Heterogeneous Design Idioms for Software Architecture

Mary Shaw
School of Computer Science
and Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Software designers use a variety of structural patterns to specify system architectures. These patterns, or idioms, are currently used informally and imprecisely. Nevertheless, they provide a useful, broadly shared vocabulary. In practice, a given design often relies on several patterns. This paper reviews some common architectural idioms, shows several ways in which they are used heterogeneously, and discusses the benefits of making these idioms and their combinations more explicit and precise.

I. Introduction

Software system design must deal with architectural issues as well as algorithms and representations. Architectural issues include gross decomposition of function, assignment of function to design elements, composition of design elements, scaling and performance, and selection among design alternatives.

Architectural structures are often described in terms of idiomatic patterns that have emerged informally over time. Typical descriptions of software architectures begin with synopses such as:

"Camelot is based on the client-server model and uses remote procedure calls both locally and remotely to provide communication among applications and servers." [Spector 87]

"Abstraction layering and system decomposition provide the appearance of system uniformity to clients, yet allow Helix to accommodate a diversity of autonomous devices. The architecture encourages a client-server model for the structuring of applications." [Fridrich 85]

"We have chosen a distributed, object-oriented approach to managing information." [Linton 87]

"The easiest way to make the canonical sequential compiler into a concurrent compiler is to pipeline the execution of the compiler phases over a number of processors. ... A more effective way [is to] split the source code into many segments, which are concurrently processed through the various phases of compilation [by multiple compiler processes] before a final, merging pass recombines the object code into a single program." [Seshadri 88]

"The ARC network [follows] the general network architecture specified by the ISO in the Open Systems Interconnection Reference Model. It consists of physical and data layers, a network layer, and transport, session, and presentation layers." [Paulk 85]

Study of architectural descriptions reveal a number of patterns that recur regularly [Shaw 89]. A few of these idioms (e.g., objects) have been carefully refined, but others are still used quite informally, even unconsciously. Nevertheless, the idioms are widely recognized. System designs often appeal to several of these idioms, combining them in various ways. Inspection of descriptions of actual systems shows that the motivations for using different idioms are often not carefully separated, and the interactions of the idioms are correspondingly obscure.

This discussion begins by reviewing some of the common architectural idioms. It then examines three examples to show how these idioms are used and combined. The first two examples show a kind of top-down design with different idioms used at discrete steps of the expansion; the third shows a global reinterpretation of the design as it proceeds. The paper closes with a discussion of the significance of heterogeneous design and the vocabulary presented by the idioms

Much of computer science includes phenomena about which we do not yet have well-established scientific truths but do have interesting observations and generalizations. Brooks has proposed that we recognize three kinds of results: *findings* (well-established scientific truths), *observations* (reports on actual phenomena), and *rules-of-thumb* (generalizations, signed by an author but perhaps not fully supported by data). He suggests appropriate criteria for quality: truthfulness and rigor for *findings*, interestingness for *observations*, and usefulness for *rules-of-thumb*, and freshness for all three [Brooks 88]. Brooks proposed these distinctions for the human-computer interface community, but his point is equally valid for specification and design, where observations and rules of thumb will help us learn *what* must be specified as well as how to specify it. In the spirit of this distinction, this paper presents observations and some rules-of-thumb, preliminary to more rigorous findings.

II. Common Architectural Constructs

Over the past thirty years, abstraction techniques such as high-level programming languages and abstract data types have improved our ability to specify and develop software. Just as these programming abstractions emerged because good programmers recognized useful structures, good system designers now recognize useful system organizations. Previous work has identified several system organizations used at the architectural level of software design [Shaw 89] and argued that these constructs are amenable to the application of language design techniques [Shaw 90]. This paper shows how well-defined abstractions for system constructs can help improve system design.

The common system organizations identified earlier provide a vocabulary of design idioms that are broadly used by system designers. Several of these appear in the case studies of the next section. The structures that play a central role are reviewed here (there have been some generalizations and changes in labelling since [Shaw 89]).

Pipeline: A series of modules is connected by data flow pipelines. Each module accepts a stream of inputs and emits a stream of outputs; usually the output stream involves a local transformation of (or selection from) the input stream. Modules are composed by connecting the output stream of one module to the input stream of another.

Data abstraction or object-oriented organization: A representation and its associated primitive operations are encapsulated in an abstract data type or object. The entity is responsible for preserving the integrity of the representation.

Layered hierarchy: A system is organized hierarchically, with each layer providing service to the layer above it and serving as a client to the layer below. Inner layers are usually hidden from all except the adjacent outer layer, except for certain functions carefully selected for export.

Table-driven interpreters A virtual machine is produced in software. Four components can usually be found: an interpretation engine to do the work, a memory that contains the pseudo-code to be interpreted, a representation of the control state of the interpretation engine, and a representation of the current state of the program being simulated.

Repositories: A central data structure represents the current state; a collection of independent processes operate on the central data store. The choice of control discipline leads to major subcategories: if the types of transactions in an input stream of transactions trigger selection of processes to execute, the repository can be a traditional database; if the current state of the central data structure is the main trigger of selecting processes to execute, the repository can be a blackboard as in the third case study below; preset execution order yields a modern compiler.

III. Case Studies: Heterogeneous Designs

This section explores the use of multiple idioms in a design through three case studies. The first is a commercial product for process control in the chemical industry; in this system one idiom (hierarchy) is used for the top level of the design and different idioms (data abstraction and repositories) are used to elaborate the layers. The second example comes from an artificial intelligence review study; one idiom (interpreter, more complex than in the first study) is used at the top level and different idioms (interpreter, pipeline) are used to elaborate the elements, but additional structure is added to the high-level design in the process. The final example is also drawn from artificial intelligence; in this case the entire high-level design (repository) is reinterpreted (as interpreter) as the second level is developed—that is, the elaboration of the design involves a shift in viewpoint rather than independent elaboration of components. All three system designs are based on the idioms reviewed in Section II. They were selected to show different ways to combine the idioms. The system descriptions draw very heavily on the cited sources; the interpretations of the architectures are my own.

III.1 Case 1: A Layered Design with Different Idioms for the Layers

The PROVOX[®] system by Fisher Controls offers distributed process control for chemical production processes [Fisher 89]. Process control capabilities range from simple control loops that control pressure, flow, or levels to complex strategies involving interrelated control loops. Provisions are made for integration with plant management and information systems in support of computer integrated manufacturing. The system architecture integrates process control with plant management and information systems in a 5-level layered hierarchy. Figure 1 shows this hierarchy; the right side is the software view, and the left side is the hardware view. Each level corresponds to a different process management function with its own decision-support requirements.

Level 1: Process measurement and control: direct adjustment of final control elements.

Level 2: Process supervision: operations console for monitoring and controlling Level 1.

Level 3: Process management: computer-based plant automation, including management reports, optimization strategies, and guidance to operations console.

Levels 4 and 5: Plant and corporate management functions: higher-level functions such as cost accounting, inventory control, and order processing/scheduling.

Different kinds of computation and response times are required at different levels of this hierarchy. Accordingly, different computational models are used. Levels 1

to 3 are object-oriented; Levels 4 and 5 are largely based on conventional data-processing repository models. For present purposes it suffices to examine the object-oriented model of Level 2 and the repositories of Levels 4 and 5.

Figure 1: PROVOX: Hierarchical Top Level

For the control and monitoring functions of Level 2, PROVOX uses a set of *points*, or loci of process control. Figure 2 shows the canonical form of a point definition; seven specialized forms support the most common kinds of control. Points are, in essence, object-oriented design elements that encapsulate information about control points of the process. The points are individually configured to achieve the desired control strategy. Data associated with a point includes:

Operating parameters, including current process value, setpoint (target value), valve output, and mode (automatic or manual).

Tuning parameters, such as gain, reset, derivative, and alarm trip-points.

Configuration parameters, including tag (name) and I/O channels.

Figure 2: PROVOX: Object-Oriented Elaboration

In addition, the point's data can include a template for a control strategy. Like any good object, a point also includes procedural definitions such as control algorithms, communication connections, reporting services, and trace facilities. A collection of points implements the desired process control strategy through the communication services and through the actual dynamics of the process (e.g., if one point increases flow into a tank, the current value of a point that senses tank level will reflect this change). Although the communication through process state deviates from the usual procedural or message-based control of objects, points are conceptually very like objects in their encapsulation of essential state and action information.

Reports from points appear as input transactions to data collection and analysis processes at higher design levels. The organization of the points into control processes can be defined by the designer to match the process control strategy. These can be further aggregated into *Plant Process Areas* (points related to a set of equipment such as a cooling tower) and thence into *Plant Management Areas* (segments of a plant that would be controlled by single operators).

PROVOX makes provisions for integration with plant management and business systems at Levels 4 and 5. Selection of those systems is often independent of process control design; PROVVOX does not itself provide MIS systems directly but does provide for integrating a conventional host computer with conventional database management. The data collection facilities of Level 3, the reporting facilities of Level 2, and the network that supports distributed implementation suffice to provide process information as transactions to these databases. Such databases are commonly designed as repositories, with transaction processing functions supporting a central data store -- quite a different idiom from the object-oriented design of Level 2.

The use of hierarchical layers at the top level of a system is fairly common. Other examples are the ISO Open Systems Interconnection Reference Model and the kernel or shell model of operating systems and programming environments. This permits strong separation of different classes of function and clean interfaces between the layers. However, within each layer the interactions among components are often too intricate to permit strict layering.

III.2 Case 2: An Interpreter Using Different Idioms for the Components

Rule-based systems provide a means of codifying the problem-solving know-how of human experts. These experts tend to capture problem-solving techniques as sets of situation-action rules whose execution or activation is sequenced in response to the conditions of the computation rather than by a predetermined scheme. Since these rules are not directly executable by available computers, systems for interpreting such rules must be

provided. Hayes-Roth surveyed the architecture and operation of rule-based systems [Hayes-Roth 85].

Figure 3: Basic Rule-Based System

The basic features of a rule-based system, shown in Hayes-Roth's rendering as Figure 3, are essentially the features of a table-driven interpreter:

The *pseudo-code* to be executed, in this case the knowledge base

The *interpretation engine*, in this case the rule interpreter, the heart of the inference engine

The *control state of the interpretation engine*, in this case the rule and data element selector

The *current state of the program* running on the virtual machine, in this case the working memory.

Interpreters of this sort are commonly used to build virtual machines that close the gap between the computing engine expected by the semantics of the program and the computing engine available in hardware. We occasionally speak of a programming language as providing, say, a "virtual Pascal machine."

Figure 4: Sophisticated Rule-Based System

Rule-based systems make heavy use of pattern matching and context (currently relevant rules). Adding special mechanisms for these facilities to the design leads to the more complicated view shown in Figure 4. In adding this complexity, the original simple interpreter vanishes in a sea of new interactions and data flows. Although the interfaces among the original modules remain, they are not distinguished from the newly-added interfaces.

Figure 5: Simplification of Sophisticated System

However, the interpreter model can be rediscovered by identifying the components of Figure 4 with their design antecedents in Figure 3. This is done in Figure 5. Viewed in this way, the elaboration of the design becomes much easier to explain and understand. For example, we see that:

The knowledge base remains a relatively simple memory structure, merely gaining substructure to distinguish active from inactive contents.

The rule interpreter is expanded with the interpreter idiom (that is, the interpretation engine of the rule-based system is itself implemented as a table-driven interpreter), with control procedures playing the role of the pseudo-code to be executed and the execution stack the role of the current program state.

"Rule and data element selection" is implemented primarily as a pipeline that progressively transforms active rules and facts to prioritized activations; in this pipeline the third filter ("nominators") also uses a fixed database of metarules.

Working memory is not further elaborated.

The interfaces among the rediscovered components are unchanged from the simple model except for the two bold lines over which the interpreter controls activations.

This example illustrates two points. First, in a sophisticated rule-based system the elements of the simple rule-based system are elaborated in response to execution characteristics of the particular class of languages being interpreted. If the design is presented in this way, the original concept is retained to guide understanding and later maintenance. Second, as the design is elaborated, different components of the simple model can be elaborated with different idioms.

Note that the rule-based model is itself a design structure: it calls for a set of rules whose control relations are determined during execution by the state of the computation. A rule-based system provides a virtual machine—a rule executor—to support this model.

III.3 Case 3: A Blackboard Globally Recast as Interpreter

The blackboard model of problem solving is a highly structured special case of opportunistic problem solving. In this model, the solution space is organized into several application-dependent hierarchies and the domain knowledge is partitioned into independent modules of knowledge that operate on knowledge within and between levels. Nii surveyed models and implementations of blackboard systems [Nii 86].

The blackboard model is usually presented with three major components:

The knowledge sources: separate and independent parcels of application-dependent knowledge. Interaction among knowledge sources takes place solely through the blackboard.

The blackboard data structure: problem-solving state data, organized into an application-dependent hierarchy. Knowledge sources make changes to the blackboard that lead incrementally to a solution to the problem.

Control: driven entirely by state of blackboard. Knowledge sources respond opportunistically when changes in the blackboard make them applicable.

Figure 6 shows the basic architecture of a blackboard system. Note the absence of a control component. Invocation of a knowledge source is triggered by the state of the blackboard. The actual locus of control, and hence its implementation, can be in the knowledge sources, the blackboard, a separate module, or some combination of these. This is a particular example of the repository system organization, in which a central data store is manipulated by independent processes.

The first major blackboard system was the HEARSAY-II speech recognition system. Nii's schematic of the HEARSAY-II architecture appears as Figure 7. The blackboard structure is a six- to eight-level hierarchy in which each level abstracts information

on its adjacent lower level and blackboard elements represent hypotheses about the interpretation of an utterance. Knowledge sources correspond to such tasks as segmenting the raw signal, identifying phonemes, generating word candidates, hypothesizing syntactic segments, and proposing semantic interpretations. Each knowledge source is organized as a condition part that specifies when it is applicable and an action part that processes relevant blackboard elements and generates new ones. The control component is realized as a blackboard monitor and a scheduler; the scheduler monitors the blackboard and calculates priorities for applying the knowledge sources to various elements on the blackboard.

Figure 6: Blackboard Model

Figure 7: Hearsay architecture

HEARSAY-II was implemented between 1971 and 1976 on DEC PDP-10s; these machines were not directly capable of condition-triggered control, so it should not be surprising to find that an implementation

provides the mechanisms of a virtual machine that realizes the implicit invocation (opportunistic) semantics required by the blackboard model.

Figure 7 not only elaborates the individual components of Figure 6; it also adds components for the previously-implicit control component. In the process, the figure becomes rather complex. This complexity arises because it is now illustrating two concepts: the blackboard model and realization of that model by a virtual machine. The blackboard model can be recovered as in Figure 8 by suppressing the control mechanism and regrouping the conditions and actions into knowledge sources. The virtual machine can be seen to be realized by an interpreter using the assignment of function in Figure 9. Here the blackboard cleanly corresponds to the current state of the recognition task. The collection of knowledge sources roughly supplies the pseudocode of the interpreter; however, the actions also contribute to the interpretation engine. The interpretation engine includes several components that appear explicitly in Figure 7: the blackboard monitor, the focus of control database, and the scheduler, but also the actions of the knowledge sources. The scheduling queue corresponds roughly to the control state. To the extent that execution of conditions determines priorities, the conditions contribute to rule selection as well as forming pseudocode.

Figure 8: Blackboard view of Hearsay-II

Here we see a system initially designed with one model (blackboard, a special form of repository), then realized through a different model (interpreter). The realization is not a component-by-component expansion as in the previous two examples; the view as an interpreter is a different aggregation of components from the view as blackboard.

Figure 9: Interpreter view of Hearsay-II

IV. Discussion

The authors of the material reviewed in the case studies were not consciously applying a systematic body of knowledge and vocabulary about software architectures. They, like the authors of the descriptions quoted in the introduction and many similar descriptions, were conveying their intentions using the best terms available. Indeed, the examples show common organizations being used without explicit vocabulary. Nevertheless, some consensus on useful design patterns can be found in the literature.

A system specification is a communication from the designer not only to the development environment but also to other developers, subsequent maintainers, and other observers. It is to everyone's advantage for the designer's intent to be as clear as possible. A list of modules and their interfaces may suffice to determine work packages for implementation. However, the designer's intentions also include the kinds of responsibility appropriately assigned to each module and the protocols and representations of interaction. If the overall system pattern and the roles of the various modules in that pattern are not included in the design, design modifications, especially by later maintainers, will be made in ignorance of the original intentions and design integrity will deteriorate. In other areas of software specification and development we have developed a more-or-less standard vocabulary of basic constructs and generally useful patterns (idioms) for using them in common situations. The examples above show how such a vocabulary can help support system-level design.

IV.1 Historical Precedents

In the shift from assembly language to higher-level languages (HLLs), common patterns of machine instructions were recognized and ratified in the HLL constructs. The critical abstraction step in this shift was the realization that a relatively small number of common patterns of machine language statements were adequate for the majority of the programs being written and further that using the HLL constructs conveyed the programmer's intentions as well as specifying an execution.

Type systems were originally added to programming languages to provide hints to compilers about the appropriate machine instructions to use for overloaded arithmetic operations. Later it became clear that types could express a programmer's intentions about the intended use of a variable. This was useful for checking ("a type system is a Neanderthal verifier" -- Jim Morris) and for restricting the use of operations which, although legal, were not consistent with intended use.

Over the past 20-25 years we have developed a shared vocabulary of common data types. The name of a type (e.g., "stack," "priority queue," "binary tree") evokes some common semantic core and a number of implementation alternatives. Even now, however, individuals' understanding of the semantics differ in details that were often not revealed at all before formal specification of the types was possible.

Differences in details notwithstanding, we have achieved a degree of shared understanding of data types that has simplified communication among designers and developers. For architectural constructs, even an informal shared vocabulary that makes common idioms explicit would be a significant contribution to our system design capabilities.

IV.2 Implications for Specification and Design

A considerable body of informal understanding about software architectures has accumulated, but it is neither systematically organized, uniformly labelled, nor effectively transmitted. As a result, the architectural design level is often treated as a simple scaling-up of the algorithm/data structure design level. The resulting lack of understanding of design implications often leads to ad hoc design or to design by default - replicating prior architectures that may or may not be appropriate in their new uses.

Good models for architectures in common use

It is now common for a presentation on a software system to devote part of the introduction to the architecture of the system. This often includes a diagram with boxes depicting major system 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 [Shaw 89]. Moreover, the connections between elements include not only procedure calls and unix pipes, but a variety of other invocation techniques and interaction protocols. This appears to provide a good base for language design [Shaw 90].

In the second and third examples above, the design is elaborated in place via case studies a series of progressively more complex diagrams. More explicit attention to architectural design would improve this, of course. Equally important, we need to understand better the relations between design stages or views as well as the relations among elements in a single view of the design. The first example above (PROVOX) expanded elements of the hierarchical view using one idiom or another. The second example (rule-based system) obtained the sophisticated case by adding mechanism to the simple case. The third example (blackboards) performed a global reinterpretation of the design.

Tools to support architectural design are also needed. Current tools include module interconnection languages, but these deal only in plain-vanilla modules—they are not able to take advantage of special properties of modules or to perform the analog of type-checking on the use of a design pattern, and they are usually restricted to one or two interconnection techniques. Further, they deal only with point-to-point connections (e.g., of procedure call to definition) and hence do not support direct use of more complex architectural idioms.

Criteria for choosing among architectures and implementation

The selection of an architecture should depend in part on the computing environment and in part on the needs of the application domain. Lane explored this problem in one domain (user interfaces) [Lane 90]. Similarly, the selection of an implementation for an architectural component should depend on the system-level requirements. For example, if you see that a design contains a module with the function of a memory (you put information in, you get information out), what questions do you ask and how should you interpret the answers to decide whether to use a file system, a database, a hypertext system, a directory, a symbol table, a persistent array, a specially-tailored data structure, or something else? Then, having decided what general class of memory you need, what questions do you ask next to decide which variant will work best?

V. Acknowledgements

This work is sponsored by the US Department of Defense, in part by Contract F19628-90-C0003 with the

US Air Force as the executive contracting agent, and in part by the Defense Advanced Research Projects Agency (DoD) ARPA Order No. 4976, Amendment 20, under Contract F33615-87-C-1499, monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, Ohio 45433-6543.

I wish to thank the Association for Computing Machinery for granting me permission to reprint Figures 3 and 5 from Frederick Hayes-Roth's article titled "Rule-Based Systems" which appeared in *Communications of the ACM*, vol 28, no 9, pp 924 and 928 and appear in this paper as Figures 3, 4, and 5. I would also like to thank the American Association for Artificial Intelligence for granting me permission to reprint Figures 1 and 7 from Penny H. Nii's article titled "Blackboard Systems" which appeared in two issues of *AI Magazine*, vol 7, no 3, Copyright © Summer 1986, pp 39 and vol 7, no 4, Copyright © August 1986, pp 89 with all rights reserved and appear in this paper as Figures 6 and 7.

VI. References

- Brooks 88**
Frederick P. Brooks, Jr. Grasping Reality Through Illusion -- Interactive Graphics Serving Science. *Proceedings of the ACM SIGCHI Human Factors in Computer Systems Conference*, May 1988, pp. 1-11.
- Fisher 89**
Fisher Controls International. *PROVOX®plus Instrumentation System: System Overview*. Fisher Controls International, 1989.
- Fridrich 85**
Marek Fridrich and William Older. Helix: The Architecture of the XMS Distributed File System. *IEEE Software*, vol 2, no 3, May 1985 (pp.21-29).
- Hayes-Roth 85**
Frederick Hayes-Roth. Rule-Based Systems. *Communications of the ACM*, vol 28, no 9, September 1985, pp.921-932.
- Lane 90**
Thomas G. Lane. *User Interface Software Structures*. PhD Thesis, Carnegie Mellon University, Carnegie Mellon University Computer Science Technical Report CMU-CS-90-101, June 1987.
- Linton 87**
Mark A. Linton. Distributed Management of a Software Database. *IEEE Software*, vol 4 no 6, November 1987.
- Nii 86**
H. Penny Nii. Blackboard Systems (two parts). *AI Magazine*, vol 7 nos 3 (pp.38-53) and 4 (pp.82-107).
- Paulk 85**
Mark C. Paulk. The ARC Network: A Case Study. *IEEE Software*, vol 2 no 3, May 1985, pp.62-69.
- Seshadri 88**
V. Seshadri et al. Semantic Analysis in a Concurrent Compiler. *Proceedings of ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*.
- Shaw 89**
Mary Shaw. Larger-Scale Systems Require Higher-Level Abstractions. *Proceedings of the Fifth International Workshop on Software Specification and Design*, IEEE Computer Society, 1989, pp.143-146.
- Shaw 90**
Mary Shaw. Elements of a Design Language for Software Architecture. Unpublished position paper, 1990.
- Spector 87**
Alfred Z. Spector et al. Camelot: A Distributed Transaction Facility for Mach and the Internet - An Interim Report. Carnegie Mellon University Computer Science Technical Report CMU-CS-87-129, June 1987.