Elements of a Design Language for Software Architecture

Mary Shaw
Carnegie Mellon University
Pittsburgh PA

Position Paper for IEEE Design Automation Workshop
January 1990

This work is sponsored by the United States Department of Defense.

Like Hardware, Software has Several Design Levels

Hardware design is carried out at several different levels, with different issues, models, and design strategies at each level. Thus circuits, registers, instruction sets, and architectures are designed in terms of different components combined with different operators; the analysis techniques (indeed, what is analyzed), notations, and appropriate tools are correspondingly different.

The same situation is true for software, except that the distinctions among levels — or even the existence of distinct levels — is not as widely recognized. For software we can identify at least the following levels and the issues that dominate at each level:

User Model	Problem descriptions in users' terms
Requirements	Abstract properties in software terms
Architecture	Global allocation of function, data/communication connectivity
Programming	Algorithms and data structures
Machine language	Memory layout, storage (& other resource) allocation

A delivered software system should consist of a model for each of these levels, each expressed in some appropriate language. (It is not reasonable to expect that a single language should serve all levels, so questions about whether some programming language is suitable for describing architectures should be addressed by asking first what needs to be expressed about architectures and then what the programming language is good at expressing.)

Major Opportunities for Software Exist at the Architecture Level

A common way to portray programming has been as successive refinements of an elementary specification. This can be depicted as a tree in which the relation between a node and its descendants is "node is-composed-of subnodes." Such portrayals usually assume that the same notations are used throughout the decomposition and that, as a result, the transition from the root of the "is-composed-of" tree to the basic statements or subroutines at the leaves is continuous — a "small matter of programming."

Much of the recent work on software reuse has focussed on the leaves of the tree — that is, the subroutines that can be collected into libraries — and on generalizations such as generic definitions. Much less attention has been directed at the common patterns of internal nodes — that is, structures such as synchronization strategies and the client/server relation between two modules — that provide organization but not ultimate detail for the code at the leaves.

Unfortunately, for large systems this picture leaves a great deal to be desired. Progress from high-level design to code (that is, the descent through the "is-composed-of" tree) does not proceed smoothly or in a single language. Only in simple examples is the system requirement a specification of a function call that can be written directly in code. More realistic systems involve a decomposition into modules, assignment of high-level function to modules, and design of the data, communication, and control relations among those modules. Analysis at this level is more concerned with system balance, bandwidth, throughput, and localization of design decisions than it is with input/output relations for specific functions. As a result, there are discontinuities in the tree; as the designer moves from level to level, the kinds of nodes and the ways of combining them change.

Software developers have accumulated enough experience to provide guidance about useful system-level structures. Although this understanding is largely informal, it provides a basis for systematization and eventual formalization.

A Design Language can be Based on Constructs now Used Informally

Papers describing software systems now often dedicate a section to the architecture of the system. This section often contains a diagram with boxes depicting major components and lines depicting some communication or control relation among the components. The boxes usually have labels that are highly specific to the particular system: "lexical analyzer," "alias table," "requisition slip data file". Connections are similarly specific: "identifiers," "update requests," "inventory levels."

Examination of these descriptions shows that if the specific functionality of the elements is set aside, the remaining structural properties often fall into identifiable classes. For example, here are some of the classes of components that appear regularly in architectural descriptions:

(Pure) computation	Simple input/output relations, no retained state
Memory	Shared collection of persistent structured data
Manager	State and closely related operations
Controller	Governs time sequences of others' events
Link	Transmits information between entities
Command drive	Discrete, usually local, syntax-intensive manipulation of state

These components are combined in a variety of ways. Some of the most common are:

Procedure call

Data flow

Implicit triggering

Message passing

Shared data

Instantiation

Code mingling

Common system patterns built out of these components are discussed in my position paper for the Fifth International Workshop on Software Specification and Design [Shaw89].

These identifications of architectural components and techniques for combining them into subsystems and systems provide the basis for designing a language for defining software architectures. This language would support not only simple expressions defining connections among simple modules but also subsystems, configuration of subsystems into systems, and common paradigms for such combinations.

Engineering Advice on Selection of Implementations is Also Required

A notation for composing systems out of subsystems also supports top-down design by supporting the definition of systems in terms of their constituent subsystems. As is usual in design, many implementation alternatives are available for each decomposition step. Support for the selection of the most appropriate alternative for a particular design is also important.

For example, Lane [Lane90] has shown how to provide design support for the user interface component of a software system. He defined explicit design spaces for the functional requirements and the architectural alternatives and identified a set of rules that map a functional requirement to appropriate architectures for that requirement.

References

[Lane90] Thomas G. Lane. "User Interface Software Structures". Carnegie Mellon Computer Science Technical Report, in progress.

[Shaw89] Mary Shaw. "Larger Scale Systems Require Higher Level Abstractions". *Proc. Fifth International Workshop on Software Specification and Design, Software Engineering Notes* Vol. 14 No. 3, May 1989, PP.143-146.