ASPECTS OF ABSTRACTION IN SOFTWARE DEVELOPMENT

10TH JULY 2012

Michael Jackson
The Open University
jacksonma@acm.org

ABSTRACT

Abstraction is a fundamental tool of human thought in every context. This essay briefly reviews some manifestations of abstraction in every day life, in engineering and mathematics, and in software and system development. Vertical and horizontal abstraction are distinguished and characterised. The use of vertical abstraction in top-down and bottom-up program development is discussed, and also the use of horizontal abstraction in one very different approach to program design. The ubiquitous use of analogical models in software is explained in terms of analytical abstractions. Some aspects of the practical use of abstraction in the development of computer-based systems are explored. The necessity of multiple abstractions is argued from the essential nature of abstraction, which by definition focuses on some concerns at the expense of discarding others. Finally, some general recommendations are offered for a consciously thoughtful use of abstraction in software development.

KEYWORDS

abstraction, analogic model, bottom-up design, grounded abstraction, free abstraction, horizontal abstraction, monsters, refinement, theory, top-down design, vertical abstraction

1 INTRODUCTION

Abstraction is a fundamental human faculty for perception, action and thought at every level. It is manifested everywhere in the growth and exercise of our practical intellect from birth. It is a vital tool in the evolution and practice of science, mathematics and engineering. The development of software for programmable digital computers, in particular, presents imperative demands and irresistible opportunities for the exercise and study of abstraction. Writers on software development [Dijkstra72, JacksonD06, Kramer07] have regarded appropriate use of abstraction as a fundamental skill. This is why a discussion of abstraction is relevant to this special issue of the SoSyM journal on software development and modelling.

Because abstraction is found everywhere in our intellectual landscape, a short discussion can scarcely aim at a comprehensive survey of its value and practice. In this essay the approach is informal, and focuses chiefly on abstraction in software development. The discussion falls into three main sections. Section 2 offers a preliminary account of the development of certain aspects and uses of abstraction, and mentions some illustrative examples. Section 3 draws on this background in discussing different dimensions and forms of abstraction, purposes that it can serve, and the practical intellectual structures it can suggest and support. Section 4 examines some critical aspects of abstraction as it is commonly practised in the development of computer-based systems. A final section draws together some general observations, and recommends a carefully thoughtful use of this fundamental intellectual tool.

2 A PRELIMINARY ACCOUNT

The essence of abstraction is simple and unsurprising: to abstract is to set aside what is less relevant, focusing attention on what we judge more important for the purpose in hand. To recognise a persistent entity we focus on what persists, and abstract away what varies from one encounter to another. We recognise classes of entities by abstracting away the differences among their members, and by a further exercise of abstraction we may recognise a superclass of some classes already recognised. We identify a quality common to many instances of different kinds, and—in the spirit of Plato—regard the abstract quality itself as an individual thing. The abstractions that we acquire in our everyday lives, by exercising our own faculties and by learning from others in our society, furnish—quite literally—our view of the world. It is well established that the larger part of the scene that we think we see with our eyes is in fact supplied by mental activity in which we interpret the purely optical signals in the light of abstractions and expectations formed from past experience. As Richard Gregory writes [Gregory05]: "Without the computing power and memory that brains bring to bear, retinal images would be meaningless patterns of limited use—hence the importance of knowledge for seeing."

Abstractions are closely interwoven with theories about how the world is and how it works. We equip ourselves with a usable—though not always well-founded—repertoire of abstractions and associated theories. The theories are based on what we believe to be the properties and behaviour of the material reality from which we have drawn our abstractions. These are *grounded* abstractions: they spring from our experience and perception of the world and our efforts to understand it. Ostensibly the criterion of their validity is an objective correspondence to material reality; but in truth the criterion is how well they work in practice for the purposes we want them to serve. In early societies these purposes are primarily social, rather than intellectual, scientific, or technical, and the criterion of validity is therefore a general social acceptance. From this intuitive, tacit and informal purpose, repertoires of grounded abstractions grow by unguided evolution, and may evolve to embrace such notions as magic spells and witchcraft, or prophecy by examining the entrails of sacrificed animals. Social acceptance of the associated theories—for example, that crop failure is always traceable to the action of a nearby malicious witch—is unquestioned: members of the society "reason excellently in the idiom of their beliefs but they cannot reason outside, or against, their beliefs because they have no other idiom in which to express their thoughts" [Evans-Pritchard37].

A society that adopts ambitious engineering purposes—building royal tombs, temples, bridges, ships and irrigation schemes—must submit at least some of its grounded abstractions to more stringent tests of objective practical validity. Trade, especially between merchants from different societies, submits the accepted abstractions of value and exchange to tests of willingness between traders and of success and failure in each merchant's ambition for wealth. Large engineering projects demand not only careful attention to predictable behaviour of static structures, but also accurate calculation of the resources necessary for construction and of the taxation that can collect these resources. Engineers, traders and tax collectors are highly motivated to recognise, discover or invent new and better grounded abstractions to meet these more objective criteria of validity. In this way, commerce, building and land surveying stimulated the beginnings of geometry and arithmetic in ancient Egypt. Babylonians developed a workable number system, and knew many calculational procedures and heuristics. They knew that (3,4,5), (5,12,13), (65,72,97) and many other integer triples correspond to the lengths of the sides of right triangles.

These were large practical achievements, applying a substantial repertoire of grounded mathematical abstractions. For the ancient Greeks the theories associated with mathematical abstractions had an intellectual interest far beyond their immediate practical uses, and they began the conscious process by which grounded abstractions could free themselves from their worldly origins and become objects of intrinsic mathematical interest and focused study. The criteria of success in this study are no longer practical utility and fidelity to a specific physical reality, but elegance, internal consistency, intellectual richness and a fruitful connection to other mathematical ideas. The evolution of such *free* abstractions, dissociated from their material ground in the world, naturally tends to increase formality, because formality supports a precision of thought that promises greater certainty in reasoning and therefore greater power in developing theories. A clear distinction emerges between grounded and free abstractions. A grounded abstraction is satisfactory only if reasoning about the abstraction can be expected to produce results that are true of the subject in which it is grounded. A free abstraction is not to be judged by this criterion: a proposed non-Euclidean geometry cannot be proved unsatisfactory by showing that there is no physical reality that it describes. For a pure free abstraction, any correspondence to physical reality is irrelevant. As David Hilbert is reported [Weyl44] as saying:

> "It must be possible to replace in all geometric statements the words point, line, plane, by table, chair, mug."

Yet free abstractions may still be brought to bear on practical non-formal problems. If the objects and relationships in a practical worldly context can be convincingly mapped to objects and relationships of a free abstraction, then the theory of the abstraction may apply—at least approximately—to the world. The relationship between the free abstraction and the physical world is described [Hardy40] by the mathematician G. H. Hardy:

> “The geometer offers to the physicist a whole set of maps from which to choose. One map, perhaps, will fit the facts better than others, and then the geometry which provides that particular map will be the geometry most important for applied mathematics. I may add that even a pure mathematician may find his appreciation of this geometry quickened, since there is no mathematician so pure that he feels no interest at all in the physical world; but, in so far as he succumbs to this temptation, he will be abandoning his purely mathematical position.”

The effective practical application of mathematics to material reality often demands large efforts of calculation. Calculation was eased by mathematicians’ development of better algorithms and eventually by printed tables of logarithms and other functions. From the seventeenth century and even earlier, inventors devised calculating machines to perform simple arithmetical operations: Pascal’s was perhaps the most famous. In the 1820s Babbage conceived his Difference Engine as a machine to calculate and print mathematical tables, and later he conceived the Analytical Engine as a machine to execute general calculations specified by programs; but full working versions of these machines were never built. From around 1850 various desktop machines were devised that could conveniently execute individual arithmetic operations, but it was not until the development of digital electronic computers in the 1940s that general programmed calculations could be effectively and reliably mechanised.

Against this background, electronic computers were naturally seen at first as a tool for numerical calculations. The essence of the program’s task was to take the place of the human calculator, instructing the computer to perform the desired calculation by specifying an appropriate sequence of arithmetic operations. Programming was a mathematical endeavour purely because the work to be mechanised was mathematical. Nonetheless, a deeper and more fruitful view gradually became widely understood and adopted. As Dijkstra wrote, looking back much later [Dijkstra76]: “It used to be the program’s purpose to instruct our computers; it became the computer's purpose to execute our programs.” Gradually, the digital computer was recognised as a machine that can take any formal abstraction, suitably expressed in a programming language, and exhibit a behaviour in which that abstraction becomes a physical reality, realised in the dynamic fabric of the computer itself.

In the 1950s and 1960s computers increased in size, reliability and speed. Programs became more ambitious, more complex, and—not infrequently—impenetrably obscure. Work on program development method focused on program intelligibility, on specification and correctness, and on program design. Structured programming advocated an abstraction of control structure that could bring program text and program execution into a clearer relationship with each other and with the problem to be solved. The programming language Simula 67 [Dahl+72], motivated as its name suggests by programming for simulation problems, introduced the idea of a program component designed to capture an abstraction of an entity in the world to be simulated, such as a bus or a truck, or a customer in a busy post office: the software component’s behaviour would reflect the behaviour of the real-world entity.

Simula 67 also encouraged a general view of program design as an exercise in designing a structure of abstractions, a view adopted by Dijkstra in the design [Dijkstra68b] of the THE operating system. In this view, there is a conceptual gap between the high-level abstractions germane to the problem and the low-level abstractions offered by the programming language. This vertical gap must be bridged by a hierarchy of abstractions, each grounded in the abstractions at the next lower level. Whether the bridge is built in the top-down or bottom-up direction, or by a combination of both, the proposed technique is largely an exercise in inventing abstractions. A design process in the top-down direction, in which an abstraction is formulated before the ground from which it abstracts has been explored, is a form of what is called *refinement*. If there are multiple levels of design, this is *stepwise refinement*.

Stepwise refinement became enormously influential in various forms, including many manifestations of top-down program and system design. Simula 67’s notion of program components as abstractions of real-world entities was even more influential, stimulating the development of object-oriented programming languages and of abstract data types as a vehicle for software specification. Object classes and abstract data types can be designed to capture—though often only approximately—some properties of entities specific to a simulated or modelled reality, such as a bank account or the components of a radiation therapy machine. They can also capture free abstractions drawn from the discrete mathematics of programming—such as sets and stacks and queues—that can prove useful in program design.

The same ideas have continued to influence the development of *computer-based systems*. These are systems in which the computer interacts directly with the material *problem world* about which it computes and whose behaviour it must monitor and control. In such a system the success of the software is judged by whether it evokes the desired behaviour in the world: its development is therefore a task crucially different from programming.

In programming, narrowly understood, the computer is insulated from the physical reality—if any—of its subject matter. The programmer's task is to satisfy a formal specification of the computation result, expressed as a relation between computer inputs and outputs or between a precondition and postcondition on program variables. Of course every computation has some subject matter, which may be abstract or concrete, and the program design must rely on some abstraction of its subject matter. An abstraction which embodies a contradiction is likely to cause the program to fail. But a program based on an abstraction which self-consistently misrepresents its subject matter will not fail for that reason alone: the specification may be ill-chosen but the program may be correct. Dijkstra saw a formal functional specification [Dijkstra89] as a firewall to separate these two distinct concerns: the 'pleasantness' question whether a program satisfying that specification is desirable, and the 'correctness' question of how to design such a program. The programmer, *qua* programmer, is not required to consider the various aspects of the pleasantness question: Is the abstraction faithful to the reality? Does the computer input correctly represent the state of that reality? What use will be made of the computed results?

For a computer-based system these questions can be neither delegated nor evaded: their answers are tightly integrated with the development of the software. Because the computer is directly interfaced to the problem world by sensors and actuators, its behaviour at the input-output interface is *ipso facto* a behaviour of some part of the physical problem world and also a cause of behaviour of other parts. For the whole functionality of a computer-based system, Dijkstra's proposed separation is impractical: one might as well try to separate the two sides of a densely meaningful human conversation. In such a system, the complexity of the problem world and of the behaviour that the software must evoke in it present a major challenge at every level to the effective use of abstraction.

3 Purposes, Dimensions and Forms of Abstraction

Abstraction, taking many forms and serving many purposes, can be considered from many points of view and structured in many ways. In this section different particular perspectives are adopted to address a different particular aspects or uses of abstraction.

3.1 Vertical Abstraction for Recognition and Theory-Building

It is common to speak of *levels* or *layers* of abstraction, suggesting a vertical dimension. A higher level of abstraction is characterised by the possession of concepts of larger granularity and greater power—intellectual constructs built upon the less powerful concepts of a lower level.

A *vertical abstraction* recognises that a subset of phenomena in the subject matter forms a cluster that is highly significant for some purpose. The recognition of any coherent entity—even the newborn baby's recognition of its mother—is an exercise of vertical abstraction. Where there are multiple instances of the recognised cluster, the clusters themselves become phenomena of a new class in our conceptual alphabet, distinct from the constituent lower level phenomena whose associations and relationships they embody. In Euclidean plane geometry the notion of a circle is a vertical abstraction: it is a plane figure bounded by a closed line such that there is one point of the figure—called the circle's *centre*—that is equidistant from every point on the line. The reward of a vertical abstraction is the richness of the associated theory. Many interesting theorems can be proved about plane figures constructed of circles and straight lines; and many useful terms can be defined, such as *diameter*, *radius*, *tangent*, and *semicircle*. The use of the circle concept, and of the associated terminology and theorems, increases the economy, and hence the potential power, of discourse in Euclidean geometry.

Vertical abstraction does not in itself imply either encapsulation or information hiding [Parnas72]: no part of the subject matter of the lower level becomes hidden in the higher level. In the example of the circle, the basic constituent phenomena of the circle abstraction are the centre point and the set of equidistant points at the lower level: these are not discarded in the abstraction, but remain visible and directly available, along with all other phenomena of the lower level, as participants in constructions and theorems applying to circles. The circle abstraction raises the level simply by introducing an additional fruitful concept that was not explicitly named and available at the lower level. Information hiding and encapsulation, by contrast, are disciplines for software development that allow programming abstractions to be devised and used independently of their implementation. They conceal the implementation—which could be considered the ground of the abstraction—by hiding it behind an impenetrable wall. Their purpose is to ensure that users of an abstraction are prevented both from damaging the detailed implementation and from illicitly exploiting any peculiar contingent properties it may have.

3.2 Horizontal Abstraction for Description

The commonest and simplest exercise of abstraction is the purposeful selection that is inherent in making any description. No new concepts or phenomena are introduced, and no new relationships among those already existing: the description is merely restricted to those selected as significant for the purpose in hand. In contrast to vertical abstraction we may call this *horizontal abstraction*. In terms of abstraction levels, a horizontal abstraction and its subject matter are on the same level.

Harry Beck's famous 1933 map of the London Underground system, appositely cited by Jeff Kramer in [Kramer07], was the product of a conscious horizontal abstraction. From 1889, earlier Underground maps [TubeMaps11] had shown the growing network of lines and stations superimposed on the background of a conventional street map of London. Over the following thirty years this background of streets became gradually fainter and less detailed in successive maps, and in 1920 it was abandoned altogether. But for the next ten or twelve years, even maps in which no streets were shown still placed the lines and stations with precise topographical accuracy, against a faintly depicted background of a few famous landmarks. Beck, perhaps influenced by his work as an electrical draughtsman, decided that this topographical accuracy was unimportant. His Underground lines ran straight up and down the map, or across it, or on 45° diagonals, and stations and lines were spaced for maximum clarity.

Beck had rightly distinguished two subsets of phenomena relevant to the Underground users. The first subset contains the phenomena of precise geographical location, which allowed users to identify the stations nearest to their destination and starting points and to estimate the journey distance as the crow flies. The second subset contains the phenomena of stations sequenced along each line and of line intersections at certain stations, which allowed users who already knew their destination and starting stations to plan a journey that minimised the number of intervening stations or the number of changes from line to line. In his abstraction, Beck judged the second purpose to be more important than the first: so he discarded the geographical phenomena to allow the journey phenomena to be shown in the clearest and most useful way. Planning a journey with his map usually proved very easy. The map was a brilliant success, and its design was copied for rail transport systems in many other countries. Harry Beck personally continued until 1960 to produce maps of the continually evolving network, and today's Underground maps still adhere to the basic principles of his design.

Horizontal abstraction of this kind is often applied to an instance—in this case, the London Underground system—rather than a class. Its product is a description of the subject matter. We might say that it is the purest form of abstraction because its essential effect is to discard some part of the subject matter: it adds no new concept to the existing repertoire, and in itself reveals no property and provides no associated theory that was not already available. Its aim and benefit is clarity and focus, achieved by discarding what is irrelevant to the particular purpose in hand.

3.3 Abstraction for Formal Analysis

By contrast, a more formal kind of abstraction can serve the specific purpose of demonstrating a desired property of its subject. This kind of abstraction has been of the greatest importance in the development of programming, allowing proof that a program satisfies its specification.

In a talk [Turing49] given in Cambridge in 1949 Alan Turing used a flowchart to explain a program and prove it correct. Given *n*, the program computed *n!* on a machine without a hardware multiplier. The flowchart served very well as a bridge between the machine-code program and the computation it was intended to perform: it provided a common intelligible abstraction both of the list of machine instructions and of the arithmetic calculation. Turing recognised that the flowchart abstracted from the machine-code program, observing that he could not show the 'routine for this process in full'. That is, he could not show the actual list of machine instructions, because 'there is no coding system sufficiently generally known': every computer of the time had its own unique order code. So the flow diagram would have to serve as a substitute: it was a grounded abstraction of the specific pattern of machine behaviour evoked by the program.

The elements of the computation were abstracted in the flowchart by a simple notation. Turing showed the allocation of the program's variables *s*, *r*, *n*, *u* and *v* to the machine's storage locations 27 through 31 respectively; and he used the variable names in simple arithmetic operations—for example "*r:=1; u:=1*" and "*test s>r*"—in the process and decision nodes. The theme of the talk was program correctness: "How can one check a routine in the sense of making sure that it is right?" With each significant process node of the flowchart he associated an assertion of the program variable values held in each storage location on entry to the node and on exit to each possible successor node. The assertions at the program's entry and exit nodes constituted a program specification in the form of a precondition and postcondition pair: if all assertions on the intermediate nodes hold separately, then when the machine halts this specification is satisfied. (Turing used a separate argument to show that the machine must indeed halt.)

Like Beck's map, Turing's flowchart provided clarity and focus, but additionally it provided a basis for formal proof of the program's correctness. In principle the same proof could have been applied to the machine code listing by regarding the machine state as consisting of its storage locations and registers together with the program counter. In practice the proof would then have been considerably complicated by the distracting intricacies of the machine code instructions. Of course the flowchart abstraction, while grounded in the machine code program, was not derived from it *de novo*. In some form it was no doubt in Turing's mind, and possibly already on paper, when the machine code program was written. So while the arithmetic operations can be regarded as vertical abstractions firmly grounded in the corresponding sequences of machine code operations, they may also have served as named design elements from which segments of the machine code program were refined.

3.4 Multiple Horizontal Abstractions

The form of horizontal abstraction so effectively practised by Harry Beck becomes more interesting and significant when multiple horizontal abstractions are made of the same subject matter and must be reconciled in some way. Are the different abstractions compatible? That is: is there a reality of which they are all faithful abstractions?

When two abstractions are fully grounded in the same existent physical reality, as Beck's abstractions were, the question of their compatibility comes down to the fidelity of the abstractions to the reality. If both abstractions are simultaneously sufficiently faithful to their subject, the physical existence of their common physical ground is an incontrovertible proof of their compatibility. However, compatibility in the physical reality may be implicitly conditional on a separation of the contexts of the different abstractions. Both of two horizontal abstractions may be true in the reality, but not at the same time. In this sense two horizontal abstractions may not be *compositional*. That is: the combination of the two abstractions may not possess all the properties of each abstraction considered in isolation.

Horizontal abstraction can capture a separation of the subject matter into *contexts* distinguished by purpose and circumstance. An identified context becomes an exclusive focus of attention in its own right. Contexts are not disjoint: some phenomena of the subject matter will be relevant to more than one context, and common phenomena will play distinct roles in distinct contexts. Program slicing is one example of horizontal abstraction.

Another example of horizontal abstraction, drawn from everyday life, is the separation of our experience and behaviour into distinct social contexts. An adult may separate working life from family life; a child may separate school from home. Different contexts bring with them different experiences, different purposes, and different expectations. To a great extent, we hope and expect that the different contexts will be neatly separated in time. This temporal separation is valuable. For practical reasons, and for emotional and intellectual comfort, we want to neglect what is irrelevant in the current context: it can only complicate matters. The discomfort felt by a child—and often not only the child—when parents appear at school, or when a teacher from school appears as a guest at the family dinner table, reflects the difficulties of simultaneously managing distinct contexts that the child had expected to keep apart. The different roles that the participants are expected to play in the different contexts are hard to reconcile.

The child's embarrassment precociously demonstrates an awareness of an intellectual problem that besets software development: horizontal abstraction aims to separate contexts and concerns, but the separation can rarely be perfect. In realistic computer-based systems the horizontal dimension of abstraction is fundamental, and often there is no temporal separation. Several horizontal abstractions must coexist in time, because the multiplicity of system features and functions, and of normal and exceptional conditions and modes of operation, does not fall into any neat hierarchical temporal structure. The structure of system functionality is more like a colour separation than it is like an assembly of parts. Horizontal abstractions—albeit imperfectly separated—are essential. We will return to this problem in the following main section.

3.5 Program Design: Top-Down and Bottom-Up

Program design, as widely advocated in the 1960s and 1970s, aimed at developing a hierarchical structure of abstractions to bridge the gap between the computational abstractions offered by the programming language and the abstractions most clearly associated with the problem to be solved. This gap appeared to be self-evidently a vertical gap: so the abstractions sought were vertical abstractions, each layer populated by abstractions of elements of the layer below it. Naturally the question arose: should design proceed from the top downwards, from the bottom upwards, or by some mixture of the two? Clarifying the distinction more precisely in terms of abstraction: top-down design begins with an abstraction and works to provide successively lower-level subjects in which it can be directly and indirectly grounded, while bottom-up begins with the ground and forms successively higher-level vertical abstractions until a top-level 'solve-the-whole-problem' abstraction has been constructed.

A kind of consensus arose almost immediately that top-down design—essentially some form of stepwise refinement—was the right choice. This consensus was strongly influenced by work on the design of small terminating programs. The specifications of many of these programs were very simple: the complete specification could be tersely and exactly expressed in a precondition and postcondition on the program variables. The classic problem corpus included: sorting an integer array; finding the greatest common divisor of two integers by Euclid's algorithm; building and maintaining a balanced tree; printing the first thousand primes; placing eight queens on the chessboard so that none is attacking any other; finding the convex hull of a set of points in three-space; and other similar problems. Brilliant design techniques were invented and described, and some brilliant solutions were found to problems in the repertoire.

In this setting the top-down consensus has an obvious appeal. The starting point of each program development is one simple abstraction: it is a terse formal specification that can form the root of a refinement tree, giving the developer a clearly identified anchor at the top end of the vertical gap. Certainly there may be choices to be made at the early steps, both in formalising the specification and in selecting the solution algorithm; but for a small program there are usually few candidate choices, and once each choice has been made it provides a firm anchor for the next refinement step.

The bottom end of the vertical gap is furnished by the defined elements and built-in structures of the programming language. This end is weakly structured with respect to the problem: the eventual need to form an assemblage of programming language statements rarely constrains the program specification at the root. Of course, this weak structuring is an advantage: it springs from, and reflects, the huge versatility of the programming language and its purposeful flexibility in realising almost any procedural abstraction. Developers who begin at the top end of the vertical gap know their starting point, and have the programming language readily in mind; they can feel confident that few or no barriers will emerge as they near their goal of an executable program. Developers who begin at the bottom have apparently more difficulty. The programming language level itself offers little or no obvious help in suggesting abstractions that can form the next level above; and often it is hard to determine when good progress is being made towards the top-level goal. For the top-down programmer the starting point is known, and there are many acceptable final destinations: for the bottom-up programmer it is harder to know where to start, but only one destination is acceptable.

Two remarks are in order about top-down and bottom-up design. First, the physicist Richard Feynman, in his personal observations about the Challenger disaster [Feynman86] castigated the top-down approach to the development of the shuttle engine, and, by implication, to system development in general. Along with other comments, he wrote "A further disadvantage of the top-down method is that, if an understanding of a fault is obtained, a simple fix, such as a new shape for the turbine housing, may be impossible to implement without a redesign of the entire engine." Second, it must be said that a rigorous distinction between top-down and bottom-up design is misconceived. Neither approach is feasible unless the designer looks far ahead in the design process, even to the extent of forming a clear mental conception of the complete program at the outset. What then proceeds in one direction or the other is the overt written representation of the levels of the program as already conceived and of their exact details.

3.6 Program Design: Horizontal Abstractions

The virtues and benefits of top-down development depend crucially on the simplicity of the problem to be solved, characterised by a complete program specification in the form of one terse simple abstraction. The refinement structure by a progression of abstractions can then be a tree, in which each link between neighbouring steps is in the vertical direction.

When the program specification is not so simple, horizontal abstraction becomes inescapable from the outset. Discussing the development of programs from specifications, Burstall and Goguen wrote [Goguen+80]:

> "...the process of implementing a large program from its specification has a two-dimensional structure. One dimension of structure, the horizontal, corresponds to the structure of the specification. The second dimension, the vertical, corresponds to the sequence of successive refinements of the specification into actual code; the specification is at the top, and the code is at the bottom."

The 'structure of the specification'—the horizontal dimension—arises from the richness of parallel concerns that may often be required in a large program. This richness must be addressed by multiple horizontal abstractions. Arising in one program, they are unlikely to be disjoint: like the embarrassed school child, the program designer may find it hard to reconcile them in one coherent design. When horizontal abstraction appears at any refinement node it threatens to complicate the whole subhierarchy rooted at that node.

It may be thought that a need for horizontal abstraction rarely appears in a small program, but this is not so. The design of any program that processes sequential input and output streams—of records, messages, or even single characters—can benefit greatly from appropriate horizontal abstractions. A suitable program design method for such programs, based on horizontal abstraction, was described by the present author [JacksonM75, JacksonM76]. Each input and output stream is the subject of a horizontal abstraction in which its sequential structure is described as a labelled regular expression. The program structure is then formed by composing these abstractions according to the correspondences among the parts of the streams at several levels. If the criteria for a satisfactory composition cannot be satisfied the design method invites a further horizontal composition in which the production of outputs from inputs is split into two or more parts interacting by appropriately chosen intermediate streams introduced for the purpose. This further horizontal abstraction—in effect, a pipe-and-filter design—closely parallels the introduction of intermediate vertical abstraction levels in top-down or bottom-up design.

In computer-based systems the need for horizontal abstraction is ubiquitous, and is exacerbated there by other considerations peculiar to those systems. We will return to this topic in the following main section.

3.7 Symbolic and Analogical Models

Russell Ackoff distinguishes [Ackoff62] three kinds of model: iconic, analogue and symbolic. An iconic model, as its name suggests, represents properties of its subject visually, as in a photograph or map. An iconic model is primarily a concrete thing whose properties can be visually inspected—for example, a portrait sculpture or a three-dimensional map; but it may also be stored in an intangible form—for example, as the bit pattern of a digital map or photograph—from which the concrete icon can be reconstructed for visual inspection.

A symbolic model is abstract: it represents the subject's properties in symbols, perhaps in a set of equations, a mathematical relation, or a logical formula. A formal symbolic model is intended to support formal reasoning about the subject. A symbolic model can, of course, be represented concretely, but the physical attributes of the representation are of no significance whatsoever. For example, a state machine can be drawn as a diagram, but the sizes and dispositions of the symbols on the page are not significant.

An analogue model is in principle concrete: as its name suggests, it represents properties of its subject by analogy with its own properties. For example, a hydraulic system may serve as an analogue model of an electrical system: the pipes are analogous to wires, and the flow of water to the flow of current. Like an iconic model an analogue model is primarily a concrete thing, but may be represented intangibly. For example, an assemblage of programming objects may constitute an analogue model, represented in the programming language; but, strictly speaking, the assemblage fulfils its function as an analogue model only when the program is executed.

Some models may combine iconic and analogue characteristics. Two dimensional maps, for example, often represent elevation by coloured contours whose gradation from green to red through brown represents increasing elevation by analogy. A photograph necessarily has properties such as the proportion and relative position of the subject's features that are clearly analogues of the reality. For this reason, the distinction between iconic and analogue models is blurred. It is comparatively unimportant for our purpose here, and we will neglect it, referring to both as *analogical* models. Symbolic models are more clearly distinguished. In general, a symbolic model is a pure abstraction of its subject: nothing in a symbolic model need look like the modelled subject or be in any other way analogous to it. This why the exact meaning of the model can be preserved under suitable symbolic manipulations that are useful in reasoning.

Where the purpose of software is to mechanise calculation, as it was in the earliest days of electronic computers, symbolic models and their known manipulation methods are the essential basis of program design. The earliest typical users were mathematicians and scientists. They understood programming as the task of presenting a symbolic mathematical model to the computer in a form in which the computer could solve the equations and evaluate the formulae of the model. The first high-level programming language, Fortran, was so named because it was conceived as a *formula translator*.

When the purpose of software is expanded to involve a more direct relationship with the human and physical world it becomes necessary for programs to embody data structures that can be used as mutable models, capturing the changing states and behaviours of their subject matter more directly. Examples of such mutable models include databases, assemblages of objects, process networks, and general data structures of the programming language. All these can be analogical models.

Unlike a symbolic model, an analogical software model is not a pure abstraction of its subject. It is, in effect, a material domain in its own right, albeit a domain enclosed within the boundary of the computer. This distinction is important, because the model has its own material properties that are distinct from those of the modelled domain, and *vice versa*. The relationship between an analogical model and its subject is therefore more complex than the relationship between a symbolic model and its subject. The two relationships are depicted in Figure 1.

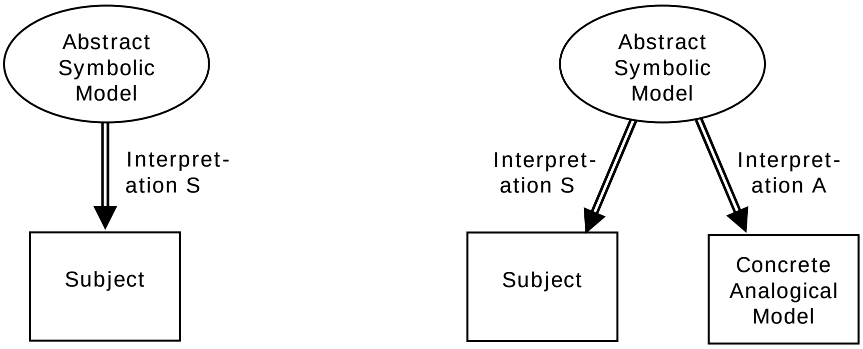

Figure 1: Symbolic and Analogical Models

On the left is a symbolic model and its subject. The interpretation S maps the terms used in the symbolic model to the parts of the subject. In practice the subject names are often used in the abstraction, and the interpretation is then implicit; but in general an interpretation is required to map the terms used in the model to the parts of the subject. On the right is an analogical model and its subject. The interpretation S maps the terms used in the symbolic model to the parts of the subject, while the interpretation A maps the same terms to the parts of the analogical model. The analogy is mediated by the common abstraction captured in the symbolic model.

In common software development practice the symbolic model is often—even usually—bypassed, and the analogical model is treated as if it were itself a symbolic model of the subject. This apparently attractive intellectual shortcut exacts a penalty. As always, the subject has properties that are not captured in the abstract symbolic model; but an analogical model—unlike a symbolic model—also has its own concrete properties that are not captured in the symbolic model. These properties may correspond, roughly or exactly, to properties of the subject, or they may simply be artifacts necessitated by the programming language or designed for efficient computation. For example, a database forming an analogical model may have such properties such as indexing, ordering of tuples in tables, deletion of inactive tuples, creation of new tuples, and the use of a null value in a field when no correct value has been entered into the system. A database null value, for example, may represent incomplete data entry; but it may also represent the value *undefined* in a three-valued logic. When the symbolic model is ignored the significance of these additional properties becomes a source of difficulty and confusion. The resulting analogical models embody ad hoc expedients or demand obscure workarounds "often making it difficult for uninitiated readers of the model to understand which aspects of the model are meant to be accurate reflections of the problem domain and which are just accidental properties of the particular workaround" [Atkinson+08].

3.8 Formal Abstractions of a Non-Formal Reality

Formal abstractions provide the most reliable domains of reasoning that are available to us. Disregarding Gödel we are able to reason with great confidence in many formal systems, relying on proven theorems to calculate about instances; mathematicians or logicians may also add to the corpus of proven theorems. We have no fear that someone will one day discover that the set of primes is finite, or find some particular triangle in the Euclidean plane the sum of whose interior angles is not π. Our purpose in constructing formal abstractions of non-formal realities is to impart a similar degree of confidence to our reasoning about those realities.

However, there is a severe limitation on the confidence that can be achieved. A properly constructed formal system is bounded by its defining set of axioms. But a non-formal reality such as the natural, human and engineered world is bounded by no defining set of axioms—at least, not at the scales of interest for practical life in general and software development in particular. To formalise is inevitably to place a bound on the phenomena, attributes, relationships and possible behaviours that we are prepared to consider. Unfortunately, those that we have oversimplified, ignored or neglected may prove relevant enough to invalidate our formal abstraction. More generally, any formal abstraction of a non-formal subject matter is at best an approximation to the reality in which it is grounded. The unpalatable consequence is that formal reasoning in a non-formal setting can signal the presence of error but cannot prove its absence. The abstractions that suffice for the premiss may fail for the logically sound conclusion: it must be tested by interpretation in the ground reality.

The history of the traditional engineering branches abounds in examples of failed abstractions. Leonid Moisseiff, the designer of the Tacoma Narrows bridge, gave careful consideration to the possibility of wind-induced horizontal oscillation of the roadway; but he neglected the vertical oscillation, which destroyed the bridge a few months after it opened in 1940. The designers of the de Havilland DH106 Comet 1 jet airliner were careful to consider all sources of stress on the aircraft's fuselage. They made separate and extensive calculations and tests to ensure that the fuselage would not fail when subjected either to the torsional stresses of flight or to the stresses of the compression and decompression necessitated by the great height at which the aircraft was designed to fly. Unfortunately, it was the untested simultaneous combination of both sources of stress which caused metal fatigue, spreading from stress singularities sited at the corners of the aircraft's square windows. The consequent rapid spread of metal fatigue in the structure caused several aircraft to disintegrate in mid-air. In another famous engineering failure in 1978 [Hartford11], the roof of the Civic Center in Hartford Connecticut collapsed. The roof structure was a space frame design that had been formally validated by what was then a state-of-the-art computer analysis. However, the abstraction on which the analysis was based took no account of certain aspects of the proposed design. It ignored the special cases at the outer boundaries of the roof, and also the possibility of failure by buckling of certain critical members. Worse—and very relevant to software development—when the space frame proved hard to assemble on the ground before being hoisted into its final position, the subcontractor for the roof overcame the difficulty by some seemingly minor changes to the joints between certain members. These minor changes had an unpredicted major effect on the properties of the structure: the roof collapsed when loaded by an unexceptional fall of snow.

Even in the most modest abstraction there is scope for failure. Harry Beck's Underground map is an analogical model. It is not itself a formal abstraction, but nonetheless it illustrates some of the difficulties of even a partial formalisation of a non-formal reality. It seems obvious that the Underground system consists of stations connected by tracks. But immediately the notion of *station* confronts us with a difficulty. What, exactly, is a station? At first sight Edgware Road is a station; but on closer inspection it appears to be two stations: to change trains there requires leaving the Underground system, walking 150 yards along the street and re-entering the Underground system at 'the other Edgware Road'. Bank and Monument are distinct named stations; but they have been linked into one since they were originally built in the nineteenth century.

Anomalies and exceptions of this kind are characteristic of non-formal realities. They arise from the interpretation of formal terms, which are the very foundation of any abstraction: what counts as a station? They arise also from peculiar corner cases of reality that present counterexamples to almost any satisfyingly elegant theory associated with the chosen abstraction. They arise from the difficulties of constructing a physical reality to conform faithfully to its design abstraction. They arise from the untoward combination in reality of two horizontal abstractions such as the two stress analyses of the Comet aircraft.

These difficulties arise for an engineered system like the London Underground, and even more strongly for the complex and arbitrary organisational, commercial, manufacturing, legal, and fiscal realities that make up the problem world for many large systems. In a non-formal world, clean abstraction for a purpose presents a discouraging dilemma. The cleaner the abstraction the more anomalies will emerge to reduce its fidelity to the reality from which it sprang and which it purports to describe.

3.9 Abstractions and Monsters

Anomalies and counterexamples, characteristic of abstraction in non-formal domains, may be encountered also in dealing with a formal mathematical domain. Lakatos recounts [Lakatos76] the mathematical history of Euler's formula V – E + F = 2 relating the numbers of Vertices, Edges and Faces of a polyhedron. He introduces a series of solid figures, proposed as counterexamples to the conjectured theorem implicit in the formula. Successively he introduces: a hollow cube whose interior faces form a smaller cube; a 'crested cube' having a smaller cube projecting from the middle of one of its faces; a pair of tetrahedra sharing only one edge; a pair of tetrahedra sharing only one vertex; a 'star polyhedron'; and several others.

A central theme in the discussion is 'monster-barring'. Some mathematicians, to preserve the conjecture, rejected putative counterexamples as 'monsters'—solid figures, but not polyhedra. Challenged to define *polyhedron*, they defended the formula by producing over time a sequence of successive definitions carefully modified to exclude each successive proposed monster. With the hindsight of the history of Euler's formula we may fault the mathematicians for being insufficiently careful in formalising the polyhedron abstraction. But in dealing with the phenomena of the physical and human world at the granularity of interest to engineers and software engineers, the existence of monsters, challenging any proposed abstraction and its associated theory, is undeniable: for every definition there are hard cases somewhere in the world; and for every theorem there are counterexamples.

How, then, can we proceed in practice if our abstractions and theories are so fragile? How can we avoid the monsters that lie in wait for us? We can adopt various strategies according to our purposes and circumstances. For example:

- We may simply accept some degree of approximation and a consequent level of imperfection and even failure. The anomalies of certain *stations* will inconvenience some Underground passengers; but not many passengers, and not often.
- We may rethink or elaborate the abstraction to preserve the associated theory, in the spirit of the successive definitions of *polyhedron*. The efforts of mathematicians to preserve Euler's conjecture

against the challenge of successive monsters find some echo in the energetic efforts of engineers to eliminate the risks of successively identified sources of failure in such safety-critical artifacts as cars, power stations and aeroplanes. National tax authorities work hard to close the successive fiscal loopholes found by ingenious lawyers and accountants.

- We may supplement the abstraction and the implementation of the theory by allowing an overriding human judgment for the hard cases. This is the usual practice in a good legal system. The law relies on abstractions of many kinds of human behaviour, and stipulates rules, rights, obligations, and penalties in terms of those abstractions. When an abstraction or theory breaks down because it is not clear how it should apply to the facts of a particular case, the matter is settled by the judgment of a court.
- We may apply the abstraction and the accompanying theory only in a strictly limited context in which we are confident that no monsters are to be found. In the context of all human life the commonplace abstraction *chair* is very problematical. Is a bar stool a chair? A bean bag? The seat of a swing? A bicycle saddle? A park bench? The T-bar of a ski lift? But in the context of the sales system of a furniture factory with a narrow product range it may work perfectly.

Strategies of these kinds are essential in the development of computer-based systems. The last strategy—context restriction—is particularly important: developers of a computer-based system must not aim at an unattainable and pointless universality.

4 ABSTRACTION IN COMPUTER-BASED SYSTEMS

In a computer-based system the computer monitors and controls parts of the human, physical and engineered world outside itself, evoking and imposing some required behaviours in that world. This is its purpose: its own internal behaviour, and its participatory behaviour in its direct interactions with the world, are merely the means to that end [JacksonM02].

Developers of such systems cannot restrict their focus to the clean and formal computational structures quarantined within the bounds of aseptic program texts executed by a machine that reliably implements a formal instruction set. In such developments abstraction remains a vital intellectual tool; but its power is comparatively diminished in relation to the overall task. Of course, most realistic system development projects involve subsidiary tasks of formal specification and program design; but these tasks are overshadowed by larger concerns. The exercise of abstraction in system development demands more judgment, more scepticism, more insight, more versatility, and more hard work. For example, the non-formal nature of the problem world calls almost every proposed abstraction into question. In a formal world, after the instruction sequence

“*x := P; y := x; x := y*”,

the condition “*x = P”* will certainly hold. But in the material world, after the instruction sequence

“*x := P; Arm.moveTo(x); x := Arm.currentPosition*”,

the condition “*x = P”* may not hold. Moving the arm and sensing its position both involve state changes in the problem world. Movement of the arm will be imprecise, and the resulting position will be imperfectly sensed by the computer and further approximated by a floating-point number. This kind of difficulty is exacerbated by the multitude of potentially conflicting system requirements to be satisfied by a realistic system and by the further complexities they add at almost every level.

This section discusses some aspects of the use of abstraction in the particular context of computer-based systems development. First some of the salient characteristics of computer-based systems are identified and briefly discussed. Then, in the remaining subsections, some of their implications for the use of abstraction are explored.

4.1 Characteristics of Computer-Based Systems

Computer-based systems are very various. Few characteristics are common to them all, and few of them exhibit all the characteristics identified here; but wherever these characteristics are found they are likely to complicate the use of abstraction in system development in some way and diminish its utility.

Many people and organisations—the ‘stakeholders’ in the accepted jargon of requirements engineering—may be in some way involved in the system operation: their various needs must therefore contribute to determining the system requirements. For example, a radiation therapy system involves therapists, patients, oncologists, radiologists, medical physicists and maintenance engineers: the behaviour of the system must allow each of them to play their part successfully. If the system is safety-critical, a safety authority is also a stakeholder, demanding that the developers show convincingly that the system is acceptably safe.

The problem world of the system is likely to be a heterogeneous assemblage of *problem domains*. Some are mechatronic devices engineered to published specifications. Some are human participants, ranging from carefully selected and highly trained operators such as aircraft pilots and train drivers to uninformed customers, randomly selected casual users, and medical patients. Some are parts of the natural or built environment such as airport runways, a tanker terminal, the earth's atmosphere at various elevations, or a network of railway tracks. Some are existing systems such as the internet, the telephone system or the Global Positioning System. Each problem domain has its own given properties, which the system must respect and exploit and the developers must therefore understand and analyse.

One system has many features. For example, an automotive system may have driving assistance features such as electronic suspension control, start-stop, anti-skid braking, cruise control, maximum speed regulation, lane departure warning, and automatic parking. These features can interact by requirement conflict: in some circumstances two features may require contradictory behaviours. They can also interact through common problem domains that are not explicitly mentioned in the individual features' requirements: for example, by imposing excessive demands on engine or battery power. In general, the relationship between problem domains and the behavioural features in which they participate is many-to-many.

The system may have multiple modes of operation demanding different functional behaviour. The behaviours of a radiotherapy system must include, for example: acquiring and validating an oncologist's prescription for a patient; determining the position of the patient for a first treatment according to the prescription; repeating a previously determined position for a subsequent treatment; managing the radiation dose in a treatment; initial setup and calibration; and operation under control of a maintenance engineer.

Inescapably, any system is designed to function in a restricted context that affords certain assumptions. For a motor car, for example, explicit context restrictions may specify such obvious factors as fuel, ambient temperature, tyre pressures, regular oil changes, and so on. Other context restrictions may be implicit: few car manuals state explicitly that the car will not function under water or on the moon, cannot be satisfactorily driven on sand dunes, will not climb a 1-in-1 gradient, and will not carry a load of ten tons. Many context restrictions are left to be understood by common sense, or expressed in such phrases as 'normal use'.

Context is important because both the requirements and the given problem domain properties depend, in general, on the context of system operation. Natural phenomena may sometimes exhibit unexpected behaviour such as a gale or a tsunami. Engineered devices may wear out and cease to function as specified. The assumed bounds of human operators' behaviour may be exceeded—for example, input speed may increase with long familiarity with the system. The maximum permitted length of a railway train may increase over time, changing its relationship to the length of a track segment. Different contexts are likely to demand different abstractions.

The operational context and its accompanying assumptions are not global to the system. A critical system must be designed to take proper account of variations in context: the operational context at any point in time is a set of overlapping current subcontexts. Fault-tolerance is merely one particular example: the fault presents a subcontext for which the system has a specified behaviour, perhaps providing a degraded functionality or shutting the system down completely according to the severity of the fault. For a passenger lift system, the presence of a fire in the building presents another subcontext: the system must provide a form of lift service specifically designed for the needs of the firefighters. Obviously, the fault and fire subcontexts are not mutually exclusive. An automotive system must behave in different ways when the car is in normal use, when it has been involved in a collision, and when it is in a repair workshop. The more critical the system the more various and extreme the subcontexts in which it is required to behave in specified safe ways. The design of a nuclear power plant was recently criticised for failing when a tsunami and an earthquake of magnitude 8.9—both rare events—occurred simultaneously.

4.2 Abstractions and Purposes

We make an abstraction to serve a purpose, and its value and success are to be judged by how well they serve that purpose. Developing a computer-based system is a task in which many different purposes must be pursued, and it follows that many different abstractions will be needed. This profusion of purposes is, in general, not found in the development of small programs whose subject matter is a mathematical abstraction. For these programs the given properties of the subject matter—or problem world—are well known, or can be reliably learned by consulting a mathematical text. The developer can exploit different aspects of the given properties by relying on a corpus of proven theorems. Essentially, the only abstraction to be invented is an abstraction of the program behaviour in terms suited to the chosen programming language.

For a computer-based system, by contrast, it is necessary to capture, at a suitable level of abstraction, the relevant given properties of each problem domain, perhaps as they vary with the operational context. Even where a problem domain is a device engineered to an explicit specification, there will still be a need to identify the properties on which the system can rely in each of its various operational contexts and modes. This is, evidently, a task of abstracting for description. In some cases the task may be essentially one of selection, presenting itself as horizontal abstraction. For example, it may be very useful to make separate abstractions of faulty and fault-free behaviour: the faulty behaviour is significant for fault detection and diagnosis, while the fault-free behaviour is significant for normal operation. In other cases description may involve vertical abstraction, abstracting and analysing a higher-level behavioural property from concrete lower-level behaviour. In neither case can the abstraction be adequately treated top-down—that is, as a task of refinement rather than abstraction. Refinement is a process of inventing and constructing something new, not a process of describing an given existing reality.

Invention and construction is the process for developing the system's behaviour—or, more properly—its many behaviours. Once a sufficient set of given problem domain properties has been captured by a process of abstraction, it may become practicable to devise a desired system behaviour from the top down, just as it may be possible to design a program from the top down once the programming language elements are known. However, there is a crucial side condition. Strict top-down development is feasible only when the required behaviour can be tersely specified to define the goal of the first refinement step. For the overall behaviour of a computer-based system this is rarely—perhaps never—possible. The overall behaviour is an assemblage of several functional behaviours that must come into play in response to changing operational circumstances and potentially unpredictable user demands. This overall behaviour as a whole cannot be usefully abstracted to give an effective starting point for refinement. Instead the individual functional behaviours may be separately designed, each taking explicit account of the subcontexts in which it is required. Recombining these separately designed behaviours becomes a further development task.

A third purpose, along with description and construction, is analysis. Given an existing reality—whether the result of description or of construction—analysis makes an abstraction with the purpose of validating a claim that the reality possesses some desired property or exhibits some desired behaviour. If the abstraction and the validation process are formal the validation may proceed by proof or model-checking, and is then usually called verification, the term verification connoting a degree of confidence associated with a mathematical demonstration. The use of this term is fully appropriate for the mathematical demonstration itself; but, of course, it is quite inappropriate to the question whether the formal abstraction corresponds faithfully to the reality in which it is ultimately grounded and in which the purpose of the system is located.

A fourth purpose arises from the multifarious nature of the stakeholders and their requirements. In a computer-based system, many requirements are characterised as 'non-functional'. A notable example is usability. This requirement is in fact purely functional in the sense that its satisfaction or non-satisfaction can be judged by observing the functional behaviour of the system—including, of course, its users, operators and other human participants. But although cognitive and ergonomic research has much to say about usability, the judgment still cannot be made by the developers themselves alone: the stakeholders or their legitimate representatives must play a decisive part. When developers design the whole system behaviour they must therefore make abstractions of that behaviour that capture the associated participating behaviours of humans in their various roles. The stakeholders must validate their proposed participation by their assent based on a full comprehension. According to the criticality of the system, and the nature of the participation and the stakeholders involved, this comprehension may be achieved by examining a symbolic abstraction such as a state machine, by viewing an animation, by interacting with a prototype implementation, or by other means.

For a realistic computer-based system, the multiplicity of purposes which abstraction can serve, together with the richness of the system functionality, properties and behaviours, makes it clear that many abstractions are necessary to support and embody the development process. In the following two subsections two particular abstractions, each an example of a widely used class, are discussed. The purpose of each one is briefly explained and some of its virtues and limitations are identified. Advocates of each will no doubt be able to enlarge the list of its virtues. The limitations, it must be understood, are not presented here as culpable defects: they merely emphasise the truth that one abstraction alone cannot suffice.

4.3 An Example Abstraction: Event-B

The Event-B refinement method [Abrial10] is based on a formal abstraction of system behaviour. The system has a global *state* that is modified by *events*. It shares this fundamental abstraction with other development methods, including Z [Woodcock96] and VDM [Jones90]. Events have arguments denoting elements of the state, and are guarded by predicates on the state. Some predicates on the state are defined as *invariants* that hold in every state. The system is consistent if all invariants hold for all possible sequences of events. The purpose of this abstraction is to support a development discipline. Starting from a very abstract model, capturing an initial understanding of the problem domains and the requirements to be satisfied, the development proceeds by successive refinement steps. In each step a more detailed model is constructed and proved to be a *refinement* of the preceding model: that is, while adding detail it preserves the invariants and other properties of the more abstract model.

Formal reasoning with this abstraction is very tractable. To prove that a model is consistent and refines its more abstract predecessor it is necessary to complete many small proofs, not all of them trivial. The chosen abstraction of system behaviour often allows most of these proofs to be performed automatically by specialised software tools, leaving relatively few proofs to be devised and carried through by hand. This is a large benefit.

Any abstraction has two faces: what is included, and what is discarded. The Event-B abstraction discards many phenomena and considerations that are significant for some development purposes. In particular:

- Different parts or domains of the system are not distinguished. No distinction is made between events occurring in the computer and events occurring in a problem domain, or between the computer's internal states and internal states of the problem world. These are important distinctions in the practical utility of the system. Without them, it is, for example, impossible to address a possible divergence between a problem domain state and the state of its analogical model in the computer.
- Causality is ignored. It appears in the model only in the association of an action with an event: each occurrence of the event may be imagined to cause the action. Since the initiator of the action is not identified this association does not capture causal relationships. Such relationships are essential. To understand how the system works, and to demonstrate that it will work reliably, it is necessary to trace the causal chains that define its proper working, and to consider the possibilities of failure in each link of each chain.
- An invariant may represent a requirement—for example, "an employee is never in a room for which the employee does not hold an access authorisation;" or it may represent a given or assumed property of a problem domain—for example, "no pair of rooms is connected by more than one door," or "a train can move out of a track segment only to an adjacent segment." A requirement can be modified by agreement with the stakeholder; a given domain property can be modified only by a change in the physical world.
- The context of system behaviour is assumed to be uniform. However, different behaviours are required in different contexts. For example: "two trains never occupy the same track segment" is true for train journeys but false when a train is being assembled in preparation for a journey or has broken down and is to be towed to a repair shop. The distinction between different contexts can

be represented only by adding the context as a state element and conjoining a predicate on its value to the affected invariants and event guards. This representation would be very cumbersome and error-prone.

- Sequential behaviours cannot be directly represented in Event-B. Sequencing can be captured only by a relationship between the changing system state and the event guards. Sometimes this relationship can be defined in terms of state elements clearly associated with problem world states; sometimes it requires the introduction of a variable that is, in effect, a partial representation of the text pointer of the sequential process. This very indirect *ad hoc* approach fragments a sequential behaviour and may destroy its unity and human intelligibility.

4.4 An Example Abstraction: Use Cases

A completely different, informal, abstraction of system behaviour is implicit in the widely practised technique of *Use Cases* [Kruchten99]. A use case is an episode of interaction between an actor—typically a human user—and the system; for purposes of the use case the actor is regarded not as a part of the system but as an external agent. The episode of interaction delivers some result of value to the actor: for example, the actor succeeds in booking a theatre seat, or in drawing cash from an ATM. A use case is described informally as an interactive sequential process. The process may have many variations. For example: the theatre may be fully booked; the user may decide not to accept any of the available seats; the ATM cash may be exhausted; the user may fail to enter the correct PIN for the card inserted; the user's account balance may be insufficient; the process may time out; and so on. To accommodate common subprocesses such as logging into the system, validating the card inserted, or paying by credit card, use cases may be structured to embody or invoke other use cases.

Use Cases are often understood as the central—sometimes the only—vehicle for describing required system behaviour. Philippe Kruchten writes [Kruchten99]:

"The use-case model is a model of the system's intended functions and its environment, and it serves as a contract between the customer and the developers. It comprises the set of all use cases for the system, together with the set of all actors, so that all functionality of the system is covered."

The value of use cases is obvious. They describe the experiences that the system must afford to its users when they avail themselves of its various user-initiated functions, and allow the developers to design those experiences for users' convenience and satisfaction.

The use case abstraction, like an Event-B model, discards some significant phenomena and considerations:

- Some system behaviour is evoked not by immediate user interaction but by a change in system state—for example, by a change in the relationship between outstanding orders and stock-in-hand for a product. Such behaviours are not easily or fruitfully described in terms of delivering a result of value to a user.
- The role of user does not accommodate other important roles that a person may play. For example, the behaviour and needs of the recipient of a heart pacemaker are not exactly those of a user. Rather, the recipient's cardiac behaviour is the subject of monitoring and control by the embedded computer: the recipient is scarcely more a 'user' of the pacemaker than the patient in a surgical operation is a 'user' of the operating theatre.
- The fragmentation of user behaviour into use case episodes works well when each use case can be regarded as an independent episode rather than as a contribution to a larger purpose that persists across distinct use case instances. This assumption largely holds for a telephone system, in which each use case can be largely understood in isolation; but it does not hold for the driver-assistance functionality of a car or for patient treatment by a radiation therapy machine.

Discarding significant aspects of a problem is not in itself a fault. On the contrary, it is essential to separation of concerns. It becomes a fault only when the abstraction in question is regarded as the only abstraction necessary for development.

4.5 Representation and Comprehension

The representation chosen for an abstraction plays a large part in its comprehensibility. Since programming and system development are essentially human intellectual activities they can be carried out most effectively when their content is thoroughly understood by the people involved. The developers must understand the abstractions they construct, and the stakeholders must understand the content of the abstractions to which they are asked to assent. An abstraction can be represented in more than one way. Whether it is comprehensible depends not only on its formal content but also—vitally—on its representation.

To take a well-worn example: a state machine can be represented by a diagram or, equivalently, by a list of nodes and transitions. The fragmented list is well suited to processing by computer, but the diagram is absolutely essential for human comprehensibility. The distinction runs deep. As the mathematician Henri Poincaré wrote [Poincaré08]:

> "When the logician has resolved each demonstration into a host of elementary operations, all of them correct, he will not yet be in possession of the whole reality; that indefinable something that constitutes the unity of the demonstration will still escape him completely."

It is a major misfortune for software and system development that a fragmentary form of an abstraction—a list of nodes and transitions, or a collection of elementary operations—is usually more tractable by mechanised processing. The danger is that as formal analysis by theorem provers becomes more powerful, and therefore more attractive, it leads to a weakening of the crucial demand for human understanding. Sequential processes are a fundamental part of human experience in the world, and we have all learned to grasp that indefinable something that they convey. A process represented by fragments ceases to be humanly comprehensible: the links between fragments formed by state variables are no substitute for a coherent representation of the whole process. Dijkstra explained their inadequacy in his famous letter [Dijkstra68a] about the structure of a program text:

> "The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the progress of the process."

To support both human comprehension and machine tractability, more than one representation may be necessary for the same abstraction. Effective development support software must at least be capable of deriving the comprehensible representation from a more machine-tractable equivalent.

4.6 Abstraction by Context

A major feature of a realistic computer-based system is its multiplicity of operational subcontexts. An aircraft must behave differently in the different phases of a flight: standing, pushback, taxiing, takeoff, climbing, en route, approaching, landing, and so on. A car must behave differently on the highway and in the repair shop. A lift in a large multi-purpose building must behave differently when the equipment is functioning perfectly and when the equipment is faulty, differently in the morning and evening, and differently at the weekend and on weekdays. In these different contexts the system requirements will be different, and so also will the envelope of given properties of the problem domains—for example, the aircraft engines, the car suspension, the lift users' behaviour.

Horizontal abstraction is the necessary tool for separating the different subcontexts. In the absence of this separation, the design of any particular functional behaviour can rely only on the weakest assumptions. For example, the design and provision of normal lift service behaviour must rely on properly functioning equipment. The developer who integrates into this design the detection, diagnosis, and handling of equipment faults is addressing a problem that is too complex for reliable solution. Almost nothing can be assumed, so at every point in the designed behaviour it is necessary to check which of a very large number of possible states currently holds. Eventually the reflective developer will find it desirable to structure this complexity by introducing additional state variables: Has a fault already been detected? What fault? Is the system already trying to recover from a fault? Has normal lift service already been abandoned? Is normal lift service currently in course of being abandoned by bringing the lift to a safe floor? This necessary structuring of system state is exactly the structuring aimed at by a horizontal abstraction by context.

Separation of concerns is a generally recognised principle in the mastery of complexity. Less generally recognised is the need to recombine the separated concerns to produce a satisfactory overall behaviour. Sometimes, in a small non-critical setting, this recombination can be almost completely avoided: the designed behaviour is aborted and the system, or the affected part of it, will be restarted later from a carefully specified initial state. For example, in the classic use case of withdrawing cash from an ATM there are many possible ways of failing: the user's card may be faulty; the PIN may be wrong; the card may have been previously reported stolen; and so on. In the case of such a failure the card may be retained in the machine and the use case is aborted. Dealing with the retained card and with the possible explanations for PIN errors are system behaviours that need not be tightly integrated into the ATM use case, but can be dealt with elsewhere as a separate behaviour for a separate context. However, many systems, including some of the most critical, are required to operate continuously. Aborting the current behaviour and restarting elsewhere from a carefully specified initial state is not a permissible design choice in automotive or avionic systems. Recombining behaviours separated by horizontal abstractions then presents various challenges.

Here we will mention two recombination challenges. First, when horizontally abstracted behaviours can overlap in time one of them may be based on the assumption that the relevant properties of a shared problem domain will be unaffected by the other. For example, the scheduling of train services may be separated from the scheduling and management of track maintenance, relying on the assumption that the separation can be perfect. At any time the rail network can be partitioned into those tracks on which services can be scheduled and those on which maintenance work can be performed. This assumption may be false. The recombination of the two behaviours then must take their mutual interference into account, modifying one or both of them accordingly. The two horizontal abstractions are not compositional.

Second, when horizontally abstracted behaviours are consecutive in time it is necessary to consider whether the problem world post-state of the earlier satisfies the assumed problem world pre-state of the later. For example, normal lift service and firefighter lift service may be consecutive: during normal lift service a fire is detected and the system must be placed under control of the fire brigade. But at the moment of detection the lift car may contain passengers, and may be engaged in a journey to satisfy their requests and other pending floor requests. It will be necessary to design what may be called a *switching* behaviour to deposit any passengers at a safe floor before handing over the lift to fire brigade control.

Horizontal abstraction by context is not, of course, restricted to computer-based systems. A very different example is seen in the parsing of an input text that may contain syntactic errors. The text is a problem domain, and its outer boundary of properties in the overall context is constrained by the input mechanism: for example, the character set may be constrained by a keyboard. In one horizontal abstraction, a syntactically faultless text is assumed, having a well defined structure of which the parser takes advantage. An element in this structure may be *white space*, an abstract lexical token in which any unbroken sequence of space, tab, and carriage return characters is equivalent to any other. For the most helpful diagnosis of errors, however, it may be important to adopt a different abstraction. If the constituent phenomena of white space are not discarded, the physical layout of the text lines can be explicitly recognised: mistyping of a right brace is then more easily pinpointed and diagnosed in a carefully indented text.

5 CAREFULLY THOUGHTFUL USE OF ABSTRACTION

The variety of software development problems is huge, and keeps growing. Some classes of system have evolved effective standard designs and development procedures, but many have not. Much software development is therefore not a routine activity: it comprises a high proportion of *radical*, rather than *normal*, design [Vincenti93]. The developer is to that extent unable to rely on a standard designs evolved and validated by many practitioners over a long period, and must fall back on a personal capacity for invention. This pleasurable innovative activity must be accompanied by a strong inclination to self-questioning. From the first investigation of requirements through to system testing and installation, developers can benefit from questioning what they are doing: from considering explicitly what abstractions they are using, questioning the nature of those abstractions, and articulating how they are related to the purposes of their work and the realities in which they are ultimately grounded.

When difficulty is encountered, it is always good to question the abstraction or set of abstractions within which the difficulty has arisen. Advocates of aspect-oriented software development speak of "the tyranny of the dominant decomposition", and the need to escape, somehow, from the straitjacket it imposes. In the same spirit we may speak of the tyranny of the dominant abstraction. An abstraction that serves well for one purpose can easily become 'sticky': we become unable to escape when for other purposes it becomes a tarpit.

A famous historical example is the Pythagoreans' dominant abstraction of numbers: all numbers are rational. According to tradition, the discovery that the square root of 2 is irrational was more than they could bear; it is even said that they murdered Hippasus, its discoverer. Another example, more obviously germane to software development, is the idea of a *telephone call*. A call is an attempt by one telephone (the caller) to establish one connection to one other telephone (the callee). When it became apparent in the 1990s that telephone systems were becoming more powerful and user features were proliferating, an international effort developed a standard conceptual model for telephony based entirely on the call abstraction. Unfortunately, many telephone features subvert the one-to-one correspondence that is the essence of this abstraction [Zave98]. Conference calls, voicemail, automatic callback, credit card calling and many other features simply cannot be clearly described on the basis of the call abstraction.

Simplification is an important use of abstraction. Harry Beck's map simplified the task of planning a route on the London Underground. It also provides an object lesson in one aspect of the dangers of an analogical model. Although in principle Beck had discarded the geographical phenomena, top to bottom of his map was still, roughly, North to South, and left to right was still West to East. His abstraction was therefore partial, in the sense that the geographical phenomena were only partially discarded: geographical location still influenced the positions of stations on the map, but imperfectly and inconsistently. Inevitably, such a partial abstraction is potentially misleading: some users wrongly suppose that distances are exactly preserved. In one extreme case, a user may undertake a journey visiting four stations and changing between two Underground lines to travel between two stations that are 250 yards apart. Both for the maker and for the user of an abstraction it is vital to understand clearly exactly what has been discarded. In the practice of abstraction, the baby is not always easily distinguished from the bathwater.

ACKNOWLEDGEMENTS

I am grateful to Daniel Jackson for illuminating discussions. I also thank the anonymous reviewers, whose careful and extensive comments and suggestions have encouraged and helped me to improve this essay.

REFERENCES

[Abrial10] Jean-Raymond Abrial; *Modeling in Event-B: System and Software Engineering*; Cambridge University Press, 2010.

[Ackoff62] R L Ackoff; *Scientific Method: Optimizing Applied Research Decisions*; Wiley, 1962.

[Atkinson+08] Colin Atkinson and Thomas Kuehne; *Reducing accidental complexity in domain models*; Software and Systems Modeling, Volume 7 Number 3, pages 345-360, July 2008.

[Dahl+72] Ole-Johan Dahl and C A R Hoare; *Hierarchical Program Structures*; in O-J Dahl, E W Dijkstra and C A R Hoare; *Structured Programming*; Academic Press, 1972.

[Dijkstra68a] E W Dijkstra; *A Case against the GO TO Statement*; EWD215, published as a letter (*Go To Statement Considered Harmful*) to the editor of Communications of the ACM Volume 11 Number 3 pages 147-148, March 1968.

[Dijkstra68b] E W Dijkstra; *The structure of the 'THE' multiprogramming system*; EWD196, published in Communications of the ACM Volume 11 Number 5 pages 341-346, May 1968.

[Dijkstra72] E W Dijkstra; *The Humble Programmer*; Turing Award Lecture; Communications of the ACM Volume 15 Number 10 pages 859-866, October 1972.

[Dijkstra76] Edsger W Dijkstra; *A Discipline of Programming*; Prentice-Hall, 1976.

[Dijkstra89] E W Dijkstra; *On the Cruelty of Really Teaching Computer Science*, with responses from David Parnas, W L Scherlis, M H van Emden, Jacques Cohen, R W Hamming, Richard M Karp and Terry Winograd, and a reply from Dijkstra; Communications of the ACM Volume 32 Number 12, December 1989, pages 1398-1414.

[Evans-Pritchard37] Edward Evan Evans-Pritchard; *Witchcraft, Oracles and Magic among the Azande*; Oxford, The Clarendon Press, 1937. Abridged with an introduction by Eva Gillies; Oxford, The Clarendon Press, 1976.

[Feynman86] R P Feynman; *Personal observations on the reliability of the Shuttle*; Appendix F to the Rogers Commission Report, 1986; available at http://science.ksc.nasa.gov/shuttle/missions/51-l/docs/rogers-commission/Appendix-F.txt, accessed 19th May 2012.

[Goguen+80] J A Goguen and R M Burstall; *Cat, a system for the structured elaboration of correct programs from structured specifications*; Technical Report CSL-118, Computer Science Laboratory, SRI International, 1980.

[Gregory05] Richard L Gregory; The Medawar Lecture 2001 Knowledge for vision: vision for knowledge; Philosophical Transactions of The Royal Society B 2005 360, 1231-1251.

[Hardy40] G H Hardy; *A Mathematician's Apology*; Cambridge University Press, 1940.

[Hartford11] http://matdl.org/failurecases/Building_Collapse_Cases/Hartford_Civic_Center, accessed 3rd October 2011.

[JacksonD06] Daniel Jackson; *Software Abstractions: Logic, Language and Analysis*; MIT Press, 2006.

[JacksonM75] M A Jackson; *Principles of Program Design*; Academic Press, 1975.

[JacksonM76] M A Jackson; *Constructive Methods of Program Design*; in Proceedings of the 1st Conference of the European Cooperation in Informatics, pages 236-262; G Goos & J Hartmanis eds; Springer-Verlag LNCS 44, 1976.

[JacksonM02] Michael Jackson; *Some Basic Tenets of Description*; Software & Systems Journal Volume 1, Number 1, pages 5-9, September 2002.

[Jones90] Cliff Jones; *Systematic Software Development Using VDM*; Prentice-Hall International, 2nd Edition 1990.

[Kramer07] Jeff Kramer; *Is Abstraction the Key to Computing?* Communications of the ACM Volume 50 Number 4, pages 37-42, April 2007.

[Kruchten99] Philippe Kruchten; *The Rational Unified Process: An Introduction*; Addison-Wesley Longman, 1999.

[Lakatos76] Imre Lakatos; *Proofs and Refutations: The Logic of Mathematical Discovery*; John Worrall and Elie Zahar eds; Cambridge University Press, 1976.

[Parnas72] 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.

[Poincaré08] Henri Poincaré; *Science et méthode*; Flammarion 1908, translated by Francis Maitland, Nelson 1914, Dover, 2003.

[TubeMaps11] Clive Billson: *A History of the London Tube Maps*; http://homepage.ntlworld.com/clivebillson/tube/tube.html (accessed 30 May 2011).

[Turing49] A M Turing. *Checking a large routine*; in Report on a Conference on High Speed Automatic Calculating Machines, pages 67-69, Cambridge University Mathematical Laboratory, Cambridge, 1949. Turing's paper is discussed in Cliff B. Jones; *The Early Search for Tractable Ways of Reasoning about Programs*; IEEE Annals of the History of Computing Volume 25 Number 2, pages 26-49, 2003.

[Vincenti93] 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.

[Weyl44] Hermann Weyl; *David Hilbert and His Mathematical Work*; Bulletin of the American Mathematical Society Volume 50 page 612-654, 1944.

[Woodcock96] Jim Woodcock and Jim Davies; *Using Z: Specification, Refinement, and Proof*; Prentice-Hall International, 1996.

[Zave98] Pamela Zave; 'Calls Considered Harmful' and Other Observations: A Tutorial on Telephony; in Tiziana Margaria, Bernhard Steffen, Roland Rückert and Joachim Posegga eds, Selected papers on Services and Visualization: Towards User-Friendly Design, LNCS 1385 pages 2-27, Springer Verlag 1998.

SoSyM Volume 11 Issue 4, pages 495-511, October 2012