Intuition in Software Development Revisited

Meurig Beynon, Russell Boyatt, and Zhan En Chan

Department of Computer Science, University of Warwick
{wmb,rboyatt,echan}@dcs.warwick.ac.uk

Abstract. The role of intuition in software development was discussed in a most original fashion by Peter Naur in 1984. Yet there has been little subsequent interest in elaborating on Naur's ideas. In seeking to explain this neglect, we argue that the accepted views of software development, both within the formal and pragmatic traditions, are deeply influenced by a conceptual framework inherited from computer science and that, within this framework, making sense of the relation between intuition and software development is inherently difficult. In much more recent publications, Naur himself has related his thinking about software development to the philosophical outlook of William James. We discuss the current status and potential implications of Naur's original reflections on the role of intuition with reference to trends in thinking about software development since 1984, and to an alternative conceptual framework for computing, afforded by Empirical Modelling, that can be directly related to a Jamesian philosophical stance.

1 Introduction

Peter Naur's paper *Intuition in Software Development* [26] appeared in 1984. It sets out to "clarify the manner in which software and some of the techniques entering into producing it are grasped by the human being involved", and attempts "to make it clear that immediate human apprehension, or intuition, is the basis on which all activities involved in software development must build". It includes (in section 9 of [26]) an explicit account of how the programmer's 'intuitive knowledge' is engaged in the software development process. Naur's paper predates the foundation of the *Psychology of Programming Interest Group*, but may be seen as closely related to its agenda. The handful of subsequent references to the paper amongst ACM citations, including recent citations by Naur himself [28], indicate that the theme of the paper has not been developed as might have been expected, especially when we consider that Naur is a most distinguished contributor to the study of software practice, and was a recipient of the Turing Award in 2005. In seeking an explanation for this neglect, we argue that the accepted conceptual framework that surrounds software development, in both its narrow formal and broader informal aspects, has its roots in a culture of classical computer science that is ill-suited to making sense of the role of intuition.

This paper reviews some of the ideas in [26] in the light of subsequent developments, and how the practices of commercial software development have been transformed by the adoption of object-oriented approaches that were at that time barely conceived. The scale of software development has also changed to such a degree that the processes involved are more appropriately considered in relation to work practices for a team of software engineers, rather than individuals. The influence of a culture that sees the need to conceptualise software development within a formal framework and is apprehensive about the role of intuition is evident in Naur's reference in [26] to a "discussion of software development methods, viewed as a means of overcoming the hazards of intuitive actions". This influence remains strong, especially in the academic computer science community. Controversy about the extent to which formal approaches can transform practice is still in evidence today, as witnessed by recent papers such as Michael Jackson's *What can we expect from program verification?* [18].

Since 1984, and in his subsequent references to [26], Naur's own perspective on intuition has been related intimately to ideas developed by William James in connection with his researches into psychology [19] and philosophy [20]. What Naur refers to as "immediate human apprehension" has clear points of contact with James's account of experience within the framework of his

Radical Empiricism. One motivation for revisiting Naur's thinking on the topic of intuition in software development is that the Empirical Modelling (EM) project, to which the authors are affiliated, has led to an approach to software construction that is particularly well-aligned to a Jamesian philosophic stance [6]. Consequently, we argue that EM can be regarded as an alternative conceptual framework in which to understand the relation between exercising intuition, in Naur's sense, and the process of software development.

2 The notion of intuition revisited

The word 'intuition' is slippery, amenable to all kinds of interpretation. In developing software, the word is often used quite loosely to refer to the mobilisation of "real" experience in the construction of a program.

At its most extreme, this might be interpreted as the kind of 'guessing the meaning' of a program or a formal specification that is often encountered in teaching, especially amongst novices. Even an experienced programmer may act on the strength of a hunch that does not reflect rigorous analysis, as in: "I think that changing the number of iterations in this *for-loop* will probably adapt this application to cope with a larger dataset". The programmer who acts in this way relies on the fact that it may be possible to vindicate such a hunch by experiment with some degree of confidence. When teaching programming, it is important to highlight the dangers of allowing presumptions about the interpretation to inform changes to a program in such a casual way. One of the main purposes of adopting a formal approach to program development is to shift the focus from "considering your intuitive ideas about what a program is intended to do" to "thinking precisely and abstractly about what is it that you have instructed it to do". As the work of Loomes et al [34] has shown, there is much more involved in achieving such detachment than merely formulating a precise specification: all too often, the human interpreter of a formal specification leaps to conclusions about its meaning based on 'intuitions'. Dehnadi and Bornat [9] relate what it is to be a good programmer to an ability to apply rules to the letter whilst ignoring their external interpretations, and effectively treating them as "utterly meaningless".

There is some tension between the dictionary definitions of 'intuition' as cited by Naur in [26] viz, "immediate apprehension by the mind without reasoning; immediate apprehension by sense; immediate insight", and his concern to overcome "the perceived hazards of intuitive actions". We are not entirely in control of what we immediately apprehend, and if this can be deemed "insight", it is hard to see how it can provoke hazardous actions. It is helpful to consider this issue in the context of programming scenarios.

The extreme forms of guesswork to which a novice programmer with minimal understanding may resort might not qualify as involving intuition in the above senses. It may be that such guessing involves no significant "apprehension of the program by the mind". A trial-and-error action of the experienced programmer (such as that cited above) may, on the other hand, be classified as involving intuition in the sense that it entails purposefully acting on "immediate apprehension by the mind without reasoning". In judging the potential for hazard in such an action, many other factors would be important. If modifying the program were to have an immediate impact on a critical application actively being deployed, such an action would be irresponsible, no matter how plausible it might appear. Provided that it was possible to make the modification in a safe environment, and confirm that it had the anticipated effect in a timely way, this would be a sensible positive step. Not taking the further trouble, by more careful analysis and reasoning, to confirm that the modification was appropriate and likely to have the intended effect in all plausible circumstances might well be hazardous, and is to be deprecated according to the degree of actual harm that might result. From such examples, it is apparent that the potential for hazard lies in the relation between what we can see immediately, and perhaps superficially, and what we might be able to see upon closer scrutiny in other contexts, possibly unforeseen.

What we can apprehend immediately is surely influenced by training. Indeed, an experienced interpreter who examines a specification is trained to consciously look critically at just what has been written and to suppress their desire for it to have a specific meaning. In contrast, the inexperienced developer can be easily seduced into imagining that a piece of code has an intended meaning, and may proceed to elaborate it despite the presence of somewhat obvious errors. Another common characteristic of inexperienced developers is a tendency to make the most optimistic and limited assumptions about the possible conditions that can arise in program execution. Where a novice might be satisfied to terminate an iteration in which the value of an index i is incremented from 1 to 10 by the condition i==10, the experienced programmer may prefer to terminate on the condition i>=10 in the knowledge that there may be exceptional circumstances or potential contexts for code re-use in which the value of i might sometimes be incremented by more than 1. This interest in investigating exceptional scenarios that lie just beyond the fringes of normal execution, and might be expected to arise under anomalous conditions, is characteristic of the experienced developer. The crucial role of specification in managing risk is apparent in this context. A specification provides a clear criterion for success or failure by which a program can be judged. Preventing a program from performing an action in inappropriate circumstances ("i>=10"), even though these circumstances apparently cannot arise, typifies the fail-safe character of a formal approach. In the absence of a specification, the consequences of specifying loop termination via i==10 as opposed to i>=10 might fortuitously be more favourable in certain application contexts.

As might be expected from Naur's later writings, there is a strong affinity between his view of intuition and the outlook of William James. For James, what is immediately given in experience must be acknowledged to include not only the discrete objects and attributes that we are accustomed to view as the direct products of our senses, but also the relationships in which these elements of experience are conjoined. The different variants in the dictionary definitions of 'intuition' cited by Naur correspond to different types of association that can be apprehended immediately: the sun might be apprehended as being the same size as a coin in some context, but reasoning would suggest that this is an illusion ("immediate apprehension by the mind without reasoning"); a pen and pencil may be seen to lie beside each other on the table, and so be apprehended as belonging to the same sensory context ("immediate apprehension by sense"); the letters on the doormat indicate that the postman has visited ("immediate insight").

The types of relationships between a program and its intended behaviour and meaning that are topical in the above discussion of intuition in programming are also examples of *conjunctive relations* in James's sense. The way in which an experienced programmer is led "by intuition" to hypothesise that the value of a particular variable determines the size of the dataset to be processed by a program might reflect many different aspects of the experience of glancing at the program: the name of the variable, the location of its declaration, the structure of the program – whether individually, or taken together, each can make its contribution. Likewise, when the programmer is tempted to perform the "intuitive action" of changing the value assigned to this variable, their perception of how the program relates to its domain of application and the potential implications for its behaviour in context will exercise its own influence. In a similar spirit, the suggestion that programming-by-guesswork should *not* be regarded as involving intuition stems from the knowledge that, below a certain level of competence, the novice programmer is unlikely to perceive any sensible connections between a program and its behaviour, and so resorts to guessing.

The centrality of the notion of associations given in immediate experience to James's philosophical stance is echoed in Naur's assertion in [26] that "immediate human apprehension, or intuition, is the basis on which all activities involved in software development must build". For James, the associations that are given in immediate experience are the basis of all knowledge, in terms of which all claims to truth must ultimately be accounted. In this perspective, there is no place for the idea of transcendental knowledge separated from experience through abstraction

and enjoying some privileged status of objective absolute truth. This is not to preclude the possibility that contexts for experience can be engineered in such a way that law-like rules apply, but to remain agnostic as to whether common experience can in general be described within a universal context of such a kind. Such a pragmatic framework for judgement is acknowledged in Naur's observations [26] to the effect that: "over a large range of a person's reactions to certain common situations of life it is impossible to classify them as successes or failures. There may be no basis for criticising the reaction that comes to us intuitively."

3 Software Development revisited

A vast amount has been written about software development since Naur's paper. It is impossible to do justice to such an extensive field of research in a few paragraphs, but – with some poetic licence – we would like to sketch a general picture of how we interpret the progression of ideas and practices.

In broad terms, we argue that there are two quite different and complementary kinds of activity that software development has needed to address over this period. One of these relates to stable contexts of experience that can be engineered to exhibit law-like characteristics. In such contexts, the nature of the state-changing agents that can act is well-understood, as is the effect of their actions and interactions. The second relates to shadowy contexts of experience whose stability is precarious, where objects are not clearly defined and where few laws are apparent. In such contexts, the nature of any latent agency that can be identified and reliably exercised is unclear. (For convenience, we shall refer to these as Type 1 and Type 2 contexts respectively.)

Historically, thinking about software development was predominantly focused on Type 1 contexts. An archetypal example might be the batch mode interaction with an idealised reliable computer that informs the basic mathematical conceptions of what a program is. One tradition of software development emphasises the kinds of techniques and conceptual framework that suits Type 1 contexts and may be seen, in some of its incarnations, as striving to demonstrate its sufficiency. Wherever software development involves interaction with Type 2 contexts, conceptually different kinds of activities need to be invoked. In the first instance, such activities would typically be classified as 'requirements engineering', that – if they exploited the computer at all – did so in ways that were distinctly different from the actual implementation of software. In practice, once software attained a certain degree of complexity, the development of software for Type 1 contexts inevitably came to require interaction with Type 2 contexts. More problematically, it came to involve a style of development in which there was no real possibility of limiting the interaction with Type 2 contexts to the initial phase of the development. The ways in which rule-based activity interacts with more open-ended problem-solving activities in the human solution of a Sudoku puzzle is a fertile source of helpful analogies here.

Over recent years, a number of factors have reinforced the need to blend the systematic techniques for software development that apply in Type 1 contexts with the essential process of understanding that is required to identify Type 1 contexts within Type 2 contexts. Software is developed for reactive systems, not stand-alone computers. Input-output is mediated through many different kinds of devices. The scale of software development is such that no single designer can appreciate all the perspectives that are relevant to the effective solution of problems. The kind of experimental use of the computer in order to 'see how something works out' that is deprecated in software development in a Type 1 context has become a pervasive ingredient of good software practice. When development of this nature ceases to be simply a means to generate disposable prototypes and becomes a vital component of the development process itself, as in an eXtreme Programming approach [2], the entire integrity of the software development activity is called into question. It is in this fashion that two quite dissimilar stances on software development have evolved, one *formal* that aspires to give an adequate account that is faithful

to the legacy of classical computer science, the other *pragmatic* that focuses on the complexity of the sociological processes that surround software practice and experience and invokes a more human centred approach. A crucial issue that distinguishes these two stances is the status – indeed the relevance – of the concept of a 'specification'. For the computer scientist, the notion of a precise specification is what makes a program meaningful — without it there can be no concept of correctness, optimisation or refinement. The status of what is built by using the computer to explore possible specifications in an experimental fashion is then wholly unclear — it can scarcely be legitimately called 'a program'.

This brief overview of the complex field of software development in theory and practice is of course simplistic in many respects. It is nevertheless helpful in orienting thinking about software development towards the perspective on intuition that has been introduced in connection with Naur and James above. What is a particularly useful, and relevant to making sensible connections between the formal and pragmatic stances on software development, is the unification that a Jamesian outlook enables. Conjunctions given in experience play the same fundamental role in sense making whether or not we are operating in contexts where law-like characteristics pertain. Naur draws our attention to this key point in his accounts of the role of intuition in interpreting text (section 6) and reasoning and proof (section 7) in [26].

To refine the high-level sketch of software development given above, we shall briefly review relevant contributions that postdate Naur's paper and help to illustrate the current trends in thinking. The overall objective will be to relate the longstanding problems that beset software development to the difficulty of making effective use of intuition within presently accepted conceptual frameworks. This is the key idea that motivates the introduction of Empirical Modelling, as will be briefly discussed in our conclusion.

One of the most important commentators on the software development process is Fred Brooks. In his paper *No Silver Bullet* [8], Brooks not only identifies the limited impact of virtually all the innovative practices for software development that had at that time been proposed; he also highlights characteristics to which effective software development should aspire. In his vision for development, he uses the metaphor of 'growing software' and speaks of the paramount significance of 'conceptual integrity' in a successful software product. In the pursuit of conceptual integrity, Brooks advocates the idea of having one person in overall control of the concept that informs a piece of software, so that whilst it may be built by many people, it has the same coherence as the product of a single mind.

A suggestive parallel may be drawn between Brooks's ideal software process and the way in which understanding is deemed to develop in James's outlook. For James, the most intimate relation of conjunction that is given in experience is the one that links experiences within the same stream of consciousness. This is precisely the kind of relationship between one stage of development of a piece of software and the next that is suggested by Brooks's metaphor of software growth. In a similar spirit, Brooks's notion of conceptual integrity echoes the way in which James identifies 'knowing' with holding the relationship between two experiences in one mind.

What is significant here is the confidence with which Brooks, a highly experienced and distinguished software engineer, invokes a vision for software development that derives no explicit support from the apparatus associated with the formal tradition of software development. Rather, there is on his part an apprehension – one might even say an intuition! – that it makes sense to speak of the growth of software as an organic process of interaction between the embryonic software product and its environment that takes place in the developer's mind. This gives some credibility to the idea that it might indeed be possible to guide software development more directly by means of what Naur refers to as 'intuition'. It also highlights some significant issues that have to be addressed in realising such a vision, namely:

- how the immediacy of intuition is to be ensured;
- how the manner in which intuition evolves over time is to be reflected;

– how intuition can play a role in achieving intersubjectivity.

In the rest of this section, we shall consider to what extent some extant approaches and proposals for software development correspond to Brooks's idealised – but for the present fictional – notion of *growing software* as an intuitive activity. In doing this, it is helpful to distinguish two quite different perspectives on software development: one that sees such development primarily in terms of effective methods for engineering software; the other representing the broader view of development as a form of theory-building within the application domain.

3.1 Software development methods

As Naur points out in [26] section 10, software development methods address concerns that relate to three main areas: what activities are peformed; what notations and languages are to be used; what ordering of activities is prescribed.

An approach such as the Abrial's B method [1] illustrates archetypal formal software development that begins with a logical specification and proceeds through a top-down process of refinement to the generation of code. Such an approach may conceivably be suited to development in what has been identified above as a Type 1 context, but the ambitions for formal development of this nature have been tempered over recent years to such an extent that methods like B are typically applied only to relatively small software components that have a safety-critical role. The current status of such methods bears out Naur's observations in [26] to the effect that (even in the case of a formal specification that may express unambiguous abstract logical relations) "the compatibility of descriptions used in developing a piece of software and the matters of the world that are supposed to be modelled by them remains a matter of human intuition" and "the most suitable form or language for any situation will depend not only on the degree of formalization of the description, but equally on the character of the aspect of the world to be modelled and the background experience of the programmer who has to establish the description".

A key issue here is that, where the "interplay of intuitive knowledge had by the programming person and the real world accessible to that person, which includes the texts used or produced by the person" described by Naur in [26] section 9 is concerned, the correspondence between features of the software text and meaningful observables in the application can take many forms. In a B specification for the safe operation of a railway crossing, for example [32], the text consists of predicates that constrain the values of certain variables that correspond in some way to observable features of the actual situation, such as whether a train is approaching, whether the barrier is up, and whether the lights are flashing etc. The precise correspondence between such 'real-world' observables and their abstract counterparts is in some respects not easily amenable to direct experience. For instance, it is impossible to observe an actual railway crossing without seeing a barrier that is in some specific position, whether up, down or somewhere in between. Yet the B specification addresses abstract relationships between variables that are to be respected in implementation, without registering any specific values these variables have *as of now*. To appreciate how significant this is in relation to Naur's perspective on the role of intuition in development, contrast the way in which 'real-world' observables are represented when software development is carried out using spreadsheets, as studied by Nardi in [24]. In a spreadsheet, the representation of the current values of observables is explicit – it is their potential relationships, as constrained by meaningful transitions to which they may be subject, that are implicit.

The approaches to software development that have been proposed by Harel offer further insight. In *Biting the Silver Bullet* [14], Harel recognises the way in which the character of an application impacts on the efficacy of a development approach. He distinguishes 'one-person programming', to which traditional formal software development may be seen as applying, from programming reactive systems, for which different techniques are required. In emphasising the vital importance of visual formalisms, such as the *statecharts* that he himself devised,

Harel is implicitly endorsing Naur's claims about the significance of intuition in development, whilst at the same time remaining faithful to the spirit of a formal approach. More recent work of Harel and Marelly on play-in scenarios [15] sheds further light upon the tensions between formal and pragmatic approaches to development. In [15], Harel explains the motivation for the play-in approach with reference to the expressive limitations of message sequence charts, a standard component – as are statecharts – of the Unified Modelling Language (UML). In order to overcome these limitations, message sequence charts are used to create software prototypes that can be explored interactively by potential end-users and other stakeholders, and responsively refined.

Deeper philosophical issues lie behind the apparent unification of formal and intuitive aspects that a hybrid approach such as Harel's affords. These can be related to James's agnostism regarding the status of law-like contexts of universal scope. Their topicality is reflected in the "Grand Challenge" of verified software posed by Jones, O'Hearn and Woodcock [21], which has, as one of its central objectives, the development of a verifying compiler.

Of particular interest in relation to the role of intuition in software development is the question posed by Michael Jackson [18] in response to the aspirations for a verifying compiler: "What can we expect from program verification?" In addressing this question, Jackson draws attention to the need to take the engineering component in software engineering seriously, and to the difficulty of achieving the separation of concerns between the abstract computational and physical worlds that a formal specification is intended to provide. In this connection, he reiterates the concerns expressed by Ferguson [10, p.168] to the effect that modern engineers devote too much attention to calculation and the formal analysis of structure and too little to the physical reality of the world of which those structures are a fundamental part. In this respect, they are in danger of underestimating the importance of "an intuitive feel for the incalculable complexity of engineering practice in the real world". Like Harel, Jackson recognises a crucial distinction between different kinds of problem context; in his case, 'routine design' and 'radical design'.

Jackson's critique of formal specification also has points of contact with Harel's, in that it recognises the limitations of imposing structure upon software solutions in an exclusively top-down fashion. For Harel, the message sequence chart has to be revised dynamically by continually exposing prototypical interaction to the users. For Jackson, the system decomposition must in general be provisional, and potentially subject to revision when due consideration is given to the interconnections between components identified through the analysis of problem frames. This shift of priority from a top-down to a bottom-up perspective is consonant with Naur's emphasis upon the role of intuition, and with James's observation [20] "that rationalist thinking proceeds by going from wholes to parts, whilst empiricist thinking proceeds by going from parts to whole".

The idea that the challenges of software development can be met by a single method is by now largely discredited (see for instance, Jackson's remarks in [17]). More perplexing from the point of view of identifying a satisfactory conceptual framework for modern software development is the way in which the characteristics of formal approaches that at one stage seemed most invulnerable are being subverted in widely adopted contemporary practices. So-called lightweight formal methods, such as those based on Daniel Jackson's Alloy [16] for instance, give useful practical support for the kinds of exploratory investigation of restricted specifications that have taken the place of attempts at monolithic comprehensive system specification. Such methods enable a software developer to gain valuable evidence for the plausibility of intuitions about relationships that might pertain in a real-world scenario. However, they have attracted criticism from logicians, who rightly regard plausible evidence of this nature as something to be clearly distinguished from proof. In a similar spirit, the agile development practices that have displaced more systematic formal approaches can represent a challenge to the idea that firm and explicit specifications are central to effective software development.

3.2 Theory building

Traditional approaches take a production view of software development. An alternative way to consider software development is to regard it as 'theory building'. This view is commended by Naur himself in [27] and can also be found in Turski and Maibaum [33]. Turski and Maibaum lean towards a formal concept of a theory that can be expressed in a suitable logic, and regard the kind of pragmatic considerations that have guided the development of systems such as real-time systems with some concern. The potentially problematic aspects of too formal a notion of a theory in this context are clearly identified by Jackson in [18]. With reference to the specification of "software-intensive" systems, he remarks:

> The formalisations of environment properties and system requirements are necessarily imperfect. First, because formal terms will be unavoidably fuzzy in their definition and interpretation. Second, because values of continuous phenomena must be approximated. Third, because there are no frame conditions: the natural world allows no bound to the phenomena or properties that may prove relevant to the truth or falsity of an assertion. Fourth, because physical properties are not in general compositional: effects that can be properly ignored for each property individually may play a critical role in their composition.

In keeping with his respect for the software developer's intuition, Naur [27] favours a 'softer' notion of theory, similar to that introduced by Gilbert Ryle in [31]: "Very briefly, a person who has ... a theory in [Ryle's] sense knows how to do certain things and in addition can support the actual doing with explanations, justifications, and answers to queries, about the activity of concern". It would be quite inappropriate here to repeat Naur's extensive discussion of "programming as theory building"; suffice to say that it underlines the way in which putting the emphasis on intuition in software development necessarily broadens the scope of concern. For instance, a programmer who has a theory about a program according to Ryle's criterion has knowledge of the domain and of unwritten programs that are neighbouring to the program "in the space of sense". They are also in a position to answer *what if?* questions about the implications of modifying the program text or the environment for its execution.

An interesting aspect of Ryle's notion of theory is its curiously personal and situated quality. What the programmer can do is in the first instance a private matter; it may or may not be something another programmer would know how to do or interpret. For the programmer to display their knowledge of the theory, they need to have to hand the program with which to do things and about which to give an explanation. There is an implicit acknowledgement that "the actual doing with explanations, justifications, and answers to queries, about the activity of concern" serves a communicative purpose that is appreciated by the observer and enquirer. Philosophically, this represents a shift towards a much more pragmatic conception of theory such as was framed by Richard Rorty in [30] (as cited by Kaptelinin and Nardi [22, p.24]):

> [it is not] a matter of getting reality right, but rather ... a matter of acquiring habits of action for coping with reality.

To make sense of the role that theory of this nature can play in software engineering it is essential to consider the social context within which the interactions between the various stakeholders take place.

In [29], Kari Rönkkö reviews the state-of-the-art where incorporating social issues in software engineering is concerned. In his paper, Rönkkö draws on the work of Brooks [8], Naur [27] and Christiane Floyd [11]. Floyd's proposal for a new framework to accommodate the social element of software development emphasises the significance of 'reality construction' in relation to software requirements, design and implementation. Floyd's "basic underlying assumptions" [11, p.95], as cited by Rönkkö in [29], but here somewhat abbreviated, are:

– We do not analyse requirements; we construct them from our own perspective ... In most cases they reflect differences in perspective and are subject to temporal changes.
– We do not apply predefined methods, but construct them to suit the situation at hand ... What we are ultimately doing in the course of design is developing our own methods,
– We do not refer to fixed means of implementation that take effect later on when working out the details of implementation decisions. Instead, we construct the meaningful use of implementation by testing, selecting, or complementing what is already available

Though Floyd's assumptions most self-evidently relate to radical software design, Rönkkö [29] illustrates their topicality with reference to what might appear to be the 'routine' examples of software development practice highlighted by Naur in [27]. The crucial issue highlighted by Rönkkö is the *indexicality* of language: "the fact that words only take on a complete meaning in the context of their production". From Naur's examples, it is apparent that, even in relation to what was considered to be the modest task of extending the first version of a compiler, and despite the fact that appropriate documentation was available, the communication between developers was not enough to overcome the problems posed by indexicality.

Rönkkö's proposed approach to tackling these communication problems is to apply the documentary method of interpretation [12] in the software engineering context. His research leads him to question "[whether] such a thing as a coherent view of software can exist" and to conclude that "massive interaction has a crucial role in software engineering".

In [29], Rönkkö suggests that "adequate indexicality of terms is ... 'the' crucial factor for practitioners in order to reach success in gaining an understanding of and cooperation around continuously growing software applications in development projects". A potential limitation of his proposal is that it puts the emphasis on resolving this problem through a process that combines additional documentation and massive interaction. This reinforces the bias within existing cultures of software development towards communication that is strongly centred on natural language.

A complementary view that gives greater weight to the role of artefacts and theory in communication is represented in Kaptelinin and Nardi [22]. The research described in [22] builds on previous research on computer-mediated activity that has its roots in the study of human-computer interaction [25]. Whilst this tradition of research is not explicitly concerned with software engineering in its narrow sense, and is accordingly not cited by Rönkkö in [29], it deals with issues that are increasingly topical in modern — for instance, agile — software development practice. It is also highly relevant to the agenda set out by Naur in [26]. In particular, Naur's account of software development in [26, p.73] anticipates in certain key aspects Nardi's vision for applying activity theory to study how artefacts serve as new mediating tools and representations in individual and cooperative work, as described in [25].

In this connection, the scale of the documentation of interpretation and interaction envisaged by Rönkkö potentially invites the sort of criticism made of ethnomethodology by Kaptelinin and Nardi in [22]. Whilst recognising the qualities of ethnomethodology in paying more attention to "subjects' rich understandings of their own experience", they also caution that "we cannot merely relate accounts of endless detail with no summarizing, shaping, transforming tools at hand". Their concern about the "avoidance of generalization and abstraction" in ethnomethodology leads them to observe that "without some theoretical connective tissue we cannot speak to each other".

Brooks's vision for 'growing software' highlights a different and bolder possibility: that to a much greater degree the understanding that is required to support the developers' intuition and communication can be embodied in the artefact itself. This idea, that – in some contexts – understanding necessarily has to be implicit in interactive artefacts and the interpretations that the human interpreter can project upon interaction with them, is vividly expressed by Gooding [13] in his notion of a *construal*. Gooding challenges received views about the extent

to which linguistic and theory-led accounts of systematic activities and methods (such as the 'scientific method') can do justice to human agency and observation in scientific experiment.

In conceiving of artefacts that can embody knowledge that develops as they are being built, it is impossible to evade the problematic issues associated with the *constructivist* ideal. In [23], Bruno Latour highlights the ways in which research on constructivism has seemed to discredit the concept, potentially beyond repair. In his attempt to rehabilitate constructivism, Latour is particularly disparaging about the impact of methods for construction based on a 'theory of action' in [23]. Despite her strong endorsement of activity theory as "a single coherent framework [that weaves together] so many interesting theoretical constructs crucial to an understanding of human activity" [25], Nardi likewise expresses some pertinent reservations. She observes that, whilst activity theory rejects a Cartesian mind-body duality, it favours "the depiction of an ordered reality regulated through objects, mediators and goal-directed behaviour". Nardi also identifies several topical respects in which activity theory does not do justice to the experience of designers of modern technology. These include the need to account for: the provenance of objects; conflicting objects within and between subjects; the capacity to be thinking about several things at once; openness to diversion and scope for making rapid shifts in attention.

4 Conclusion

The purpose of this paper has been to set out the context for the research we have done on applying Empirical Modelling (EM) to software development (cf. e.g. [7]). A detailed discussion of this work is beyond the scope of this paper, but we conclude by summarising some key features of EM that resonate with the themes that have been mentioned in tracing significant developments that relate to Naur's discussion of intuition in software development [26]. The term 'intuition' will here be used to refer to immediate human apprehension of conjunctive relations in a Jamesian sense, as discussed in section 2.

EM purports to provide a conceptual framework that is well-aligned to Brooks's vision in so far as it relates to building 'construals' that serve the needs of theory development. The choice of the term 'construal' rather than 'program' is deliberate, and reflects the idea that a radical reconceptualisation is required to account for computer artefacts that serve in the role that programs ostensibly do in Naur's vision for software development as theory building [27]. In EM, this reconceptualisation is associated with regarding a computer artefact as embodying patterns of observables, dependencies and latent agency.

There are some essential characteristics that the computer artefact needs in order to serve as a construal. They relate to the following issues (cf. section 3), each of which is problematic when we conceptualise a computer artefact as a traditional computer program:

- *Immediacy:* Supporting the intuition involves enabling the programmer to perceive connections between a computer artefact and the situation to which it refers. Rendering changes to a computer artefact immediately perceptible in such a way that their implications for the referent can be directly appreciated is difficult. The benefits of immediate feedback are apparent in spreadsheets. They have also been a major motivation for changes to the development process itself, and the advent of test-driven, data-driven and agile approaches. The classical concept of 'program' offers scant support for designing computer artefacts in this fashion. Putting the emphasis on what the programmer perceives is in tension with the concept of "abstracting away from the domain" that dominates classical thinking about programming.

- *Dynamism:* During the development process, the connections that a programmer can make between a computer artefact and its referent typically become progressively richer and more remote from computational mechanisms. Capturing the dynamic nature of the intuitions

surrounding a piece of software is problematic in the traditional view of software development, where the emphasis is on refining the functionality of the product and elaborating the implementation. The manner in which associations with a piece of software develop over time motivates a very different treatment. Making connections is an activity that evolves as new layers and modes of observation are introduced, and as the skills necessary to sustain such observation are acquired. A feature of this activity is that what is at first consciously interpreted subsequently passes into the subconscious, and no longer requires explicit interpretation. In EM, substituting a dependency in place of the explicit computation of a functional relationship introduces automatic updating actions that no longer need to be interpreted by the programmer. The essential mechanism exploited by EM is evident in the dependency maintenance activity that underlies a spreadsheet.

– *Intersubjectivity:* Linking intuition to perception of conjunctive relationships given in experience necessarily lends a personal subjective character to the programmer's theory building. James [20, p49] identifies the perception of a conjunctive relationship with a transition from one experience to another that is *continuous*. He adds: "Continuity here is a definite sort of experience; just as definite as is the *discontinuity-experience* which I find it impossible to avoid when I seek to make the transition from an experience my own to one of yours". This subjectivity is reflected in the meaning of an EM construal, which cannot be defined in isolation from the programmer's live interaction with — and live interpretation of — the artefact. By contrast, the formal conception of a program emphatically belongs to the objective common world. Where EM construals are concerned, following the precepts of James [20, p.141], subjectivity and objectivity are matters of classification of experience. And since the same construal can sustain many different patterns of interaction and interpretation, it can help to overcome the problems of indexicality highlighted by Rönkkö in [29] and contribute to the unification of diverse perspectives that is needed to realise conceptual integrity [8]. The way in which such negotiation of meaning and viewpoints can be supported through exploiting networks of dependency relations is partially illustrated in Nardi's account of collaborative software development with spreadsheets [24].

When taken together, the above observations indicate that EM artefacts are well-suited to expressing the immediate and dynamic quality of intuition, and that their development can trace the path from subjective to objective understandings of experience. On this basis, they have been found to have particular promise as a technology to support learning [3] — as is appropriate for any technology that aspires to support the sense-making activities that play an essential role in radical software design. Whilst dependency maintenance is a crucial ingredient in EM, yet more important are the principles — rooted in observation and experiment — by which artefacts are constructed.

In the patterns of interaction that typically emerge in association with the successful deployment of EM, what is initially seen as no more than a source of experience — a mere artefact — begins to serve an explanatory function as it migrates into a construal. As these patterns of interaction become more familiar they may come to be associated with more clearly defined referents and conventions for interaction and interpretation, and so serve in the role of a model. It is at this stage that, by selectively enacting and automating interactions and interpreting them in preconceived ways, an EM artefact can be viewed as a program. In the progression from artefact to construal to model to program there is in principle nothing to arbitrate between one interpretation of the artefact and another other than the discretion exercised over interaction with the artefact by its maker(s). To make the transition to a conventional program requires a further, more radical, step, whereby the link with living observation is severed by abstraction, and the automated activities are optimised to their specific functions in the manner of classical computer science.

This picture of development within an EM framework highlights the predominant emphasis placed in EM on the most primitive aspects of design activity (cf. [5]). In this respect, EM is well-suited to the agenda for designers of modern technology as identified by Nardi in [25] (cf. section 3.2). In our view, as discussed in [4], it is also well-aligned to fulfilling the promises of constructivism to which Latour alludes in [23].

References

1. Jean Raymond Abrial. *The B Book*. Cambridge University Press, 1996.
2. Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.
3. Meurig Beynon. Computing technology for learning — in need of a radical new conception. *Journal of Educational Technology and Society*, 10(1):94–106, 2007.
4. Meurig Beynon and Antony Harfield. Lifelong Learning, Empirical Modelling and the Promises of Constructivism. *Journal of Computers*, 2(3):43–55, 2007.
5. Meurig Beynon and Steve Russ. Experimenting with Computing. *Journal of Applied Logic (to appear)*, 2008.
6. W. M. Beynon. Radical Empiricism, Empirical Modelling and the nature of knowing. In Itiel E Dror, editor, *Cognitive Technologies and the Pragmatics of Cognition: Special Issue of Pragmatics and Cognition*, volume 13, pages 615–646. December 2005.
7. R. C. Boyatt, W. M. Beynon, and S. B. Russ. Rethinking Programming. In *ITNG '06: Proceedings of the Third International Conference on Information Technology: New Generations*, pages 149–154, 2006.
8. Frederick P. Brooks. No Silver Bullet - essence and accident in software Engineering. *Computer*, 20(4):10–19, April 1987.
9. Saeed Dehnadi and Richard Bornat. The camel has two humps. In *Little PPIG*. Coventry, UK, 2006.
10. Eugene S. Ferguson. *Engineering and the Mind's Eye*. The MIT Press, 1992.
11. C. Floyd. Software development as reality construction. In C. Floyd, H. Züllighoven, R. Budde, and R. Keil-Slawik, editors, *Software Development and Reality Construction*. Springer Verlag, Berlin, 1992.
12. H. Garfinkel. *Studies in Ethnomethodology*. Polity Press, 1996.
13. David Gooding. *Experiment and the Making of Meaning: Human Agency in Scientific Observation and Experiment*. Kluwer Academic Publishers, 1990.
14. David Harel. Biting the Silver Bullet - Toward a Brighter Future for System Development. *IEEE Computer*, 25(1):8–20, 1992.
15. David Harel and Rami Marelly. *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.
16. D. Jackson. *Software Abstractions: Logic, Language and Analysis*. MIT Press, 2006.
17. Michael Jackson. *Problem Frames and Methods: Analysing and Structuring Software Development Problems*. Addison Wesley, 2000.
18. Michael Jackson. What can we expect from program verification? *IEEE Computer*, 39(10):53–59, October 2006.
19. William James. *The principles of psychology*. Macmillan and Co. Ltd, 1890.
20. William James. *Essays in Radical Empiricism*. Longmans Green, 1912.
21. Cliff Jones, Peter O'Hearn, and Jim Woodcock. Verified Software: A Grand Challenge. *Computer*, 39(4):93–95, 2006.
22. Victor Kaptelinin and Bonnie A. Nardi. *Acting with Technology: Activity Theory and Interaction Design (Acting with Technology)*. The MIT Press, 2006.
23. Bruno Latour. Promises of Constructivism. In Don Idhe and Evan Selinger, editors, *Chasing Technoscience: Matrix of Materiality*. Indiana University Press, 2003.
24. Bonnie Nardi. *A Small Matter of Programming*. MIT Press, 1993.
25. Bonnie A. Nardi, editor. *Context and Consciousness: Activity Theory and Human Computer Interaction*. The MIT Press, 1995.
26. Peter Naur. Intuition in Software Development. In *TAPSOFT*, volume 2, pages 60–79, 1985.
27. Peter Naur. Programming as theory building. *Microprocessing and Microprogramming*, 15:253–261, 1985.
28. Peter Naur. Computing versus human thinking. *Commun. ACM*, 50(1):85–94, 2007.
29. Kari Rönkkö. Interpretation, interaction and reality construction in software engineering: An explanatory model. *Information and Software Technology*, 49:682–693, 2007.
30. Richard Rorty, editor. *Objectivity, relativism and truth: Philosophical papers*. Cambridge University Press, 1991.
31. Gilbert Ryle. *The Concept of Mind*. Penguin, 1949.
32. Ib Sorensen and David Neilson. The specification of an automatic train control system. Technical report, B-Core (UK) Ltd., 2002.
33. W.M. Turski and T. S. E. Maibaum. *The Specification of Computer Programs*. Addison-Wesley Publishing Company, 1987.
34. Rick Vinter, Martin Loomes, and Diana Kornbort. Applying Software Metrics to Formal Specifications: A Cognitive Approach. *IEEE Metrics*, pages 216–223, 1998.