Position Paper: A Science of Software Design?

Michael Jackson (jacksonma@acm.org)

1. My background

I have worked in software development since 1961, with a heavy emphasis since 1966 on development method. My work has fallen into a number of phases. In 1966-1975 I worked on a data-oriented design method for sequential programs that based program structure on the structures of input and output streams. In 1975-1985 I worked on a method for analysis and design of data processing systems in which real-world entities (such as customers, accounts and so on) are viewed as processes. In 1985-1994 my work focused on the relationship between the hardware/software machine and the problem world in which it is embedded, drawing out the consequences of this relationship for descriptions made in all phases of software development. In the latter part of 13 years' work (1989-2002) with Pamela Zave at Bell Labs (latterly AT&T Research) I worked on an abstract architecture for telecommunications systems, aimed at the feature interaction problem. Since 1994 I have been working on an approach to classifying, analysing and structuring software development problems and their solutions, based on the idea of decomposition into subproblems fitting known problem frames and eventual recombination of subproblem solutions.

2. Where do we fail in software design?

Development of a science of software design—or mere recognition of its desirability—must be based on identification of specific defects attributable to its absence, on diagnosis of their causes, and on a convincing notion of how the science sought could remove, or at least mitigate, those defects. Three conspicuous related classes of defect are:

- defects of software construction causing unexpected and undesired behaviours including crashes ("we didn't build the system right");
- defects of software specification causing a mismatch between the assumptions of the hardware/software system and the properties of its environment or problem world ("we didn't build the right system"); and
- failures to address obvious concerns, including failures to ensure effective priority for critical functions ("we did something really stupid").

Defects like these are sometimes found in products of established engineering disciplines: failure of the Tacoma Narrows bridge and the Kansas City Hyatt walkway collapse are notorious. But in most classes of software development product they are ubiquitous and have come to be accepted as standard.

3. Why do we fail?

Our failures are partly attributable to social, educational and managerial causes: remedies are sought in agile methods, in licensing of software engineers, in technology transfer, or in purely managerial approaches to software development such as CMM. But a more immediate cause, and one more directly addressable by development of a software design science, is the complexity of the software we attempt to build.

Some of this complexity arises in individual features considered separately. A central-locking system for a car must deal with four doors, each with an exterior handle, an interior handle and a button, one or two of the doors also having an external key, a tailgate with an external key and an internal release lever, and an interlock with the ignition control. Potentially, this environment has a very large number of states and transitions, and this complexity must be related to the several requirements—child safety, guarding against carjackers, preventing locking of the car while the key is inside, convenience of use in shopping, preventing theft of the car or its contents, and accessibility in the event of a crash.

Even when individual features are relatively simple their interactions are likely to give rise to huge complexity in the whole system. The interactions are potentially found everywhere in the development. The customer or end-user *requirements* of different features may conflict, or they may interact in a way that makes them impossible to consider separately and very hard to understand in combination. The *problem world* for a significant system is likely to be heterogeneous, including human beings (for example, operators and users), inert physical structures (for example, the track layout of a railway), mechanical devices (for example, lift cars, doors, and winding gear), actuators and sensors (for example, motor relays), and physical realisations of lexical structures (for example, swipe cards). The *software specifications* must combine

satisfaction of all the requirements, relying on different and possibly conflicting assumptions about problem world properties. The *software structure* must implement some approximation to the conjunction of the software specifications, while ensuring that the most critical functionality is the most reliable.

4. Normal and radical design

In established branches of engineering much—or most—design is *normal*. A product of a standard form is designed to meet a standard need: design choices are largely restricted by experience. Normal design is typical of specialised engineering branches, such as automobile engineering, with a substantial history of design experience. Normal design of complex products, such as automobiles, deals effectively with feature combinations by staying within, or very close to, the bounds of what is already known. Different feature combinations define different products, each with its own specialised branch of design knowledge—product-specific and relatively well understood. Fixed combinations allow specialised interfaces to be built into the components at every level from requirements to physical realisation.

Radical design, by contrast, delivers new product forms—if only by new combinations of old parts—to meet new demands. The designer tackles new problems and devises new solutions whose properties are not known in advance. Much—perhaps most—software development involves radical rather than normal design. This is a mark of our immaturity, lack of specialisation, and inability to learn from our mistakes. But it is also a consequence of the versatility of computers (unparallelled by any other product). An unprecedented combination of features never seems impossible; we readily merge the functionality of barely understood disparate products into one new product that is even less understandable; and we welcome without complaint our customers' apparent preference for complex and richly interoperating functionality over simplicity and reliability.

5. A science of software design

A science of software design, then, must be about radical, not about normal, design. Its chief goals are to explicate the properties of decompositions and combinations, in requirements, specifications, assumptions about problem worlds, and software components, and to support the tasks of choosing and making appropriate decompositions and combinations.

Such a science has both formal and informal aspects. Some combinations of requirements or of specifications can be regarded as problems in concurrency writ large: they may yield to already known formal results in concurrent programming, or to suitable extensions or adaptations. Combining software components is certainly a formal task, if only because software is primarily a formal domain. On the other hand, the environment or problem world for most software developments is essentially physical, and can not be treated by purely formal techniques. The design of human-computer interfaces, whether for single simple features or for complex combinations of features, is concerned with human ergonomics and psychology, and certainly will not yield to formal calculational methods.

Application of a design science, like the practice of mathematics itself, has an overarching informality. The formal parts—theorems, models, programs, equations, formulae—are embedded in an informal intellectual structure that motivates them and gives them meaning. A science of design must be at root a discipline of devising, understanding, populating and exploiting this informal structure, and of applying, within it, established formal knowledge (such as mathematical theorems) relevant to the formal parts.