"Large" Abstractions for Software Engineering

Dewayne E Perry
The Department of Electrical and Computer Engineering
The University of Texas at Austin
perry@ece.utexas.edu

ABSTRACT
Abstraction is one of the primary intellectual tools we have for managing complexity in software systems. When we think of abstractions we usually think about "small" abstractions, such as data abstraction (parameterization), type abstraction (polymorphism) and procedural or functional abstraction. These are the everyday kinds of things we work with – finding the right concepts to make the expression of our software solutions easier to understand and easier to reason about. Here I propose we think about "large" abstractions – abstractions that provide critical distinctions about our field of software engineering as a whole; abstractions that enable us to see what we do in different and important ways and provide significant improvements in how we do software engineering. I give a number of examples and delineate why I think they have been, and still are, important.

Categories and Subject Descriptors
D.2.0 [**Software Engineering**]: Software Engineering Foundations

General Terms
Design, Theory

Keywords
Abstraction-in-the-large, Foundations of Software Engineering

1. INTRODUCTION
Modularity, encapsulation and abstraction are the primary intellectual tools for managing complexity in software systems, but the greatest of these is abstraction. Finding the right, or most appropriate, abstractions is the most important part of engineering software systems: they provide understanding; they provide the right veneer over complex underlying implementations; they provide the basic vocabulary (data and operations; nouns and verbs) for each layer of our virtual machines to express the solutions to the problems to solve. They are the fundamental job of software engineering.

Indeed, abstraction is the fundamental job of software engineering research as well: finding the right abstractions for software engineers to use in their quest for the right abstractions to solve their problems; finding the right abstractions that remove more of the "accidental underbrush" of complexity [1]

I call them "large" abstractions because they address large issues in software engineering and provide large insights into the way we think about the software engineering of software systems. "Large" abstractions address foundational issues of how we do software engineering. "Small" abstractions are the grist of our daily lives building software systems. I will explore how these various "large" abstractions have affected the way we engineer software systems.

ROA'08, May 11, 2008, Leipzig, Germany.

2. Structured Programming
Structured Programming [2] is one of the earliest such "large" abstractions and one that focuses on programming: it provides precisely the right focus on what is critical in writing the simplest, understandable programs whose static structure provides us with useful clues and understanding about their dynamic structure.

As we spend a significant amount of time rediscovering the design of a system by looking at code, the issues here are critical in the life of a software system and its successful evolution. The code as presented represents a static view of the program and its presentation is a fundamental aspect contributing to its comprehensibility. The fact that this static view represents a dynamic structure compounds the need for mechanisms to ease, enhance, and promote the necessary ease of comprehensibility.

Thus, the limited use of program structures to the basic sequence, selection and iteration, while constraining, go to significant lengths towards providing a static program view that supports the needed comprehensibility of the dynamics of that program.

3. Virtual Machines
While structure programming provided a focus on the structure of code statements, virtual machines [3] provide a focus on the structure, the organization, of abstractions themselves. As commonly noted, component interfaces provide a language for the concepts provided by that component. Done well, the interface of a component provides a coherent and related set of abstractions.

The important question is how to organize a set of abstractions that we use in building a system? The "large" abstraction of a virtual machine provides the mechanism for such an organization. The basic strategy in creating a virtual machine is to build, layer by layer, an increasingly rich set of abstractions. You start by building a layer on top of the operating system or the programming language run-time system, building increasingly abstract interfaces until you have a layer with just the right concepts and abstractions to easily build you application so that it is simple to understand and simple to determine whether it is

doing what it is supposed to do. You build increasingly higher-level languages for each next layer to implement their abstractions in.

The primary utility of virtual machines is that a layer or any part of a layer can be swapped out for a different implementation. For example, the underlying operating system and the underlying machine may be swapped out by building a new but identical virtual machine on top of another operating system and machine.

4. Program and Product Families

Parnas [3] also introduced the notion of a family of programs where parts of those families are common and parts are variable. The original idea was addressed to the problem of evolution: a module evolves over time and parts of those evolved components are common with the previous ones, and parts are new – i.e., parts have been changed to accommodate new features, new improvements, or to fix faults. The family orientation is primarily a vertical one: the sequence of changes to a module and the issues of encapsulating those sets of changes.

None-the-less, this "large" abstraction set the stage for what has become product-line architecture, where the commonality is across products, not across version, and the variability is what differentiates the various products from one another. As apposed to program families where the relationship is a vertical one, the relationship here is a horizontal one.

This notion of product families has been one of the more successful drivers of modern-day reuse by means of capitalizing on the assets common across related products. In addition to the benefits of reuse, there are the benefits of reduced maintenance across products via the inherent commonality.

5. Essential versus Accidental Characteristics

In his seminal paper, "No Silver Bullet" [1], Brooks distinguishes between *essential* and *accidental* characteristics of building software systems. Essential characteristics are basic facts of life and need to be managed with the best intellectual and automated tools we have at our disposal. Accidental characteristics are those that we may, and indeed should, remedy if we can find appropriate solutions.

These accidental characteristics typically represent inadequacies or limitations that we encounter in various aspects of engineering software systems. I characterize the accidental characteristics that Brooks discusses as follows:

- Inadequate abstractions, expressions;
- Inadequate modes of expression;
- Inadequate support and resources; and
- Inadequate knowledge.

Because of these inadequacies, the complexity (our most critical essential characteristic) of our systems and the processes of building those systems is greater than it needs to be. They form the "accidental underbrush" of complexity that needs to be cleared away.

The importance of this "large" abstraction is that it focuses on a critical distinction that is useful in driving critical research in software engineering, first to find ways of removing this accidental underbrush, and second to find ways to manage the problems of the essential characteristics.

6. Problem versus Solution Space

In an invited keynote talk at ICSE '95 in Seattle [4], Jackson delineated a useful distinction between the *problem space* and the *solution space*, and proceeded to expand on the differences between them and the interesting and useful relationships between them that are incorporated into our software systems.

The problem space is in the world and the solution space is in the machine. It is all too common for software engineers to think entirely in the solution space, even when talking with customers for the software systems they are building. It is incumbent on the software engineers to fully understand the problem space, the domain of the problem and its domain abstractions and concepts. Indeed my work studying software faults shows that the easiest way to improve the quality of a system is to hire people who understand the problem domain [5].

Jackson especially draws attention to two important aspects of the problem space relative to the solution space. First, there is a distinct shape to the problem. Further the shape of the solution should reflect that shape of the problem. Second, while we often hear the design advice for a module to do one thing well, the world – that is, the problem space – is design with the fullest application of the Shanley Principle of multifaceted design.

7. Components and Connectors

Software architecture [6] is commonly considered to be composed of components (originally processing and data elements, which have been conflated) and connectors. While there are "connector atheists" who maintain there is structurally no difference between connectors and components (which we admit is true), there is an important logical difference. It is a logical rather than a physical distinction that is important.

Components represent computations and behaviors that are to be composed to create larger architectural components as well as a system itself. Connectors represent interactions between and among components which may provide the following in supporting those interactions:

- Communication (the most commonly considered purpose);
- Coordination (there is an entire community dedicated to this purpose – separating computation from coordination); and
- Mediation.

The advantage of this "large" abstraction is that of separation of concerns. There is some hope that further exploitation of this separation may enable some of the currently integral non-functional properties to be come composable – that is, that some of the nonfunctional properties may be isolated into connectors and thus be separated from the computations and behavior they mitigate.

8. Computations versus Behaviors

One of the most important insights recently is one due to Turski, made in a keynote address at FASE 2000 [7] in which he

distinguished between programs as computations and programs as behaviors. These two abstractions provide a keen and fundamental distinction and a deep understanding of different views about programs and systems, and clarify much of the confusion about differing and often conflicting views of software engineering.

Computer scientists often view programs and systems as computations and typically these computations viewed as

- Bounded, neat problems, with
- Underlying theory available, that
- Admit of clean, theoretically nice solutions.

Software Engineers on the other hand tend to view programs and systems as behaviors, which often contain computations as components of some of that behavior. These behaviors represent

- Unbounded, messy problems, where there is
- Little theory is available (and often have to make it up as we go along), and which are
- Harder to formally describe and reason about.

This large abstraction is important because it makes a critical distinction that is needed to understand the tension between computer science and software engineering and the different aspects that are critical in building and evolving software systems.

9. Theories and Models Self-Applied

It is this fact mentioned above that we have many situations where little theory available that has focused some of my recent research [9] on the problem of theories and models [8]. My concern in this research is a "large" abstraction for software engineering: defining a notion of theories and models that provides a unifying approach to software engineering as a both scientific and engineering discipline. It is the empirical aspect of science that is of most concern in my research and its relationship to software engineering.

Clearly, there are cases where our underlying theories are well developed: relational databases, the generation of lexical analyzers and parsers, etc. But what about those cases where we have little theory at all and have to make it up. How do we approach the engineering of a software system in this case. I believe that with an appropriate understand of theories and their models, we can provide a "large" abstraction that provides a unified view of our endeavor and clarifies both practice and research in software engineering.

The theories we use in the engineering of software systems are, in a variety of ways, richer that we normally think of. They may include the *hard* aspects of theory we associate with the physical sciences. They may include, and often do, the *soft* (probabilistic) aspects we associate with the behavioral sciences. Interestingly, both of these approaches are based on observations of the world. These aspects of theory change on the basis of new observations or new interpretations of observations.

But our theories also include *arbitrary* aspects that come about by design decisions (which is what makes us a science of the artificial). In this, our theories are like legal theories; they are based on decisions about the world. These theories change on the basis of new decisions or new interpretations of those decisions.

Thus, I believe it is a useful abstraction to equate theory and requirements, models and implemented systems (with out stretching Maibaum and Turski too much). This then forms an interesting and fruitful basis for unifying a number of seemingly unrelated aspects of software engineering research and practice. My work on this idea is still on-going.

10. Conclusions

There is a tendency when considering abstraction to think of *abstraction in the small*: types, objects, functions, procedures etc – i.e., getting just the right concepts for the problem and its solution. Obviously these are important. They enable us to manage complexity and evolution.

But, there are also *abstractions in the large* that help us to understand better how to think about software engineering itself as well as how to think about software systems as a whole (rather than just about their bits and pieces).

It is with *abstraction in the large* in mind, that I have presented a number of what I consider to be useful examples of them, why they are important, and how they help us at the engineering in the large level.

11. References

[1] F P Brooks. "No Silver Bullet" in *The Mythical Man Month, Anniversary Edition*, Reading Mass: Addison Wesley, 1995.

[2] O-J Dahl, E W Dijkstra and C A R Hoare. *Structured Programming,* London and New York: Academic Press, 1972.

[3] D L Parnas. "On the Design and Development of Program Families," IEEE **TSE**. SE-2(1):1-9, 1976

[4] M Jackson. "The World and the Machine". In *Proceedings of the International Conference on Software Engineering,* Seattle WA, 1995.

[5] D E Perry and C S Stieg. "Software Faults in Evolving a Large, Real-Time System: a Case Study". In *Proceedings of the 4th European Software Engineering Conference,* Garmisch Germany, 1993.

[6] D E and A L Wolf. "Foundations for the Study of Software Architure. *ACM SIFSOFT Software Engineering Notes*, 17:4 (October 1992).

[7] W M Turski. "Essay on Software Engineering at the Turn of Century". Fundamental Approaches to Software Engineering, Berlin Germany, 2000.

[8] W M Turski and T S E Maibaum. *The Specification of Computer Programs.* Reading Mass: Addison Wesley, 1987.

[9] D E Perry. "A Foundation for Empirical Software Engineering", March 2007. http://users.ece.utexas.edu/~perry/work/papers/070319-DP-fese.pdf