Problem Architectures

A Position Paper for the ICSE-17 Workshop
on Architectures for Software Systems

Michael Jackson

14 November 1994

1 Problems and Solutions

The activity of software development is an activity of *building machines* to solve problems. We construct a machine, in the form of a specialisation by software of a general-purpose computer; and we install it in the world to serve a purpose—just as an architect constructs a building and installs it in the world to serve a purpose. The analogy is natural and direct. But the word *architecture* connotes a concern with *solutions* rather than *problems.* Schools of architecture sometimes refer to their subject as the *built environment*, a term entirely appropriate both to architecture and to the analogous concept in software. But in one way it is a narrow term: it focuses our attention on the artifact, not on the problem it is intended to solve.

Some architects, especially Alexander [1], have paid explicit attention to the problems they are solving, recognising that a building can not be conceived and designed, or even criticised, without explicit analysis of the purpose to which it will be put. In software development we should do no less. But, as some workers in the area of software development patterns [11, 4, 10] have recognised, we have too few intellectual tools for examining, analysing, or classifying *problems.*

This shortcoming must be remedied if the field of software architecture is to reach its fullest fruition. The virtues of an architecture are relative to the problems for which it might be used. Comparison, analysis and evaluation of architectures must relate them [19] to problems. But the obstacles to focusing on problems are real. The traditional difficulties of separating the **what** from the **how**, and of avoiding implementation bias in specifications [12], show how easily one can slip from stating the problem into postulating a solution.

The study of problems is of interest in its own right, but it also has very practical consequences for software development. In software, above all, where the medium and product of work is description, effort devoted to problem description and analysis is doubly repaid. Not only can design and construction—as in any work of producing artifacts to solve problems—proceed far more confidently and purposefully when the problem has been explicitly described. In software, many parts of the problem description often can—and should—be transformed into directly useful parts of the solution.

2 Problem Frames

Polya's well-known book *How To Solve It* [18] expounds and amplifies the ideas of the ancient Greek mathematicians on problem-solving method. The central idea is that each problem has *principal parts* and a *solution task*, according to which it can be classified. The Greeks recognised two problem classes:

problems to find or construct The principal parts of a problem to find or construct are an *unknown*, the *data*, and the *condition*. The *solution task* is to find the *unknown*, correctly related to the *data* by the *condition*. For example: Construct a triangle whose sides are of lengths 6, 7, and 8. The *unknown* is the triangle; the *data* is the lengths 6, 7, and 8; the *condition* is that the triangle's sides are of those lengths; and the *solution task* is to find the triangle.

problems to prove The principal parts of a problem to prove are the *hypothesis* and the *conclusion*. The *solution task* is to prove that the *conclusion* follows from the *hypothesis*. For example: Prove that if the sides of a quadrilateral are equal its diagonals bisect each other. The *hypothesis* is that the quadrilateral's sides are equal; the *conclusion* is that its diagonals bisect each other; and the *solution task* is to show that the latter follows from the former.

These structures, of principal parts and solution task, may be called *problem frames*. They can be thought of as templates against which each new given problem may be checked. If a template fits the problem, then the methods and techniques appropriate to the frame can be used. Polya discusses several heuristics for solving problems of the two classes. For example: Break the *condition* into parts; Change the *unknown* to bring it closer to the *data*. Each heuristic is expressed in terms of the principal parts of the problem frame. This allows Polya to discuss problems in terms of their parts and structure without slipping into a discussion of putative solutions.

3 Problem Contexts and Domains

The principal parts of a problem frame are furnished by the *problem context* and the relationships among parts of the context. The contexts of mathematical problems are populated by abstract mathematical objects such as triangles and quadrilaterals. Their properties are assumed to be well-known to the problem solver, at least to the extent necessary for solving the problem.

But the context of a software development problem is located in the natural and human world. There are employees, and aeroplanes, and tax laws, and lifts, and bank accounts. It is not, in general, to be assumed that their properties are well-known. On the contrary, they vary—widely and sometimes arbitrarily—from one problem to another. So the context of each particular software development problem must be made the subject of a particular ad hoc study. This study is not an optional or ancillary part of the development work, but a central and integral part of what developers must do.

It is natural to structure a problem context as a number of *domains*. A *domain* in this sense is not a general category of applications, such as the banking domain or the process-control domain. Rather, it is a specific part of the particular problem's context that can be studied and described in comparative isolation. For example, in a simple system to provide management information about the progress of a chemical process in a plant, it may be appropriate to consider the managers, the plant, and the information to be produced as three separate domains. Each domain has its own *phenomena*, and its own properties that may be regarded as relationships among those phenomena [9].

In the simple information system the domains do not interact with each other in the absence of the machine to be built. It is—as always—the purpose of the *machine* to bring the domains into the required relationships by interacting with all of them. This interaction can be viewed as interaction by shared phenomena [20]. When a manager requests a report or an answer to a question, the request is an event shared by the manager and the machine; when a vessel in the plant is at a certain temperature, that is a state shared by the machine and the plant; when a graph is displayed in response to a manager's request, the events and structures involved in the display are shared by the machine and the domain of information outputs.

4 Software Problem Frames

The chemical process information system fits into a simple problem frame:

simple IS frame The principal parts of a simple IS problem are a *real world* domain, a domain of *information requests*, a domain of *information outputs*, the *system*, and the *information function*. The *solution task* is to construct the *system* so that it produces the *information outputs* in response to the *information requests*; the *information outputs* must contain the requested information about the *real world*, as specified by the *information function*. In our example, the *real world* is the chemical plant, and the *information requests* are made by the managers.

A problem frame excludes many problems. The **simple IS frame** treats the *real world* as autonomous. There is no notion that the chemical plant can be controlled by the system: it is only a subject of observation and reporting. So this frame does not fit any process-control problem. The *information outputs* are concerned solely with the *real world* and so can not furnish any information about previous requests. The system therefore can not provide an analysis of its own usage or of any other aspect of the managers' behaviour.

Such simplifications and restrictions are essential to problem frames. A frame delimits a class of problem by stripping away all the incidental complications that characterise realistic problems. The purpose of this simplification is to allow systematic methods of solution. There is a clear analogy with the usual procedures of mathematics.

In general, the more constraining the problem frame, the more useful an associated method can be. For example, the **simple IS frame** can be tightened even further to fit a still smaller class of problem. We may stipulate that the *real world* must be dynamic, not static; and that the *information requests* must be regarded as an unstructured stream of atomic events. These further simplifications are exploited by the JSD method [8] to give a reasonably systematic procedure for analysing the problem and developing a solution.

A sufficient set of problem frames for software development will have very many more than two members. Very few frames have yet been identified and defined. Purely as illustrative examples, three frames are sketched here in outline:

simple control frame The principal parts of a simple control problem are the *controlled domain*, the *desired behaviour*, and the *controller*. The *solution task* is to construct the *controller* so that it interacts with the *controlled domain* and brings about the *desired behaviour*.

This frame would be appropriate for controlling the chemical plant. The *controlled domain* must be dynamic, and partly autonomous and partly reactive. The *controller* must interact directly with the *controlled domain*. A method for a more elaborate version of this problem frame, in which input and output subsystems are interposed between the *controller* and the *controlled domain*, is described in [17].

workpieces frame The principal parts of a workpieces problem are the *workpieces*, *operation requests*, a *machine tool*, and *operation properties*. The *solution task* is to construct the *machine tool* so that in response to the *operation requests* it performs operations on the *workpieces* satisfying the *operation properties*.

This frame would be suitable for a simple text editor. The *workpieces* must be inert, and the *operation requests* must be viewed as an unstructured stream of events. For a problem fitting this frame it may be effective to use a model-oriented method [12], treating the *workpieces* as instances of an abstract data type.

supervised domain frame The principal parts of a supervised domain problem are the *supervised domain*, the *desired interaction*, the *supervisor*, and *safety actions*. The *solution task* is to construct the *supervisor* so that it interacts with the supervised domain, performing the *safety actions* when the *supervised domain* fails to interact with it in accordance with the *desired interaction.*

This frame is suitable for problems in which human operators—for example, aeroplane pilots or drivers of railway trains—are required to behave according to certain rules. The *supervisor* is the air traffic control system or railway signalling system with which they interact.

These frames must be filled out with much fuller characterisations of their principal parts and associated methods. Here only the briefest sketches are given.

5 Problem Complexity

A problem frame strips away the incidental complications that characterise realistic problems. A realistic problem can be regarded as an assemblage of simplified problems, each of which fits into a well-defined problem frame. Such a realistic problem exhibits *problem complexity.* It must be decomposed into simple problems for which effective methods are known. This decomposition is guided by an initial description and analysis of the problem context, and by knowledge of a repertoire of frames and associated methods.

In general, problem decomposition is into overlapping parallel frames: hierarchical problem structure is unusual. Two frames are interconnected by the domain phenomena they have in common. Two principal parts in different frames share domain phenomena in roughly the same kind of way as parallel CSP [6] processes share common events.

Consider, for example, a simple text editor. A first complexity may be that one view of the text is insufficient. For inserting and deleting words it is appropriate to view the text as a simple character string; but for screen display and cursor movement, and for printing, the text is structured as a sequence of lines. The problem can be regarded as requiring two **workpieces frames**, one for each view. The two views are pinned together by their common phenomena: the characters in the string are the characters in the lines (excluding the newlines and hyphens). An illuminating discussion of this complexity in terms of Z schemas may be found in [7].

Another complexity in the text editor may be that the users are to be constrained by certain rules of procedure. The text editor itself fits into one or two instances of the **workpieces frame**; imposition of the constraints fits into the **simple control frame**. Both are concerned with the same events: the *operation requests* and the resulting performance by the *machine tool* of operations on *workpieces.* Notice that in the **workpieces frame**, these are events in separate domains; but in the **simple control frame** they are events in a single *controlled domain.* Notice also that the *machine tool*, which in the **workpieces frame** is the machine to be developed, appears in the **simple control frame** as a part of the *controlled domain.*

Typically, different frames demand different views of the problem context, and different groupings of domains and their phenomena for allocation to the principal parts of the frames. Different views are also needed of the *control* characteristics of domains and their phenomena. In the **workpieces frame** the *operation requests* are autonomous: the users perform them at will and the *machine tool* responds reactively in accordance with the *operation properties.* But in the **simple control frame** these events become subject to inhibition by the *controller.*

Different views are similarly needed of the solution parts of the machine being built. This means composing different architectures in ways that are not normally considered when the architectures are considered separately. A clear example is given by Garlan and Shaw [5]. From one point of view a pipe-and-filter architecture is appropriate to the processing needs of a certain problem in oscilloscope design. Each filter can be thought of as an autonomous process, whose execution is constrained only by the availability of its input. But the users of the oscilloscope must be able to interact with it, and so must be able to impose some control on the filter processes. This is the solution counterpart of the different views of the control characteristics of problem domains as seen in different problem frames.

6 Related Work

The notion of problem frames is rooted in the work of Polya and the Greek mathematicians. That work appears not to have been exploited elsewhere. The approach to complexity has something in common with the work commonly subsumed under the term ‘viewpoints’, but there are substantial differences.

The work of Finkelstein Nuseibeh and Kramer [16, 3] is more concerned with the management of development methods and their products, and less directly with the analysis of the problem in hand.

The CORE method [15] is also based on viewpoints: a CORE viewpoint is associated either with an entity that interacts with the system in some way or with a sub-process of the system.

The work of Leite [14] focuses on viewpoints based on the differing perceptions of people involved in a development. “A viewpoint is a standing or mental position used by an individual when examining or observing a universe of discourse.” There is a strong emphasis in this work, as in the work of Easterbrook [2], on the pitfalls of natural language, and the need to resolve conflicts in its use.

An overview of viewpoints work can be found in [13].

References

[1] Christopher Alexander, Sora Ishikawa and Murray Silverstein; A Pattern Language; Oxford University Press, New York, 1977.

[2] Steve Easterbrook; Handling Conflict Between Domain Descriptions with Computer-Supported Negotiation; Knowledge Acquisition Number 3 pages 255-289, 1991.

[3] A Finkelstein, J Kramer, B Nuseibeh, L FInkelstein, M Goedicke; Viewpoints: A Framework for Integrating Multiple Perspectives in System Development; International Journal of Software Engineering and Knowledge Engineering Volume 2 Number 1 pages 31-57, March 1992.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides; Patterns: Elements of Object-Oriented Software; Addison-Wesley, 1994.

[5] David Garlan and Mary Shaw. An Introduction to Software Architecture; in Advances in Software Engineering and Knowledge Engineering Volume 1, V Ambriola and G Tortora eds; World Scientific Publishing Co, New Jersey, 1993.

[6] C A R Hoare; Communicating Sequential Processes; Prentice-Hall International, 1985.

[7] Daniel Jackson; Structuring Z Specifications with Views; Carnegie Mellon University Report CMU-CS-94-126, March 1994.

[8] Michael Jackson; System Development; Prentice-Hall International, 1983.

[9] Michael Jackson and Pamela Zave; Domain Descriptions; in Proc RE'93 pages56-64; IEEE, 1993.

[10] Ralph E Johnson; Documenting Frameworks using Patterns; OOPSLA'92 Proceedings, ACM SIGPLAN Notices Volume 27 Number 10, pages 63-76, October 1992.

[11] Ralph E Johnson; Why a Conference on Pattern Languages? ACM SE Notes, Volume 19 Number 1, pages 50-52, January 1994.

[12] Cliff B Jones; Systematic Software Development Using VDM; Prentice-Hall International, 2nd Edition 1990.

[13] Gerald Kotonya and Ian Sommerville; Viewpoints for requirements definition; Software Engineering Journal Volume 7 Number 6 pages 375-387, November 1992.

[14] Julio Cesar Sampaio do Prado Leite and Peter Freeman; Requirements Validation Through Viewpoint Resolution; IEEE Transactions on Software Engineering Volume 17 Number 12 pages 1253-1269, December 1991.

[15] G P Mullery; CORE – a Method for Controlled Requirements Specification; in Proc ICSE-4 pages 126-135; IEEE, 1979.

[16] Bashar Nuseibeh, Jeff Kramer, Anthony Finkelstein; Expressing the Relationships Between Multiple Views in Requirements Specification; in Proc ICSE-15 pages 187-196; IEEE, 1993.

[17] D L Parnas and J Madey; Functional Documentation for Computer Systems Engineering (Version 2); CRL Report 237, McMaster University, Hamilton Ontario, Canada, 1991.

[18] G Polya; How To Solve It; Princeton University Press, 2nd Edition, 1957.

[19] Mary Shaw, David Garlan, Robert Allen, Dan Klein, John Ockerbloom, Curtis Scott, Marco Schumacher; Candidate Model Problems in Software Architecture; Software Architecture Group, Carnegie Mellon University, internal document, December 1993.

[20] Pamela Zave and Michael Jackson; Conjunction as Composition; ACM Transactions on Software Engineering and Methodology, Volume 2 Number 4 pages 379-411, October 1993.

14/11/94

M A Jackson
101 Hamilton Terrace
London NW8 9QX
mj@doc.ic.ac.uk
jacksonma@attmail.att.com
+44 0171 286 1814 (voice)
+44 0171 266 2645 (fax)