Problem Structure And Dependable Architecture

Michael Jackson

Faculty of Mathematics & Computing, The Open University,
Milton Keynes MK7 6AA, England
jacksonma@acm.org
http://mcs.open.ac.uk/mj665

Abstract. An approach to software development is sketched in which problem structuring is separated from software architecture. The problem is decomposed into subproblems of familiar classes that can be considered in isolation; then the interactions among the subproblems are considered. The architectural task is seen as the task of composing the software machines associated with each subproblem and with the more complex interactions among them. It is suggested that such an approach embodies a good separation of concerns that can contribute to achieving system dependability.

Introduction

Software architecture, according to Shaw and Garlan [9] is concerned with "... the organization of a system as a composition of components; global control structures; the protocols for communication, synchronization and data access; the assignment of functionality to design elements; the composition of design elements; physical distribution; scaling and performance; dimensions of evolution; and selection among design alternatives." We may ask how these concerns impinge on system dependability, and—if they do—how to address them in a way that will improve dependability.

An obvious analogy is with the dependability of engineered physical structures. Many notorious engineering failures can be traced to structural design faults. A structure that has been incorrectly designed to carry the imposed loads will fail in use. The careful investigation that follows a failure reveals the design error; popularising books on engineering [7, 2, 5] provide lucid explanations for lay readers. But it is far from clear that the analogy is sound. Software is not itself a physical product, and the forces imposed on it are, for the most part, not usefully quantifiable like the forces on a beam or a truss. There are, of course, aspects of some systems where numerical calculations of bandwith, network traffic, response times, or computational complexity are critical to successful design. But for most aspects of the broad range of systems this is not so.

Another important difference is that software is extremely malleable. Software for a digital computer evokes a computation describable by a state machine: the transitions of this machine can be grouped and configured in many different ways

without affecting the evoked computation. Presenting two candidate modularisations of the KWIC problem [6], Parnas wrote:

"The systems are substantially different even if identical in the runnable representation. This is possible because the runnable representation need only be used for running; other representations are used for changing, documenting, understanding, etc. The two systems will not be identical in those other representations."

The 'runnable representation' is only one of several architectures of a system. Other representations—and hence other architectures—are largely concerned with human understanding. They embody attempts to master the complexity of a real-world problem and of the software that must lie at the core of its solution, and to ensure that all important concerns are adequately addressed. The goal for the software architect is to avoid certain classes of system failure. Not all failures can be avoided by software structure aimed at mastering complexity: examples of those that can not include failures arising from poorly designed human interfaces, configuration errors, unreliable hardware, slow response, inadequate throughput, and from many other causes. But one important class that can be so addressed is functional failure, in which the observable behaviour of the system is not what was intended or desired. In this class we include failures to meet requirements of safety and reliability, and also failures to repair or conceal or mitigate a malfunction where such response to the malfunction is, or should be, a functional requirement of the system.

A View of Software Development

The principal parts[1] of a software development problem are:

- the *problem world*, where the problem is located: for a lift control system this is the users, floors served, lift car and shaft, doors, request buttons, winding gear, indicator lights, floor sensors, and so on;
- the *requirement*, which is the behaviour to be established and maintained in the problem world: for example, that the doors open only when the lift car is at a floor, and that the lift comes when summoned and goes to the requested floor;
- the *machine*, which is the hardware-plus-software computer to be designed and installed in the problem world and connected to it by the *machine interface*: for the lift system this interface would be the port connections to the motor control, button sensors, indicator lights, floor sensors, and so on.

The goal of the development is to devise, specify and build a machine that will guarantee satisfaction of the requirement by exploiting and respecting the given properties of the problem world. In the lift control problem these are the physical properties that cause the lift car to rise when the motor is set *on* and *up*, the floor sensor to close when the lift car arrives at the floor, and so on.

Because the requirement and the problem world are complex, the development can fail in many ways. The requirement may have been misunderstood; the given properties of the problem world may have been misunderstood; the machine that is

[1] 'Principal parts' is a term taken from [8].

built may not satisfy its specification; the specification may be faulty—not guaranteeing satisfaction of the requirement even if the requirement and the problem world properties have been correctly understood and represented. It is a principal goal of problem structuring—that is, of problem architecture—to achieve a clarity of understanding that makes such failures avoidable.

Architecture and Decomposition

The key to mastering complexity is the separation of concerns, but we must clarify what this means for software development. Architecture is concerned with structuring the machine by organising it "as a composition of components" and with "the assignment of functionality to design elements". This demands a structuring of functionality. The problem must be structured into *subproblems*, whose solutions can eventually be assigned to software components. This structuring into subproblems is primarily a decomposition of the problem requirement. Each subproblem has its own requirement and its own problem world, which is a projection of the problem world originally given. The problem world too demands to be structured, both to support the problem decomposition and to separate parts whose interactions will be mediated by the machine. For example, it is convenient to separate the lift car in the shaft from the buttons and lights. We will regard the problem world, then, as an assemblage of *problem domains*, but we must not expect that exactly the same structuring will be appropriate for all subproblems. As Shaw and Garlan point out, there will be a need for "the composition of design elements". In fact we can go further: there will be a need for composition of problem elements more generally, including requirements and problem domains. Composition, as we shall see, is a major development task in its own right, with its own characteristic concerns.

The structuring of requirements or functionality is rarely a concern in the established branches of engineering, where most design work is *normal*, rather than *radical*, design [10]. The engineer engaged in normal design knows the *operational principle* of the device to be designed: that is, how it works, and how its characteristic parts fulfil their special function in combining to an overall operation which achieves the purpose. The designer of a car, for example, does not spend effort in decomposing the functionality that converts fuel combustion into movement of the car. Normal design dictates a decomposition into reciprocating engine, flywheel, gearbox, cardan shaft, differential gear, half-shafts and road wheels, arranged in a standard configuration and connected by well-understood interfaces.[2]

In software, by contrast, the decomposition of functionality is very often a task of radical design [1], in which:

"... how the device should be arranged or even how it works is largely unknown. The designer has never seen such a device before and has no presumption of success. The problem is to design something that will function well enough to warrant further development."

---

[2] Where there are choices—for example, between front-wheel and rear-wheel drive—the designer must choose from a very small number of such standard configurations.

A developer confronted by a genuinely radical design task can do little but resort to general principles and broadly formulated methods or design disciplines. They are, of course, a very inferior substitute for an established normal design practice specialised to the problem in hand.

A Problem Decomposition Discipline

One approach to the development task [4] is to regard it initially, and primarily, as a task of problem decomposition rather than of solution design. The approach does not aim at 'seamless development': no assumption is made that the problem structure will suffice for the solution architecture. Taking the view presented earlier of the principal parts of a software problem, the developer seeks to decompose the problem into a collection of subproblems, each with its problem world, requirement, and machine.

At this stage, conceptually, each subproblem is considered in isolation, supposing all the remaining subproblems to have been solved. For example, the *service* requirement of one identified subproblem might be to provide normal lift service on the assumption that the electromechanical equipment is functioning correctly, while the *safety* requirement of another is to monitor the equipment behaviour and, if serious malfunction is detected, to apply the emergency brake and hold the motor switch off. Then in the service subproblem the problem world properties take no account of possible malfunction or of the emergency brake. In the safety subproblem the problem world properties take no account of service requests or of indicator lights; the requirement is to monitor only the lift and door movements in response to the changing motor and door control states, and to take appropriate action in the event of malfunction.

This functional decomposition is guided above all by a need to identify subproblems of known classes. The space, *a priori*, of possible decompositions is very large. By insisting, so far as possible, that the arrangement and characteristics of the principal parts of each subproblem must conform to a known pattern or *problem frame* [4], the developer aims at two related goals, both contributing directly to dependability. First, it becomes easier to grasp and communicate the decomposition itself because an appropriate vocabulary is ready to hand. Just as a developer who uses an object-oriented design pattern [3] such as *Decorator* can easily hold in mind and communicate the pattern elements and the part of the problem to which it relates, so too a developer who identifies a *WorkPieces* or an *Information Display* subproblem can do the same. Second, a known problem frame to which an identified subproblem conforms should already be, or can eventually become, the object of specialised normal design practice and knowledge. The decomposition itself is locally validated by the knowledge that the identified subproblem is soluble and that its solutions have certain properties.

If the whole requirement has been structured as a set of subproblems of known classes then the design task is radical only in the sense that it is a novel composition of normally-designed components. The radical aspect of the development is restricted to the *composition concerns* (which we discuss in a later section), and does not reach down to the individual subproblems. Being able to treat the subproblems as objects of

normal design has a large positive effect on dependability. This positive effect goes well beyond the saving of development effort by the adoption of ready-made, tested, solutions. Any software-intensive system that interacts with the natural world is potentially vulnerable to that world's unbounded capacity for varied and novel behaviour. Developing a successful system depends on identifying and selecting those behaviours that are likely to prove significant, and making soundly judged decisions about the system properties needed to deal with them effectively. This selection and judgment can scarcely be achieved by working from first principles: it emerges as the fruit of long experience of the kind that is captured in a normal design.

The Impact of Decomposition

The kind of problem decomposition discussed here departs from current common practice. It is basically parallel rather than hierarchical. The service and the safety subproblems for the lift are parallel: neither one is a part of the other; and their solutions must run concurrently. Monitoring for equipment malfunction must continue alongside the provision of lift service in the absence of serious malfunction. Essentially, each subproblem is directly connected to the parts of the problem world that are relevant to satisfying its requirement.

This is not to say that there is no hierarchical structure anywhere in the decomposition. The practice of normal design is itself concerned with a structure of parts fulfilling a requirement, and this structure may be partly hierarchical. Normal design practice for the safety subproblem, for example, may dictate a decomposition into a monitoring subproblem and an action subproblem, and a further decomposition of the monitoring subproblem into a part that builds and maintains a model or simulacrum of the equipment and its behaviour, and another that diagnoses malfunctions from the model. Within the safety subproblem, then, there is a local hierarchical structure fitting into the larger parallel structure of the whole problem.

The basic decomposition technique achieves a simplification of the individual subproblems. The developer of the service subproblem is not concerned with the possibility of malfunction: it has been specifically excluded from consideration. Similarly the developer of the safety subproblem is not concerned with whether or how lift service is provided: the problem world to be monitored is simply one in which motor and door control states are changing spontaneously, and the states of the door and floor sensors may or may not be changing as they should in response. This separation of concerns is quite subtle, but, like many successful separations, it makes a substantial contribution to the reduction of complexity: for the safety problem the rich possibilities of scheduling lift movements in response to service requests are abstracted away, leaving only a much simpler world of spontaneous changes of motor and door control states.

Another impact of this decomposition is that the problem worlds of different subproblems intersect, but analysis and solution of the subproblems may depend on assuming different—and possibly incompatible—properties of their problem worlds. This difference may be no more than a difference in the granularity with which the behaviour of a particular problem domain is viewed; but it may be much more than

that. For example, in the service problem the analysis assumes that the lift car always moves upwards when the motor state is *on* and *up*; in the safety subproblem the assumption is that it may fail to do so because of some equipment malfunction.

Composition Concerns

A very large part of the complexity of any realistic system lies in the interaction of subproblems. A major motivation for decomposition into distinct subproblems is to avoid the combinatorial explosion of the possible states in each subproblem. Misguided decomposition can lead to gratuitous complexity, forcing the developer to consider combinations of requirements or behaviours that a better decomposition would reveal to be orthogonal. But some of the subproblem interaction complexity is inherent in the problem.

The form of decomposition we are discussing here postpones consideration of problem interactions until the interacting subproblems have been identified and analysed. The interactions then present themselves in the form of composition concerns. If we imagine conjunctions of all the subproblems' machine behaviours, all the problem domain properties on which they depend, and all their requirements, we may ask whether these conjunctions, taken together, constitute an adequate analysis and solution of the original problem? If not, what additions and changes are necessary? To ask and answer these questions is to address the composition concerns.

One example of a composition concern is direct requirement conflict. The requirements of two subproblems may, in some circumstances, contradict each other. If a malfunction has been detected in the lift equipment at a time when a user has just pressed a button to request lift service, then the service requirement demands that the motor be switched on to move the lift car in response to the request, while the safety requirement demands that the motor be switched off. To address this concern it is necessary to give precedence to one of the conflicting requirements, and to describe their composition in a way that embodies this decision. This may, in some cases, demand the recognition of a fresh subproblem, in which the machines of the subproblems to be composed appear as problem domains and the composition rule is regarded as a fresh requirement to be satisfied by the new machine.

Another example of a composition concern is interference. If the safety subproblem has been decomposed into a part that builds and maintains a model of the lift equipment behaviour, and another part that diagnoses malfunctions by inspecting the model, then the composition must deal with the resulting interference. The model is shared data for the two subproblem parts, and a suitable granularity must be chosen for the necessary mutual exclusion.

As a third example, consider a decomposition of a lending library system in which one subproblem deals with membership, regarding book loans as atomic events, and another deals with loans, regarding membership as static. In their composition it is necessary to deal with the interactions that arise from these two simplifications. What, for example, is the required system behaviour when a two-week book loan is requested by a member whose membership is due to lapse in one week?

These composition concerns seem to arise from the simplification (oversimplification, we may honestly say) of the subproblems. But they were always present in the original problem, and the decomposition has merely placed them in a context in which they can be dealt with explicitly. In a more usual approach the composition concerns are dealt with piecemeal as they come to attention in each subproblem. This piecemeal approach has severe disadvantages. One disadvantage is the added complication of the subproblem while its basic substance is not yet well understood: this is an unwelcome distraction from the subproblem concerns in hand. Another is that the composition concern itself is then being approached from one side rather than the other, leading potentially to an asymmetry that distorts what may very well be a symmetric composition concern. Another, deeper, disadvantage is that the composition itself may well merit the status of a subproblem in its own right, but yet be denied the appropriate focused concentration of the developers' attention.

Architectures and Subproblem Implementations

Having addressed the decomposition, the resulting subproblems, and their subsequent composition, the development must proceed to an implementation. In the view we are taking here, this obligation focuses on designing a software structure that will accommodate all[3] of the subproblem machines—including any additional machines arising from their composition.

We may identify this design task with a central aspect of what is usually considered to constitute software architecture design. The functionality of the system, including the subproblem interactions, has been fully specified in the machines to be accommodated in the architecture. These specifications, however, are still in some respects abstract. Consider, to take a simple example, a pair of subproblem machines *M1* and *M2* that interact by respectively writing and reading a sequential data stream *S*. The granularity of the interaction has already been determined, but the interleaving of the machines, and the interfaces they present to other software components, have not. The possible implementations, exploiting the malleability of software, include:

- *M1* and *M2* are run as separate threads communicating by a bounded buffer *S* that enforces the necessary write-read exclusion;
- *M1* and *M2* are run sequentially in that order, communicating by a buffer *S* (possibly on disk) that accommodates the whole of *S*;
- *M2* is implemented as a procedure invoked by *M1*, each invocation passing a record of *S* from *M1* to *M2*;
- *M1* is implemented as a procedure invoked by *M2*, each invocation passing back a record of *S* from *M1* to *M2*.

Choosing an implementation from such a set of possibilities is a local choice of architectural style: there is no reason *a priori* to assume that the choice of architectural style must be global for the system. The primary concern in architectural design of this kind is clearly to accommodate the subproblem machines correctly,

[3] For brevity and simplicity, we are ignoring the possibility of an implementation using distributed hardware.

ensuring that their inputs are made available, their outputs sent to the appropriate destinations, their persistent data preserved, enough compute cycles provided for their execution, and so on.

Many other architectural concerns must also be addressed. One important such concern is reliability with respect to failures— for example, failures of computer hardware —that can not be addressed conveniently, or at all, except in the context of architectural design. In the problem analysis that conceptually precedes architectural design, it is a useful separation of concerns to assume that the computer executing the software for each subproblem machine is perfectly reliable: unreliability in the problem world—for example, malfunction of the lift equipment—is dealt with as a problem decomposition concern, but computer malfunction is not. It is a part of architectural design to consider the use of such techniques as triple modular redundancy to avoid system failure in the presence of computer hardware malfunction.

The possibility of failures of the software itself, due to faults in the problem analysis, subproblem machine specification, or programming, must also be addressed. In addressing requirement conflict among the composition concerns it was necessary to establish a precedence among requirements: the safety requirement was more important than the service requirement, so the safety requirement took precedence in the event of conflict. The conceptual relationship between these two requirements is clear: ideally we would like both good service and safety; but if we are ever forced to choose we will choose safety. A similar conceptual relationship holds with respect to functional dependability in the presence of software faults: ideally we would like all system functions to be fully dependable; but if we are forced to choose we would certainly prefer a system in which the safety function is more dependable than the service function. To ensure this ordering of dependability is an architectural concern. Suppose, for example, that the dependability of the requirement satisfied by our subproblem machine *M1* is more important than that of the requirement satisfied by *M2*. Then the architect must choose an implementation structure in which software failure of *M2* can not cause failure of *M1*. This consideration should probably lead the architect to exclude, for example, the tightly-coupled architectural designs in which the two components are connected by procedure call.

Summary

The approach roughly sketched here pays explicit attention to the problem architecture before addressing the software architecture. Subproblems of familiar classes can be solved more reliably than unfamiliar problems, because their solutions draw on the communal experience that is embodied in normal design practice. In the problem architecture subproblems of familiar classes are identified, and their composition in the problem space is then considered. The implementation and configuration of the resulting machines then becomes the central theme of the software architecture. The time ordering of development tasks implicit in this sketch can be viewed as a methodological prescription for development. But it can also be viewed more abstractly as a basis for understanding the relationship of problem

structure to software architecture, or even for reverse-engineering an existing architecture to expose its structural relationship to the problem it solves.

Both in the analysis of the problem and the design of the software architecture the approach could be characterised as bottom-up rather than top-down. Subproblem composition is deferred until the subproblems have been analysed and, essentially, solved. Software architecture is deferred until the components—the subproblem machines—that are to be accommodated are well understood. Essentially this means that the requirements and problem domain properties have been analysed and a specification has been derived of the external behaviour of the machine that can guarantee satisfaction of the requirement. Much of the complexity of software development, and hence the potential for failure, springs from undesired or unforeseen interactions. By postponing composition until the parts to be composed—whether subproblem requirements or subproblem machines—are well understood, the approach aims to get a better grasp of interaction complexity and so to improve system dependability.

References

1. E W Constant; *The Origins of the Turbojet Revolution*; The Johns Hopkins University Press, Baltimore 1980.
2. Eugene S Ferguson; *Engineering and the Mind's Eye*; MIT Press, 1992.
3. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides; *Design Patterns: Elements of Object-Oriented Software*; Addison-Wesley, 1994.
4. Michael Jackson; *Problem Analysis and Structure*; in *Engineering Theories of Software Construction*, Tony Hoare, Manfred Broy and Ralf Steinbruggen eds; Proceedings of NATO Summer School, Marktoberdorf; IOS Press, Amsterdam, Netherlands, August 2000, pp3-20.
5. Matthys Levy and Mario Salvadori; *Why Buildings Fall Down: How Structures Fail*; W W Norton and Co, 1994.
6. D L Parnas; *On the Criteria To Be Used in Decomposing Systems into Modules*; Communications of the ACM Volume 15 Number 12, pages 1053-1058, December 1972.
7. Henry Petroski; *To Engineer is Human: The Role of Failure in Successful Design;* St. Martin's Press, New York, 1985; Macmillan, London, 1986.
8. G Polya; *How To Solve It*; Princeton University Press, 2nd Edition 1957.
9. Mary Shaw and David Garlan; Software Architecture: Perspectives on an Emerging Discipline; Prentice-Hall 1996.
10. Walter G Vincenti; What Engineers Know and How They Know It: Analytical Studies from Aeronautical History; The Johns Hopkins University Press, Baltimore, paperback edition, 1993.

Architecting Dependable Systems III
edited by R de Lemos, C Gacek, A Romanovsky