Table of Contents and Extract from “The Structure of Software Development Thought”

Michael Jackson

Independent Consultant

Table of Contents

Introduction
Intellectual Structure
The Sluice Gate: Problem Introduction
Problem Phenomena and the Requirement
Problem World Decomposition
Problem World Properties
Specification of the Machine
Problem Reduction
A Specification Difficulty
Specification by Rely and Guarantee Conditions
The Breakage Concern
The Initialisation Concern
Combining Machines
Problem Decomposition
The Domain Reliability Concern
Defining and Diagnosing Equipment Failure
The Approximation Concern
Combining the Safety and Irrigation Requirements
A Recapitulation of Principles
The Primacy of Normal Design
Software Developers and the Problem World
Deferring Subproblem Composition
Separating the Normal and Error Treatments
Problem Scope and Problem Domains
References

Extract

Introduction

Software developers have long aspired to a place among the ranks of respected engineers. But even when they have focused consciously on that aspiration [15, 3] they have made surprisingly little effort to understand the reality and practices of the established engineering branches.

One notable difference between software engineering and physical engineering is that physical engineers pay more attention to their products and less to the processes and methods of their trade. Physical engineering has evolved into a collection of specialisations–electrical power engineering, aeronautical engineering, chemical engineering, civil engineering, automobile engineering, and several others. Within each specialisation the practitioners are chiefly engaged in *normal design* [18]. In the practice of normal design, the engineer

> *"... knows at the outset how the device in question works, what are its customary features, and that, if properly designed along such lines, it has a good likelihood of accomplishing the desired task."*

••• •••

Software engineering, or, more generally, the development of software-intensive systems, has not yet evolved into adequately differentiated specialisations, and has therefore not yet established normal design practices. There are, of course, exceptional specialised areas such as the design of compilers, file systems, and operating systems. But a large part of the development of software-intensive systems is characterised by what Vincenti [18] calls *radical design*:

> *"In radical design, 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."*

Developers compelled to engage in radical design are, by definition, confronted by a problem for which no established solution–or even solution method–commands conformity. In some cases the problem has no clearly applicable precedent; in some cases there are precedents, but they lack the authority of a long history of success. Naturally attention turns to the question "How should we tackle

this problem?". Many widely varying methods and approaches are proposed, and advocated with great confidence.

Intellectual Structure

Some of these methods and approaches are chiefly concerned with managerial aspects of development: for example, application of industrial quality control techniques, or the use of team communication practices such as stand-up meetings, open-plan offices and pair programming. But software development has an inescapable intellectual content. Whether they wish to or not, developers of software-intensive systems inevitably separate concerns–well or badly–if only because it is not humanly possible to consider everything at once. They direct their attention to one part of the world and not to another. Some information they record explicitly, and other information is left implicit and unrecorded. Consciously or unconsciously they reason about the subject matter and about the product of their development, convincing themselves, well enough to allay discomfort, that what they are doing is appropriate to their purposes. In short, development work is performed within some intellectual structure of investigation, description and reasoning. In the small this structure sets the context of each task; in the large it gives grounds for believing that the system produced will be satisfactory.

This intellectual structure is the topic of the present chapter. The emphasis will be chiefly on the understanding of requirements and the development of specifications, rather than the design of program code. Among requirements the focus will be on functional requirements–including safety and reliability–rather than on non-functional requirements such as maintainability or the cost of development. The intellectual structure is, necessarily, a structure of descriptions of parts and properties, given and required, of the whole system, and of their creation and their use in reasoning. Because we aim at brevity and a clear focus on the structure rather than on its elements, we will present few full formal descriptions: most will be roughly summarised to indicate their content and scope. A small example, more formally treated elsewhere [8], will be used as illustration; its limitations are briefly discussed in a concluding section.

●●● ●●●

The Sluice Gate–1: The Problem and Its World

A small example problem [8] provides a grounding for discussion. Although it is both too narrow and too simple to stand as a full representative of development problems in general, it allows us to pin our discussion to something specific.

The Problem

The problem concerns an agricultural irrigation system. Our customer is the farmer, whose fields are irrigated by a network of water channels. The farmer has purchased an electrically-driven sluice gate, and installed it at an appropriate place on his network of irrigation channels. A simple irrigation schedule has been chosen for this part of the network: in each period of the schedule a specified volume of water should flow through the gate into the downstream side of the channel. A computer is to control the gate, ensuring that the schedule is adhered to. Our task is to program the control computer to satisfy the farmer's requirement.

According to one engineer's definition [17], this is a classic engineering task:

> *"Engineering ... [is] the practice of organizing the design and construction of any artifice which transforms the physical world around us to meet some recognized need."*

The farmer's recognized need is for scheduled irrigation; the physical world around us is the sluice gate and the irrigation channel; and the artifice we are to design and construct is the working control computer. The computer hardware, we will suppose, is already available; our task is only to program it appropriately, thus endowing it with a behaviour that will cause the recognized need to be met. Fig. NNN.1 depicts these elements of the problem and connections between them.

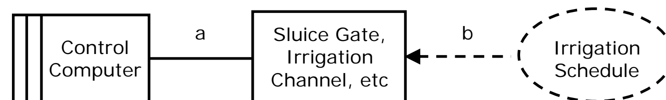

Fig. NNN.1. A Problem Diagram

The striped rectangle represents the *machine domain*: that is, the hardware/software device that we must construct. The plain rectangle represents the *problem world*–the physical world where the recognized need is located, and where its satisfaction will be evaluated. The dashed oval represents the customer's *requirement*. The plain line marked *a*, joining the machine domain to the problem world, represents the interface of physical phenomena between them–control signals and states shared between the computer and the sensors and actuators, by which the machine can monitor and control the state of the sluice gate equipment. The dashed line with an arrowhead, marked *b*, joining the requirement to the problem world, represents the phenomena to which the requirement makes reference–the channel and the desired water flow.

These three main elements of the problem, and the relationships among them, are fundamental to the intellectual structure presented here. The requirement is a condition on the problem world, not on the machine domain: no water flows at the machine's interface with the sluice gate equipment. Nonetheless, the machine can, if appropriately constructed, ensure satisfaction of the requirement by its interaction with the sluice gate at interface *a*. This is possible only because the problem world has certain physical properties that hold regardless of the presence

or behaviour of the machine: the gate is operated by an electric motor driving two vertical screws which move the gate up and down between its open and closed positions; if the gate motor is set to upwards and switched on, then the motor will turn the screws and the gate will open; when the gate is open (and the water level is high enough) water will flow through the channel, and when the gate is closed again the flow will cease. These are the properties we must exploit in our solution.

For some well-designed behaviour of the machine, the resulting openings and closings of the gate will ensure the required irrigation pattern. To demonstrate eventually that this is so we must offer an *adequacy argument*. That is, we must show that

machine $\wedge$ *problem domain* $\Rightarrow$ *requirement*

We may regard the problem as a *challenge* [7] to the developers: given the problem domain and the requirement, devise and build a machine for which the adequacy argument will go through and the implication will hold. For the problem in hand the specific implication is:

Control Computer $\wedge$ *Sluice Gate, Irrigation Channel etc*
$\Rightarrow$ *Irrigation Schedule*

Although informally stated, this is the essence of what it will mean to satisfy the customer's requirement.

••• •••

A Recapitulation of Principles

In this final section we recapitulate some principles that have already been stated, and briefly present some others that have so far been only implicit.

The Primacy of Normal Design

An overarching principle must always be borne in mind: by far the surest guarantee of development success is normal design practice developed over a long history of successful products in a specialised application area. Even in a small problem there are many imponderables to consider in understanding the properties of the physical problem world. Which failures of the Sluice Gate Equipment are most likely to occur? How far is the equipment likely to stray from its designed performance in normal operation over its working life? Which normal operation regimes place least strain on the equipment? Which degraded operation modes are really useful for staving off impending failures? What is the best way to separate and then to compose the subproblems? What are the best choices to make in each stage of problem reduction?

An established normal design practice does more than provide explicit tested answers to these difficult questions. It also provides the assurance of successful experience that all the important concerns have been addressed. A normal design

practice does not address all conceivably relevant concerns explicitly: it embodies the lessons of experience that has shown that some concerns which might, *a priori*, appear significant are in fact not significant and can be neglected without risk of serious system failure, while others, apparently unimportant, are essential. This assurance is of crucial practical importance. The natural world is unbounded, in the sense that all the concerns that may conceivably be important can not be exhaustively enumerated. The designer starting from first principles, however sound they may be, cannot hope to address all the important concerns and only those. This is why the radical designer, in Vincenti's words [18], "has no presumption of success", and can hope only to "design something that will function well enough to warrant further development." Many of the system failures catalogued in the Risks Forum [16] arise from errors that are perfectly obvious–but obvious only after they have been highlighted by the failure.

Software Developers and the Problem World

The distinction between the machine and the problem world is fundamental. It is a distinction between what the programmer sees and what the customer or sponsor sees; between what is to be constructed and what is, essentially, given. It is not a distinction between computers and everything else in the world: in Fig. NNN.4 the Irrigation and Safety machines are treated as problem domains although each one is certainly to be realised as software executing on a general-purpose computer–probably sharing the same hardware with each other and with the Composition machine.

Our discussion of the development has focused entirely on the problem world in the sense that all the phenomena of interest–including those shared with the machine–are phenomena of the problem world. The requirements, the problem domain properties, and even the machine specifications, are expressed in terms of problem world phenomena. We have stayed resolutely on the problem side of Dijkstra's firewall.

It may reasonably be asked whether in our role as software developers we should be so concerned with the problem world. We may be more comfortably at home in an abstract mathematical problem world, in which the problem is one of pure graph theory or number theory; or in an abstract computer science problem world, developing a theorem prover or a model checker. But what business have we with irrigation networks and the electro-mechanical properties of sluice gates?

The answer can be found in the distinction between the earlier, richer, descriptions of the problem world properties–all of them contingent and approximate–and the later, formal and more abstract descriptions of the rely and guarantee conditions used in the machine specifications. The later descriptions must be formal enough and exact enough to support a notion of formal program correctness with respect to the specifications. Constructing them from the richer descriptions must be a task for software developers, even if responsibility for the richer descriptions and for the choice of the properties reliable enough to to be formalised may often–perhaps almost always–lie elsewhere. The domain expert and the software expert must work together here.

Deferring Subproblem Composition

In the preceding sections we followed the principle that subproblems should be identified and their machines specified before the task of composing or recombining them is addressed. The composition of the Initialisation and Irrigation subproblems was considered only after each had been examined in some depth; and the further composition of these subproblems with the Safety subproblem was similarly deferred.

This postponement of subproblem composition is not, by and large, the common practice in software development. More usually consideration of each subproblem includes consideration of how it must interact, and how it is to be composed, with the others. The apparent advantage of this more usual approach is that subproblem composition ceases to be a separate task: effort appears to be saved, not least because subproblems will not need reworking to fit in with the postponed composition.

The advantage of the common practice, however, is more apparent than real, because it involves a serious loss of separation of concerns. When composition is postponed, subproblems can be seen in their simplest forms, in which they are not adulterated by the needs of composition. Sometimes the simplest form of a subproblem can be recognised as an instance of a well-known class, and treated accordingly: the subproblem, considered in isolation, may even be the object of an established normal design practice. If all the subproblems can be treated in this way, the radical design task becomes radical only in respect of the subproblem composition. Whether the subproblems are well known or not, postponed composition is itself easier, simply because the subproblems to be composed have already been analysed and understood. By contrast, when composition is considered as an integral part of each subproblem, the composition concerns–for example, subproblem scheduling and precedence with respect to requirement conflicts–must be dealt with piecemeal in a distributed fashion, which makes them harder to consider coherently.

Separating the Error and Normal Treatments

Separating the development of normal operation of the sluice gate from the detection and handling of problem domain failures led to two distinct descriptions of problem domain properties. The properties of the correctly functioning equipment are captured in the *gate_movement* and *sensor_settings* (and also *gate_movement_1*) descriptions; its properties when it is failing are captured in the *gate_failure_properties* description. The two descriptions capture different and conflicting views of the domain, useful for different purposes.

This separation is salutary for the usual reasons that justify a separation of concerns. Each description separately is much simpler than they can be in combination; and each contains what is needed to carry through the part of the adequacy argument that relates to its associated subproblem.

It is worth observing that this kind of separation is hard to make in a traditional object-oriented style of development. The original basic premise of object

orientation is that software objects represent entities of the problem world, and each one should encapsulate all the significant properties of the entity that it represents. Adopting this premise requires the developer to combine every view of the entity, in all circumstances and operating modes, in one description. However, the patterns movement [6, 2, 5], showing a more insightful approach, has been busily working to discard this restriction by recognising the value of *decorator* and other such patterns.

Problem Scope and Problem Domains

We have assumed until now, as the basis of our discussion, that the farmer has chosen an irrigation schedule and accepted that this is the requirement to be satisfied by the development. Why should we not instead investigate the farmer's larger purpose, which is, probably, to grow certain crops successfully? And, beyond that, to run the farm profitably? And, going even further, to provide eventually for a financially secure retirement? In short, how can we know where to place the outer boundary of the problem? The inner boundary, at the machine interface, is fixed for us in our role as software developers: we undertake to develop software, but not to assemble the computer hardware or to devise new chip architectures or disk drives. But the outer boundary in the problem world is harder to fix. What, so far as the development are concerned, is the overall requirement– that is, the 'real problem'? How much of the problem world do we have to include?

The outer boundary is restricted by the responsibilities and authority of the customer[1] for the system. If our customer were the company that manufactures the sluice gate equipment, we would probably be concerned only with operating the gate according to a given schedule, and not at all with the irrigation channel. Sometimes the customer chooses to present the developers with a problem that has already been reduced: our customer the farmer, we supposed, had already performed at least one reduction step by eliminating consideration of the crops to be irrigated. Whenever such a prior reduction has taken place, we can, of course, deal only with the corresponding reduced requirement: if the requirement is expressed in terms of crop growth, then the crops must appear explicitly as a domain in our problem world.

---

[1] We use the term 'customer' as a convenient shorthand for the people whose purposes and needs determine the requirement: that is, for those who are often called 'the stakeholders'.