

Research Questions for Foundations of Complex Software Design

Mary Shaw, Carnegie Mellon University, Pittsburgh PA

The complexity and interdependence of software-intensive systems that play a central role in modern society strain our ability to create, comprehend, and control these systems. Unlike traditional products, software is intangible, hard to examine, and easy to replicate or modify. Unlike traditional products, the principal effort in creating and maintaining software arises from its design, not in its manufacture. Good engineering practice is the most promising route to improving the quality of software and software-intensive systems, but the scientific and mathematical foundations that underlie engineering practice are still immature. This paper describes the character of the design problem for modern software and identifies research questions whose answers should improve the basis for an engineering discipline of software and thereby the quality of software-intensive systems.

Design and Software

The principal problems in creating software-intensive systems are problems of design, not of construction; this is certainly the case for the software elements. Because software is an information product rather than a physical product, its manufacture requires little more than copying digital media, so in some real sense all of software development and evolution is design. However, the current usage of “design” in the software development community is narrower, usually covering only the technical plan for the structure of the software system (e.g., diagrams of the structure of the program) and the overall plan of the code for the components.

Both these views of design neglect one of the most difficult aspects of software design, namely understanding the client’s needs and translating them to a description of the software that needs to be created (or “design”ed, in current usage). Software system designs must reflect what the client really wants (beyond simple capability, e.g., how the software fits into business plans, existing systems, and corporate culture), usability issues, the economic and legal obligations that limit the possible solutions, the implications of coupling and interaction with other systems, and so on. Both these views of design neglect the need for representations of a software design that support predictions about the properties of a system that implements the design. Designers need to be able to sketch several design alternatives, predict how they will work in practice, and choose which to invest in refining.

Engineering

Engineering involves finding cost-effective solutions to complex practical problems; it relies preferentially on systematic, codified knowledge but falls back to empirical and heuristic knowledge when established theory isn't available. Engineering has a very substantial design component, and engineering design involves reconciling conflicting constraints under limitations of time and resources. Engineering includes both innovative design, in which novel solutions extend current practice, and routine design, in which solutions follow current practice. Our base of codified knowledge about software design still falls far short of what’s required for a true engineering discipline. Improving the basis for engineering practice is certainly not the only motivation for developing a science of design for software-intensive systems, but it is a fruitful generator of ideas.

Science and Theories

Science is an approach to explaining phenomena that emphasizes observation, creation of theories, prediction, evidence, improvement through confirmation and refutation, and openness to scrutiny. Science is broad enough to include descriptive, qualitative, and empirical theories as well as quantitative, structural, mathematically sound theories. A theory explains some phenomena; we value theories more when they are more concise and more general, but we accept theories that provide partial explanations as improvements over simple observations and as steps toward better theories.

Current Challenges of Software-Intensive Systems

The widespread incorporation of computing technology in everyday activities has broadened the community of users well beyond the technically-adept users of a decade or so ago. This period has seen not only the ubiquitous spread of the internet, both wired and mobile, but also increasing online business activity and personal use that places control of computing – and hence the design of computing services – in the hands of end users. This

introduces new challenges such as designing systems that can be understood, used, and managed by people who are not computing professionals, that are sufficiently robust for their intended tasks, and that observe appropriate policies on shared resources and rights in information.

Raw computing and communication power is now cheap and abundant, but it is not packaged or delivered in such a way that everyday people can *evaluate* the available resources, *select* those that meet their needs, *configure* a system with resources from diverse sources, *adapt* resources to meet their individual needs, *produce* well as consume content, and know how much to *trust* individual resources and compositions of resources.

Research Opportunities in a Science of Design for Software-Intensive Systems

A hallmark of software-intensive systems is that they do not admit of fully-proven correctness, absolute reliability, error-free operation, or other properties often assumed in computer science research. For complex systems, this kind of exact reasoning from first principles does not scale, in part because of the cost of information. This distinction between practical systems and idealized programs opens research questions such as:

How can we gain intellectual control of complex under-specified systems? How can we draw useful conclusions from partial, unreliable knowledge about software systems – and know how reliable those conclusions may be?

How can we develop theories that rely on aggregate reasoning about overall behavior rather than exact reasoning about all details? Some models of network behavior have this property; can we develop others, and can they be useful? (Compare, for example, the gas laws of physics with the N-body problem.)

How can everyday people gain control of their software, especially as the ubiquitous computing infrastructure provides ready access to information resources? How can we create non-critical systems from components of uncertain dependability, establish how dependable they should be for the task at hand, and gain appropriate confidence in whether they are sufficiently dependable?

How can we provide systematic guidance to support design decisions? Can we develop theories good enough to support the engineering practice of objectively evaluating and comparing design or implementation alternatives, choosing the most promising for further investigation, and iterating?

How can we explain, predict, and control emergent properties? Are these properties simply consequences of the design that we hadn't yet inferred or truly new consequences? Can we develop theories that improve our understanding of complex system interactions and couplings? Can we exploit system engineering for this?

How can we develop theories that consider user preferences in design? How can we elicit users' values or utilities for different levels of service at different price points? How can we recognize that different users depend in different ways on a given software-intensive system? How can we accommodate this during system development even though the user's preferences and tolerances change dynamically?

How can we develop theories that incorporate costs as well as benefits in evaluating software designs? How can design theories reflect cost of ownership, cost of gathering information to support analysis, and costs associated with uncertainty and risk?

How can we gather and share information to improve our theories? How can we represent software system designs in such a way that we can mine information from them to refine our design knowledge?

Biography

Mary Shaw is the Alan J. Perlis Professor of Computer Science, Co-Director of the Sloan Software Industry Center, and member of the Institute for Software Research International, the Computer Science Department, and the Human Computer Interaction Institute at Carnegie Mellon University. Her research interests in computer science lie primarily in the areas of programming systems and software engineering, particularly value-driven software design, software architecture, programming languages, specifications, and abstraction techniques. Dr. Shaw is an author or editor of seven books and more than one hundred forty papers and technical reports. In 1993 she received the Warnier prize for contributions to software engineering. She is a Fellow of the ACM, the IEEE, and the AAAS.

Further information is available at URL <http://www.cs.cmu.edu/~shaw/>