Formalism and Intuition in Software Engineering

Michael Jackson

Department of Computing
The Open University
Milton Keynes MK7 6AA
United Kingdom

Abstract. A major and so far unmet challenge in software engineering is to achieve and act upon a clear and sound understanding of the relationship between formalism and intuition in the development process. The challenge is salient in the development of cyber-physical systems, in which the computer interacts with the human and physical world to ensure a behaviour there that satisfies the requirements of the system's stakeholders. The nature of the computer as a formally defined symbol-processing engine invites a formal mathematical approach to software development. Contrary considerations militate against excessive reliance on formalism. The non-formal nature of the human and physical world, the complexity of system function, and the need for human comprehension at every level demand application of non-formal and intuitional knowledge, of insight and technique rather than calculation. The challenge, then, is to determine how these two facets of the development process—formalism and intuition—can work together most productively. This short essay describes some origins and aspects of the challenge and offers a perspective for addressing it.

Keywords: approximation, behaviour, correctness, description, formalism, interpretation, intuition, problem world, requirement, structure.

Introduction

Dieter Rombach's work has been admirably characterised by a resolve to pay attention to the reality of software engineering practice and to the multitude of intuitive and informal insights that have been offered [Endres+03] to clarify its challenges and support its improvement. This short paper follows his excellent example, addressing a specific challenge in software development practice: the proper relationship between formalism and intuition.

Intuition is the faculty of recognition, understanding and action in the world on the basis of experience, insight and knowledge, with little or no appeal to conscious reasoning. The strength of intuition is that it is unbounded: in exercising our intuition we are not restricted to a limited set of observations and considerations decided *a priori*, but we draw whatever presents itself to us from the situation in hand. When we read an intuitive description the words are not opaque: we are looking at the subject matter through the medium of the description. This is how human oral and written communication works: as I hear or read your words I experience or enact through them, in my imagination, what you are saying about the world.

Some extreme examples of human intuition dispense with conscious use of language altogether. Studying how firefighters decide how to tackle a fire leads one researcher [Klein03] to define intuition as "the way we translate our experiences into judgments and decisions ... by using patterns to recognize what's going on in a situation." Another researcher [Rochlin97] describes how operators in military, air traffic control, and other critical environments rely on maintaining an integrated cognitive map drawn from diverse inputs: they call it 'having the bubble'. The map allows them to maintain and act on a single picture of the overall situation and operational status without conscious description, analysis or reasoning.

Formalism, by contrast, relies entirely on conscious description, analysis and reasoning. Its use is not an innate human faculty, but a skill that must be learned. Formalism is an intellectual artifact that evolved from the development of mathematics in ancient civilisations. Its essence is abstraction. Arithmetic and geometry emerged from practical needs: counting shepherds' flocks, measuring farmers' land, paying taxes, and laying out the structures of large buildings. The Greeks saw that mathematics had an intrinsic intellectual interest. Numbers, planes, points and lines could be completely separated from their practical utility. Plato's rule that no-one ignorant of geometry should enter his Academy in Athens was not an expression of welcome to land surveyors or estate agents: it expressed the conviction that knowledge of the material world was inferior to knowledge of mathematics. Only in the abstract world of mathematics could the conclusions of reasoning be proved correct beyond all doubt.

In modern times some mathematicians have expressed the essentially abstract nature of formalism uncompromisingly. In an address [Weyl40] at the University of Pennsylvania, the German mathematician Hermann Weyl said:

> "We now come to the decisive step of mathematical abstraction: we forget about what the symbols stand for. [The mathematician] need not be idle; there are many operations he may carry out with these symbols, without ever having to look at the things they stand for."

Weyl's doctoral advisor was David Hilbert, whom he reported [Weyl44] as saying:

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

For Weyl and Hilbert, the symbols used in a formal description are arbitrarily chosen: any reference to the material world is a mischievous and misleading irrelevance.

Extreme forms of pure intuition or pure formalism are unlikely to appear in any practical enterprise, and certainly not in software development. In practice, formalism is more like applied than like pure mathematics: application to the material world is never very far away, and intuition plays a significant part. In practice, intuition finds expression in semi-formal documents and discourse: some lightweight formal notions may be introduced to avoid obvious potential confusions, and sound reasoning is recognised—though not always achieved—as a desirable goal. How the two should be balanced and combined, both in the large and in the small, is still an open question.

Some Software History

Two streams may be distinguished in the evolving modern practice of software development since it began in the 1940s. One may be called the formal stream. Programs are regarded as mathematical objects: their properties and behaviour can be analysed formally and predictions of the results of execution can be formally proved or disproved. The other stream may be called the intuitive stream. Programs are regarded as structures inviting human comprehension: the results of their execution can be predicted—not always reliably—by an intuitive process of mental enactment combined with some informal reasoning.

Both streams have a long history. A talk by Alan Turing in 1949 [Turing49] used assertions over program variables to construct a formal proof of correctness of a small program to compute the factorial function. Techniques of program structuring, devised and justified by intuition, came to prominence in the 1960s with the control structures of Algol 60 [Naur60], Conway's invention of coroutines [Conway63], and the class concept of Simula67 [Dahl72]. Dijkstra's advocacy of restricted control flow patterns in the famous GO TO letter [Dijkstra68] rested on their virtue of minimising the conceptual gap between the static program text and its dynamic execution: the program would be more comprehensible. In further developments in structured programming the two streams came together. A structured program text was not only easier to understand: the nested structure of localised contexts allowed a structured proof of correctness based on formal reasoning.

At this stage the academic and research communities made an implicit choice with far-reaching consequences. Some of the intellectual leaders of those communities were encouraged by the success and promise of formal mathematical techniques to focus their attention and efforts on that stream. They relaxed, and eventually forsook, their interest in the intuitive aspects of program design and structure. For those researchers themselves the choice was fruitful: study of the more formal aspects of computing stimulated a rich flow of results in that particular branch of logic and mathematics.

For the field of software development as a whole this effective separation of the formal and intuitive streams was a major loss. The formal stream flowed on, diverging further and further from the concerns and practices of realistic software development projects. The intuitive stream, too, flowed on, but in increasing isolation. Systems became richer and more complex, and the computer's role in them became increasingly one of intimate interaction with the human and physical world. Software engineering came to be less concerned with purely symbolic computation and more concerned with the material world and with the economic and operational purposes of the system of which software was now only a part. Development projects responded increasingly to economic and managerial imperatives and trends rather than to intellectual or scientific disciplines.

In recent decades advocates of formal methods have made admirable efforts to reconnect the two streams to their mutual advantage; but the very necessity of these efforts is an indictment of the present state of software development practice and theory as a whole. Formalism and intuition are still too often seen as competing adversaries. Some formalists believe that their work offers powerful solutions that practitioners have wilfully ignored. Some practitioners believe that formalists have simply ignored the real problems and difficulties of software engineering. The purpose of this essay is to offer a little relationship counselling to the parties, and to address the implicit challenge: How can we combine the undoubted benefits of formal techniques with the more intuitive and informal aspects that have always been an integral part of the practice of traditional branches of engineering?

Software Engineering

Structured programming was ideally suited to what we may call *pure programming*. The archetypical expository examples of pure programming are calculating the greatest common divisor of two integers, sorting an array of integers, solving the travelling salesman problem, or computing the convex hull of a set of points in 3-space. These problems proved surprisingly fertile in stimulating insights into program design technique, but they were all limited in a crucial way: they required only computation of symbolic output results from symbolic input data. The developer investigates the problem world, identifies a symbolic computational problem that can usefully be solved by computer, and constructs a program to solve it. The user captures the input data for each desired program execution and presents it as input to the machine. The resulting output is then taken by the same or another user and applied in some way to guide action in the problem world. The process is shown in the upper part of Figure 1.

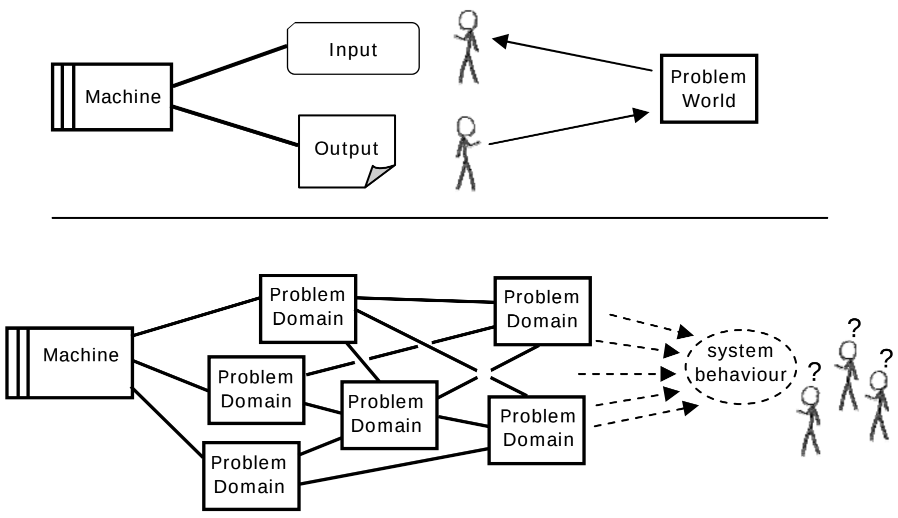

Figure 1: A Pure Program and a Software Engineered Cyber-Physical System

A realistic program of this kind may be designed to solve a general mathematical problem—for example, to solve a set of partial differential equations. It may or may not embody some more specialised theory of the problem world. For example, an early use of electronic computing was to print tables of calculated trajectories for artillery under test at the Aberdeen Proving grounds in Maryland USA [Dickinson67]. The programs may have explicitly embodied a substantial ballistic theory, or they may have been programmed only to solve general systems of partial differential equations. In either case, the machine executing a pure program is isolated from the problem world by the operators who prepare and present the machine's inputs and collect and use its outputs.

By contrast, the lower part of Figure 1 shows a *cyber-physical* system, whose development is a task, not of pure programming but of *software engineering*. In such a system the *machine*—the computing part—is introduced into a material *problem world* to serve specific purposes. The problem world consists of interconnected *problem domains*. some of these domains are physical parts of the world such as mechatronic devices, other computer systems, parts of the built environment, parts of the natural world, and objects such as credit cards that encode lexical information in physical form. Additionally, some other problem domains are human beings participating in the system behaviour, interacting with each other and with the other domains, in both active and passive roles as users, operators, patients, subjects, passengers, drivers, and so on. All of these problem domains have their own given properties and behaviours.

The function of the machine is to ensure a certain desired behaviour in this world, by monitoring and controlling the parts of the world to which it is directly interfaced. The desired behaviour in the world is not limited to these directly interfaced parts, but also embraces other more remote parts which are monitored and controlled through their interactions with other, neighbouring, parts and thus, indirectly, with the machine. The purpose of this desired behaviour is to satisfy the needs of the system's *stakeholders*. Some stakeholders, such as operators, patients and users, are not mere observers but also participate as problem domains in the system behaviour. Others, such as safety regulators and business managers, observe the system behaviour only from a distance. All stakeholders legitimately expect the system behaviour, seen in particular projections from their individual perspectives, to satisfy their needs and purposes.

The Development Task

The behaviour of a cyber-physical system is governed by the interacting behaviours of the machine and the problem domains. Within the limits of the hardware and operating system, the machine's behaviour can be freely defined by the software developed for the system. The behaviour of each problem domain is constrained by its given properties; superimposed on these is the effect of its interactions with other parts of the system. To achieve the desired overall system behaviour the machine must both respect and exploit the given properties and behaviours of all the problem domains.

The overall system behaviour must satisfy the needs of the stakeholders. It is a mistake to suppose that this behaviour is understood in advance by the stakeholders, either individually or collectively, and is waiting only to be discovered and documented. The stakeholders do have various needs and desires, but they may be only dimly perceived. A major part of the development task—explicitly recognised in the past twenty years as *requirements engineering*—is designing behaviour projections that will satisfy the needs of each stakeholder, and combining these projected behaviours into a design for the overall system behaviour. Each desired projected behaviour, and the complete system behaviour that somehow combines them all, must be *feasible*: that is, it must be achievable by the machine, suitably programmed and interacting with the problem domains.

The development task, then, has many facets and parts. The properties of each problem domain must be studied, described and analysed; the many projections of the desired system behaviour must be designed, described and presented to the stakeholders for their critical approval; the combination of these projections must itself be designed; and the behaviour of the machine must be designed and specified at its interface to the problem world. The resulting system is a complex artifact. Before examining the sources and nature of its complexity we will first look briefly at the ubiquitous intellectual activity of software engineering: describing a material reality and reasoning about its properties and behaviour.

Describing and Reasoning

Figure 2 outlines the general process of forming a description and reasoning about it to draw useful conclusions about the machine or the problem world, expressed in a modified or new description.

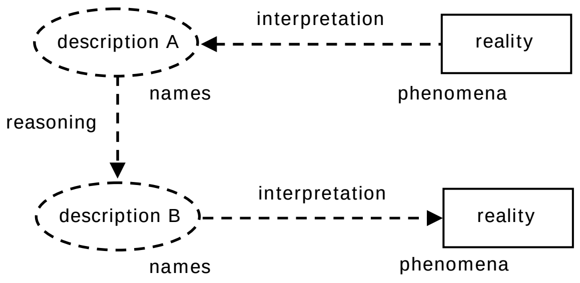

Figure 2: Describing a Material Reality and Reasoning about it

Description A is constructed first. Phenomena of the reality, relevant to the concern in hand, are selected and named, the mapping between names and phenomena being given by the interpretation. The meaning of the description—what it says about the world—depends on the interpretation and on the language in which the description is expressed. Given description A, it is then possible to reason about the world on the basis of that description, deducing a conclusion in the form of description B. This conclusion has a meaning in the reality, which can be understood by reading the derived description in the light of the interpretation.

This simplified account allows us to recognise the difference between formal and non-formal description and reasoning. In a formal setting the chosen language of description is a formal language, rigorously specified. The selected phenomena must then be regarded as elements of types supported by the language. For example: in the language of propositional calculus each relevant phenomenon must be an atomic uninterpreted truth-functional proposition; in the language of predicate calculus it must be a predicate, a function, or an individual. The grammar of the language also includes a small set of connectives, such as logical operators, allowing meaningful statements to be made in the language and combined in various ways. Descriptions are assembled from these elements according to rigid syntactic rules. The advantage purchased by this linguistic rigidity is a formal calculus of reliable reasoning. All or part of the initial description can be treated as a premiss from which conclusions can be derived and proved with mathematical certainty.

The diagram applies equally well to the structure of intuitive or informal description and reasoning. An informal description must be expressed in some language. The language has symbols, and the symbols have some interpretation—that is, they denote some phenomena of the described reality. Yet the content, character and virtues of the intuitive process are quite different from the formal. Symbol choices are very significant in informal description, especially if the descriptions are expressed in natural language: they remind us to look across from the description to the reality it describes and to check continually whether the description remains valid. The logic of informal description is unconstrained: it is nearly true to say that in a rich natural language like English we can say anything whatsoever. We can even define and use new linguistic features within one description. The price for this linguistic freedom is some imprecision in description, and unreliability in both the process and the results of reasoning. Nonetheless, intuition and informality are not merely degraded and incompetent cousins of formalism. Imprecision and unreliability bring major compensating benefits.

In practice the activity of describing and reasoning is rarely perfectly formal or perfectly informal. Rejecting Hilbert's maxim, most formalists usually choose symbols intended to remind the reader of the phenomena they denote in the reality; and many intuitive practitioners use natural language description with careful definitions of the meanings of names, or include embedded formal notations such as finite state machines where greater precision seems necessary.

Formalism and System Complexities

Cyber-physical systems exhibit complexity in more than one dimension. The functional complexity of a realistic system is immediately obvious. Typically a system has many functional features whose purposes are not harmonious or even consistent. The individual features may be intrinsically complex, and the complexity of the whole system is greatly increased by their interactions. Some features may be mutually exclusive in time, but during system operation multiple features may be simultaneously active. Further, many systems are required to operate essentially continuously, scarcely ever reaching a quiescent state in which the system can be removed from service, isolated from the rest of the world, and returned to a well-understood initial state before resuming operation. So the system may be required to achieve smooth transitions between different functional behaviours adjacent in time. For an avionics system, for example, there are transitions from taking-off to climbing, from landing to taxiing, and so on; and a lift control system must maintain user safety and reasonable convenience in the transition from normal lift service to firefighter operation.

One effect of this functional complexity is that there are few or no invariant properties of the required system behaviour. For example, it might be thought that in a system to control the movement of railway trains over a region of track a safety invariant must hold: no two trains must ever be present in the same track segment. But in reality this cannot be a required invariant: it would make it impossible to assemble a train from two trains, or for a breakdown train to deal with the aftermath of a collision or to rescue a locomotive that has lost tractive power. An access control system might seem to demand that no person is ever present in a room for which they have no access authorisation. But this property would restrict escape routes from the building in case of fire, and in that context would be forbidden by fire regulations. In a lift control system an apparent safety invariant stipulates that the lift car doors are never open unless the lift is in home position at a floor. But a firefighter who is in the lift at a high floor must not be prevented from descending even if the doors refuse to close.

The given properties and behaviours of a problem domain—those that it possesses independently of the behaviour of the machine—exhibit a similar dynamic complexity. The given properties and behaviours are determined by four factors, at least two of which are dynamic. A fifth factor determines which properties are of interest at any time.

The first determining factor is *scientific law*—for example, the laws of physics. At the granularity relevant to most software engineering these laws are constant and well understood.

The second factor is what we may call the *constitution* of the domain. This is its shape and material, and the designed, evolved or otherwise determined configuration of its constituent parts. For example, within the bounds set by physics, a person's body weight, physical strength and reaction speed are determined by human physiology in general and the individual's physiology in particular. The maximum acceleration of a lift car rising in its shaft is determined not only by the laws of physics but also by the design of the motor, the power supply and the lift car and counterweight. This second factor, constitution, is more or less constant for each particular problem domain, and is open to study and analysis.

A third, time-varying, factor is the *condition* of the domain. Engineered devices degrade over time, especially if they are not properly maintained or subjected to misuse or to excessive loads. A human operator becomes tired in an extended session of participation in the system; and, in the contrary direction, an operator's speed and skill may increase with practice over a number of similar sessions.

A fourth factor is variation of the *environment* over time. Carefully engineered devices assume an acceptable operating environment, specifying such conditions as wind speed, ambient temperature, air purity and atmospheric pressure. Human behaviour, too, depends on such environmental conditions. If the environment changes the domain may exhibit changed properties.

Broadly, we may say that the first two of these four factors—scientific law and domain constitution—can be investigated and analysed at system design time. The third and fourth—condition and environment—vary during system operation.

The fifth factor, *domain role*, is of a different kind. At any particular time, a problem domain has a large set of potentially observable properties subject to the first four factors, but only a small subset are significant for the system behaviour. The domain itself participates only in some of the system's functions, and in those it plays only a limited role exhibiting only a subset of its given properties. For example, the aerodynamic properties of a car body are highly significant while it is being driven at high speed on a motorway, but irrelevant to its desired behaviour in automatically assisted parking, in the aftermath of a collision, or while undergoing maintenance in the workshop.

These considerations may be summarised by saying that the rarity of required invariants of system behaviour is parallelled by the rarity of invariants of problem domain properties.

Contexts of Domains and Behaviours

There is an important interplay between the variation of domain properties and the variation of the active set of system functional behaviours. For each domain the properties of current significance varies according to its role in each system behaviour of the currently active set. They vary also with changes in the environment, and some of those changes will naturally demand different system behaviours. For example, a power failure in the lift control system seriously affects the properties of the mechatronic equipment, which is now running on emergency power supplies of limited capacity; at the same time it also requires transition to a special parking behaviour in which passengers are brought safely to the nearest available floor.

The most obvious examples of this interplay of domain properties and system behaviour are found in fault-tolerance. In the lift control system, to provide normal lift service the machine must directly control the motor power and direction, and monitor the floor sensors to detect the arrival and departure of the lift car at each floor. This behaviour is possible only if the relevant problem domains of the lift equipment are in healthy condition: this is therefore a *local assumption*, on which the behaviour will rely [Hayes+03]. It then becomes necessary to develop another system behaviour whose specific purpose is to monitor the health of the lift equipment by observing its run-time behaviour. These are therefore at least three distinct system functional behaviours: one to provide normal lift service; a second to detect and perhaps diagnose equipment faults; and at least one other to provide the appropriate behaviour in the presence of a fault. The domain properties of the equipment on which they rely are quite different: one relies on fault-free behaviour; the second relies on the estimated probabilities of different equipment faults and on their consequences in observable phenomena; the third relies on the residual functionality of the faulty equipment.

This restriction of each projection of system behaviour to a particular context in which particular assumptions hold is only a finer-grain version of the inevitable restriction on the whole system's operating conditions. No system, however critical, can aspire to operate dependably in every circumstance that is logically or physically possible. Tall buildings are designed to withstand high wind speeds, but only up to a limit of what is reasonably plausible in each building's particular location. Passenger aircraft are designed to fly in the earth's atmosphere, but not in air of unlimited turbulence or in a high density of volcanic ash. Even when we choose to extend the proposed operational conditions to allow graceful degradation of system function we must still accept some limitations. We can aim only to choose reasonable limits on the circumstances our system will be designed to handle, and to design with adequate reliability within those limits.

The resolutions of functional and domain complexity come together in the assumed *context* of each projected functional behaviour. Each projected functional behaviour can then be represented as shown in the lower part of Figure 1. In each projection the impediments to successful application of appropriate formalism have been greatly diminished. How and why this is so is discussed in the following section.

Structure, Invention and Proof

The great French physicist and mathematician Henri Poincaré wrote [Poincaré08]:

> "For the pure geometer himself, this faculty [intuition] is necessary; it is by logic one demonstrates, by intuition one invents. To know how to criticize is good, to know how to create is better. You know how to recognize if a combination is correct; what a predicament if you have not the art of choosing among all the possible combinations. Logic tells us that on such and such a way we are sure not to meet any obstacle; it does not say which way leads to the end. For that it is necessary to see the end from afar, and the faculty which teaches us to see is intuition. Without it the geometer would be like a writer who should be versed in grammar but had no ideas."

Poincaré is speaking of mathematics, but what he says applies no less to software engineering. It is worth understanding what he says.

The key point is the distinction between demonstration or proof on one side, and invention or discovery on the other side. The primary role of formalism is proof. Before engaging in proof we must know what we wish to prove and the exact context and subject matter for which we wish to prove it. Then we are able to choose an appropriate formal language for our description, knowing that its supported types can represent the relevant phenomena of the reality, and that its logic allows the kind of reasoning on which we are embarking.

In inventing and discovering, on the other hand, we do not know exactly what we wish to invent or discover: if we did we would already have it in our hand. In Poincaré's words, it is necessary to see the end from afar, and the faculty that teaches us to see is intuition. By this we do not mean that we should leap foolishly to a wild guess, impatient of careful thought and reasoning. Rather, invention and discovery are learning processes of a particular kind, in which we need to explore a space of possibilities, sketching our thoughts and perceptions at each resting place that seems promising. For this kind of intellectual activity we need freedom to record our perceptions while they are inchoate, imprecise and even inconsistent. We need a loose structuring of our descriptions and reasoning in which we can reconsider any step without invalidating every other part of what we have done so far. We need to be able to add modal statements about time or obligation to a description that so far contains nothing alien to classical logic. We need to be able to offer temporary accommodation to counterexamples to ensure that they will not be forgotten, without undermining or erasing the imperfectly general but still valuable observation or conclusion that they disprove.

Formalism militates strongly against these purposes. Even if we eschew Hilbert's insistence on extreme mathematical abstraction, the very formality of the chosen language focuses our attention on its abstract logical content and distracts us from attending to the reality described. We are compelled to choose the descriptive language at the outset, when we know least about the terrain to be explored and the flora and fauna we will find there. Worse, a formalism encourages the construction of a single mathematical structure whose virtue is founded on its internal consistency. A single counterexample or a discovered contradiction is a complete disproof: from the contradiction every truth and every falsehood follows without distinction, and the whole edifice becomes discredited.

By contrast, an informal process of discovering properties of the problem world and of the stakeholders' requirements allows the invention of instances of a conceptual structure such as the assemblage of system behaviours sketched in the preceding section. Within such a structure it is possible to separate distinct projections of the system behaviour. Each such projection rests on explicit assumptions of problem domain properties in the context for which the behaviour is designed, and is accompanied by an informal design of the relevant projection of the machine behaviour relying on those assumptions.

Within each of these limited projections formalism can then play its most effective role. The operational context, the problem domain properties, and the desired functionality are restricted: within those restrictions, uniform and relatively simple assumptions can be captured in axioms and a well-chosen formalisation can achieve a good approximation to the problem world reality. The informal design explains how the projected system behaviour is to be achieved, and this explanation can then be made precise and subjected to formal analysis to detect any logical errors. Formalism is deployed locally within each part of the structure. The structure itself, and the substance of its parts, are the product of an intuitive and informal approach.

Envoi

To a committed formalist, advocacy of intuition in software engineering may seem a heretical denial of the value of formalism and rigour. Not so. The point is that formalism has its proper place. Its place is not in the early stages of exploration and learning, where it is premature and restrictive, but in the later stages, where we need to validate our informal discoveries, designs and inferences by submitting them to the rigour of formal proof. Its place is not in the processes of conceiving, designing and forming large structures, but in the later stage of constructing and checking the smaller parts for which those structures provide their carefully defined and restricted contexts, and the relationships among those parts. The essential point is that at every level informal and intelligent use of intuition must precede application of formalism. It must shape the large structure of the whole set of development artifacts; and within that structure it must guide the process of learning, understanding, inventing and documenting the given and desired properties and behaviours of the problem domains. Only then can these descriptions be profitably formalised and their formal consequences verified.

Acknowledgments

This essay owes much to years of cooperation and stimulating discussion with many people, especially Anthony Hall, Ian Hayes, Daniel Jackson, Cliff Jones, Thein Than Tun and Yijun Yu. Since none of them has yet seen even a draft of the essay, they cannot be held responsible for its deficiencies.

References

[Conway63] Melvin E. Conway; *Design of a separable transition-diagram compiler*; Communications of the ACM Volume 6 Number 7, pages 396-408, July 1963.

[Dahl72] 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

[Dickinson 67] Elizabeth R Dickinson; *Production of Firing Tables for Cannon Artillery*; Report No 1371, US Army Materiel Command, Ballistic research Laboratories, Aberdeen Proving ground, Maryland, USA, November 1967.

[Dijkstra68] E W Dijkstra; *A Case Against the Go To Statement*; EWD 215, published as a letter to the Editor (Go To Statement Considered Harmful): Communications of the ACM Volume 11 Number 3, pages 147-148, March 1968.

[Dijkstra89] E W Dijkstra; *On the Cruelty of Really Teaching Computer Science*; Communications of the ACM Volume 32 Number 12, pages 1398-1404, December 1989.

[Endres+03] Albert Endres and Dieter Rombach; *A Handbook of Software and Systems Engineering*, Addison-Wesley, 2003.

[Hayes+03] Ian J. Hayes, Michael A. Jackson, and Cliff B. Jones; *Determining the specification of a control system from that of its environment;* in Keijiro Araki, Stefani Gnesi and Dino Mandrioli eds, Formal Methods: Proceedings of FME2003, pages 154-169, Springer Verlag, Lecture Notes in Computer Science 2805, 2003.

[Jackson00] Michael Jackson; *Problem Frames: Analysing and Structuring Software Development Problems;* Addison-Wesley, 2000.

[Klein03] Gary Klein; *Intuition at Work*; Doubleday, 2003.

[Naur60] J W Backus, F L Bauer, J Green, C Katz, J McCarthy, A J Perlis, H Rutishauser, K Samelson, B Vauquois, J H Wegstein, A van Wijngaarden, M Woodger, ed Peter Naur; *Report on the Algorithmic Language ALGOL 60*; Communications of the ACM Volume 3 Number 5, pages 299-314, May, 1960.

[Poincaré08] Henri Poincaré; *Science et Méthode*; Flammarion 1908; English translation by Francis Maitland, Nelson, 1914 and Dover 1952, 2003.

[Polanyi58] Michael Polanyi; *Personal Knowledge: Towards a Post-Critical Philosophy*; Routledge and Kegan Paul, London, 1958, and University of Chicago Press, 1974.

[Rochlin97] Gene I Rochlin; ; *Trapped in the Net: The unanticipated consequences of computerization*; Princeton University Press, 1997.

[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. 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.

[Weyl40] Hermann Weyl; *The Mathematical Way of Thinking*; address given at the Bicentennial Conference at the University of Pennsylvania, 1940.

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

Juergen Muench and Klaus Schmid eds, Perspectives on the Future of Software Engineering: a Festschrift in Honour of Dieter Rombach, Springer verlag 2013.