. "Foundations for the Study of Software Architecture". *ACM SIGSOFT Software Engineering Notes*, 17:4 (October 1992).

[17] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[18] W.A. Wulf, R.L. London, and M. Shaw, "An Introduction to the Construction and Verification of Alphard", ICSE 1976.

[19] B. Curtis, S.B. Sheppard, P. Milliman, A. Borst, and T. Love. " Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics." IEEE Transactions on Software Engineering", 1994, 96-104.

[20] D.E. Perry, N. Staudenmayer and L.G. Votta, "People, Organizations, and Process Improvement", *IEEE Software*, July 1994.

[21] www.acm.org/education/curricula-recommendations - ACM Curricula Recommendations, 11 October 2013.

[22] P.G. Selinger et al. Access Path Selection in a Relational Database Management System. InACM SIGMOD, 1979.

[23] M. Didonet, D. Fabro, J. Bézivin and P. Valduriez, "Weaving Models with the Eclipse AMW Plugin", Eclipse Modeling Symposium, 2006.

[24] Z. Diskin, S. Kokaly, T. Maibaum. "Mapping-Aware Megamodeling: Design Patterns and Laws", Software Language Engineering 2013.

[25] G.M. McCue. IBM's Santa Teresa Laboratory - Architectural Design for Program Development IBM Systems Journal, Volume 17, 1978

[26] A. N. Habermann and D. E. Perry. *Ada For Experienced Programmers*. Reading, Mass: Addison-Wesley, May 1983.