Ghosts of Programming Past, Present and Yet to Come

Russell Boyatt Meurig Beynon
Computer Science
University of Warwick
{rboyatt, wmb}@dcs.warwick.ac.uk

Megan Beynon
C3RI
Sheffield Hallam University
megbey@gmail.com

Keywords: programming; computational thinking; making construals; Empirical Modelling; music

Abstract

Twenty-five years ago, when the Psychology of Programming Interest Group (PPIG) was established, the concept of programming seemed rather robust and clear. With Turing's characterisation of algorithms at its core, its meaning could be safely broadened to include the many peripheral activities surrounding the production of software that – as can be inferred from the Call for Papers – have become the natural habitat for PPIG. Today, the nature and status of programming is much murkier. For instance, in his state-of-the-art reflections on programming [13], Chris Granger poses the questions *What is programming?* and *What is wrong with programming?*. Despite the great promise – and perhaps even greater expectations – for applications of computational thinking, programming issues that were perceived as of the essence are no longer so prominently represented in computing-in-the-wild. For instance, the programming languages SCRATCH and Go – with radically different motivations – shift the focus from programming paradigms, formal programming language semantics and sophisticated abstract programming constructs to highly pragmatic issues relating to the experience of the developer. This paper makes the case that in understanding this intellectual development in approaches to program-related activity it is helpful to stop stretching the notion of 'programming' ever wider and to appeal instead to a broader complementary concept. To this end, in this paper, we propose to confine the term 'programming' to the authentic meaning it acquires through Turing's characterisation, and address the wider agenda to which it has been enlisted – and through which its meaning has been adulterated – by introducing the notion of 'making construals'. This proposal reflects twenty-five years of parallel development of Empirical Modelling, an approach to computing that has taken much inspiration and encouragement from the PPIG agenda [7, 9, 11].

Introduction

PPIG stands at the intersection between computing and psychology. Its primary concerns are the *psychology of programming* and *computational aspects of psychology*. When PPIG was conceived in the late 1980s, it was particularly fashionable to regard *computation* as central to the connection between computing ("computational thinking" [23]) and psychology ("the computational theory of mind"). On that basis, it might have seemed that research on programming paradigms and methods of formalising software development would prove to be the driving forces behind subsequent new developments that have brought computing to the forefront of so many people's lives.

In practice, the vibrant culture of computing-in-the-wild that has been so well-represented in PPIG conferences over the last twenty-five years seems to have thrived on a raw pragmatism that admits only a vicarious relationship with its abstract computational roots. For instance:

What is the role for standards and formal specifications of software systems? – witness the pragmatic way in which the communications software behind the internet has been developed.

What is the contemporary status of Codd's elegant mathematical relational model for databases? – witness the logically flawed but pervasive implementation of SQL relational databases, and the adoption of alternative database models [5].

What prospect is there that new programming paradigms can resolve the problems of software development? – witness the sterility of the forty year old controversies that have surrounded different paradigms and the rise and demise of OOPSLA.

The disconnect between theory and practice reflected in these questions and responses has great relevance for PPIG – it is a symptom of topical concerns about the nature and status of programming itself. In a very recent post [13], the software developer Chris Granger identifies problematic aspects of programming and poses the question *What is wrong with programming?*. After talking to about four hundred people with a wide variety of perspectives on programming, he concludes that 'programming is our way of encoding thought such that the computer can help us with it' and what is wrong with it is that programming is *unobservable*, *indirect* and *incidentally complex*.

Granger's conclusions are revealing. They point to technical issues to be addressed in developing programs of just the sort that have been the focus of interest for the PPIG community. They also highlight a more fundamental conceptual problem. Programming has unambiguous historical roots in the work of Turing; in no way is programming intended for 'encoding thought' – it is what is appropriate for specifying algorithmic behaviours conceived by 'a mind following rules' [20,6]. And whereas Granger is here observing Bret Victor's injunction: "we must change programming" [22], it is more appropriate to seek an alternative concept – *complementary* to programming – to account for computing practice.

Our title and the character of our message take their inspiration from Charles Dickens's *A Christmas Carol*. Though the celebrated ghosts who haunted Scrooge had a common mission, their perspectives were individual: they surveyed the past, the present and the future and found in each a different provocation. In a similar spirit, the three perspectives in this paper – past, present and yet-to-come – reflect a common aim but are informed by different kinds of experience. These are broadly correlated with the experience of the three authors of, respectively: finding ways to teach programming to novices that can provide an appropriate foundation for a mature understanding of computing; establishing a conceptual framework for studying programming that does fuller justice to computing-in-the-wild; making the transition from specialist work on software integration to teaching music.

The Ghost of Programming Past

In which the Ghost of Programming Past visits schools to find old initiatives for teaching programming renewed – and facing familiar challenges, and is perplexed by new ventures that sidestep the wisdom of the ancients to promote programming as a lived experience.

Teaching computer programming – yet again!

Understanding and teaching computer programming has been a central concern for PPIG. The challenges that this presents have been highlighted by topical developments in UK schools.

The status of Computer Science (CS) in UK schools is changing, with the challenging nature of the discipline becoming more widely recognised. In January 2012, the UK ICT national curriculum was disapplied, signalling the shift away from ICT skills, e.g., teaching students how to use proprietary software packages, towards a broader range of CS skills and knowledge. These changes have been endorsed by initiatives such as *Computing at School* that are committed to building resources and support networks. However, understanding the requirements of both teachers and students is complex and time consuming.

Adopting an enhanced computing curriculum taxes teachers across all levels of school education. The first author's experience of engaging with school teachers demonstrates that a significant number of teachers are struggling to develop the skills and resources required to deliver academic CS content. These difficulties arise for a variety of reasons: many teachers have educational backgrounds unrelated to CS, have to cope with real and perceived time pressures, and lack experience with appropriate pedagogy. Further, few schools have the flexibility or resources to allow teachers the time or funding to attend courses. While myriad websites, resources and forums are available for sharing and discussing, many teachers currently find it difficult to acquire the skills they need and to put this content in the context of a classroom delivery.

Students must be taught to develop the cognitive and computational thinking skills, and not simply skills and knowledge specific to applications or programming languages. Though programming is a

core skill for Computer Science, it is only a preliminary step towards appreciating programming in its relation to the subject. And though concrete experience of what is involved in programming is essential, teaching someone to program in a specific language does not, in itself, convey an understanding of relevant general Computer Science concepts. For example, the process of wrestling with the syntax and semantics of C or Java can *detract* from the idea that programming is based on a set of abstract, language-independent, principles of computational thinking.

Appreciating the place of programming within the broader framework of practical computing is even more challenging. In this context, computer programming can be viewed from two perspectives:

- as an abstract activity concerned with devising recipes to establish a specified functional relationship between input and output;
- as a core activity for the implementation of software applications, where the concrete real-world interpretation of abstract program constructs, such as variables, must inform the practical construction of programs.

A further distinction must be made where software development is concerned. If the design of a software application is routine then the machine-like agency that is exploited in programming is easily identified and stable in character. Such applications are amenable to the principles and techniques of computational thinking, where the programmer's task is to figure out the machine instructions that specify a required behaviour. If a software application demands radical design, the processes of prescribing computations and of identifying the supporting agency are entangled in such a way that a less constrained way of conceiving programming activity is required. In effect, the construction of software must proceed in parallel with learning about the application domain and the empirical identification of the machine-like agency to support its implementation [8].

Towards programming as a lived experience

Grasping the syntax necessary to write a program, and being able to figure out how this prescribes a behaviour, is the first step towards 'understanding programming'. What distinguishes the experienced programmer from the novice is the degree of fluency with which they can connect program code with a behaviour. The novice needs much practice to acquire the ability to interpret a program without having to analyse it laboriously symbol-by-symbol and statement-by-statement and perform a conscious process of interpretation.

In the first instance, understanding a program involves knowing how it prescribes the behaviour of the computer. For this purpose, analysing a program step-by-step is a mechanical process of translation that in principle requires knowledge of how the computer itself works. In practice, the programmer is typically insulated from the details of the machine-level execution through a high-level programming language. The behaviour of a procedural program, for instance, is to be conceived in terms of variables and data structures whose values change. In making this translation to a behaviour, there are specific rules to follow, and these apply in all contexts.

One of the conceptual difficulties faced by the novice programmer is that understanding how program constructs manipulate variable values is not enough. These manipulations of state have to be related to the function of the program, and further interpreted in terms of the application domain. For the experienced programmer, the process of interpreting program constructs as abstract state manipulations becomes routine and potentially subconscious. Making the further connection between the program and its function is more challenging, and this skill is much more difficult to learn. It is important from the very beginning, however, since the learner is soon obliged to consider how to develop a program to carry out a particular task.

It is in this broader sense of 'program understanding' that the psychological aspects of programming are most topical. In interpreting a program in functional and contextual terms, the programmer typically gains clues from the informal characteristics of the program. The layout of the program can be important in recognising the flow of control and the significant phases in the program execution. The names of variables may connect them with observables in the application domain. Comments may be used to document the behaviour and guide the human interpreter.

The traditional emphasis in teaching computing has been on teaching the skills and concepts behind 'computational thinking'. The presumption that underpins academic computer science, and dictates the core characteristics of the ‘new’ computing curriculum for schools, is that computer science education is first and foremost concerned with the fundamental notion of ‘algorithm’ and the profound mathematical insights about the scope and limitations of this concept. On that basis, the most important consideration is that students learn how to express computational recipes for machines in an abstract programming language. The above discussion highlights a complementary perspective on programming activity – one in which the PPIG community has been prominently engaged – where the emphasis is on the moment-by-moment experience of the programmer and the mental processes involved. This shifts the focus from the program as an abstract product to programming as an activity that, from an experiential perspective, can be helpfully likened to a musical performance.

When viewing programming as a moment-by-moment experience, it is helpful to adopt an alternative broader perspective than computational thinking alone affords. This reflects the wide variety and scope of questions that a novice programmer typically has to address, such as: *'what is the syntax error in my code?', 'what if I change the value assigned to this variable at this point?', 'what if I reorder the statements in this loop?', 'how can I modify this statement to achieve the intended effect?', 'what if the input to the program has the following character?'* To answer such questions, the learner needs an appropriately rich mental model of a program that embraces more than abstract logical analysis alone affords. The computer itself can give direct and immediate support for such mental models. This is illustrated in many contemporary environments to support learning programming:

The Scratch environment [19]: In Scratch, a program is assembled using a visual interface, thus eliminating syntactic errors and complex syntax. The appeal of the environment owes much to the fact that pre-programmed input and output methods for accessing devices such as sensors and actuators and resources such as video can be readily integrated into programs without specialist knowledge. Scratch programs are relatively unsophisticated but can be readily published to social networks.

The W3Schools environment for web development [24]: One of several similar environments to enable non-specialists to develop and program webpages, W3Schools supports editing of code fragments in conjunction with the webpages they describe, conveniently set up so that a programmer can experiment by modifying code and observing the impact. This exploits a distinctive characteristic of browser environments: their robustness when programs are syntactically or logically incorrect.

Bret Victor's Learnable Programming (LP) environment [22]: Victor has developed a remarkable prototype environment to convey his vision for 'learnable programming'. This environment focuses squarely upon supporting the novice in making the direct association between code and meaningful behaviour. Victor's vision is close in spirit to the idea of programming as performance, as is illustrated by the way in which his virtuosic real-time demonstrations incite his audience to applaud [21].

All three environments illustrate a trend towards giving greater prominence to the actual experience of the programmer and the way in which this informs program development. Characteristic of this experience is its holistic nature. The programmer interacts in one and the same environment to modify the programming code, to test the effect of executing a code fragment, to see whether the program gives plausible output on different inputs, to inspect the behaviour of the program and correlate this with the code state-by-state. Throughout the development, the programmer shifts between using and developing roles in the spirit of Papert's constructionism [17]. The aspiration for the environment is to support the development of a program through self-directed learning conducted by experimenting with the code and observing the impact of changes immediately. Shifting the focus from thinking about behaviour in purely mechanistic computer-oriented terms to attending to how meaningful changes are effected is a critical step towards meeting this aspiration. This shift is particularly well-represented in Victor's work, where there is strong support for freely modifying variable values and observing the effect as in spreadsheet-style dependencies. For instance, as is illustrated in Victor's serendipitous discovery of a visual animation effect that moves the blossom on a tree as if in a wind [21], changes that are first explored manually with the idea of comprehending the code can be re-purposed through automation. But the constructionist ideal cannot be realised without setting to one side an essential characteristic of *programming*: prescribing a recipe to achieve a functional goal [2].

The Ghost of Programming Present

In which the Ghost of Programming Present points out the direct relevance of established principles for making construals to the challenge of realising programming in lived experience, and the merits of adopting an empirical and pragmatic semantic stance that is broader than computational thinking.

A conceptual framework for 'programming as a lived experience'

The conceptual framework of 'computational thinking' is not in itself sufficient to understand the character of the support for the novice programmer's mental models represented in the emerging programming environments mentioned above. The focus in such environments, unlike Turing's focus, is on the states of mind of a person *devising* rather than following rules. By way of a simple illustration, in Victor's account of creating the effect of wind on blossom, he is evidently concerned with the experience and support that the computer offers in the process of developing a program whose precise functionality is emergent. What works depends on pragmatic considerations such as the speed of the processing, the colour and resolution of the display and the cultural background and imagination of the human observer. As Victor himself emphasises, the response to the feedback from the experimental construction is more than prosaic 'modelling of a requirement' – open-ended exploration of what is feasible and effective by way of dynamic visualisation is involved.

To express the creative nature of the role being attributed to the 'programmer' in this setting, the term 'maker' is adopted. The maker engages with activities that reflect an uncertainty about the scope and potential for program development – and indeed for computer application – that can derive from many sources. For instance, it may stem from the status of a task as (e.g.) as yet ill-specified, or essentially concerned with experiential issues, or presenting a challenge in radical software development, or acknowledging a novice's limited understanding and prior experience of the capability of a machine. The key components representing in 'making' are *constructing* and *learning*, and this choice of term reflects the essential creative aspect of constructionist learning.

To do fuller justice to the experiential aspects of programming activity, it is helpful to invoke the modelling framework that has been developed for Empirical Modelling (EM) [25] depicted in Figure 1 below. The true significance and nature of the concepts represented in Figure 1 will be clarified later, when they are applied to programming as approached from an EM perspective.

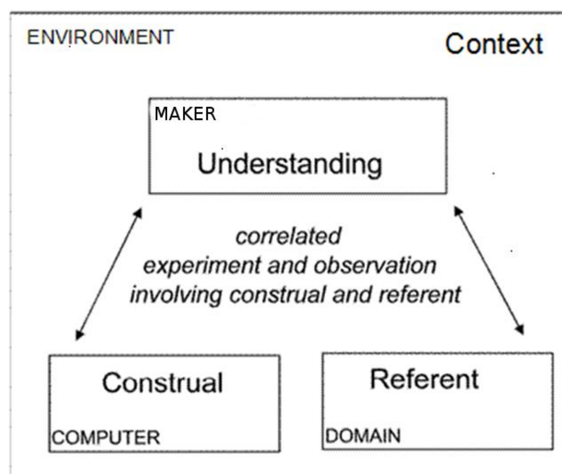

Figure 1 Making an EM construal

It is helpful to give a brief characterisation of the four principal ingredients depicted in the diagram: the *construal*, its *referent*, the maker's *understanding* and the *context*. Figure 1 should be interpreted as depicting the way in which these ingredients are at all times related, though all four evolve throughout the process of learning and development. With reference to Figure 1:

The **construal** refers to the computer-supported artefact within the programming environment that helps the maker to realise their mental model. The scope of the construal reflects the many ways in which the maker can observe and interpret the program and modify it interactively. It will typically

include an environment for editing the program code, assembling the program from visual components (cf. Scratch), allowing the values of variables to be reassigned using sliders as in Victor's LP environment, observing the impact of changes to program code upon behaviour etc.

The **referent** is the behaviour associated with interaction with the program as apprehended by a user. Depending on the current context, this may be behaviour in the application domain (e.g. the rippling movement of blossom on a tree), or a more prosaic machine-oriented interpretation of behaviour.

The maker's **understanding** takes the form of tacit personal knowledge of skilful interaction, familiar interpretations, and expectations that the maker acquires through experience of interaction with the construal and with its referent. This typically evolves moment-by-moment as the program development and use proceeds, and goes beyond knowledge of abstract logical relationships.

The **context** for interaction with the construal and its referent is dynamic and potentially volatile. It reflects the motivations of the maker from moment-to-moment, and the roles they adopt. The variety of ways in which the maker can interact is well-illustrated in Victor's LP environment [22], where sometimes interaction is for the purpose of demonstrating what has been established, is sometimes open-ended experiment, sometimes concerns the way in which program code is framed, is sometimes carried out with the interpretation of the execution of a programming construct in mind, etc.

The correlation between interaction with the construal and its referent can take many forms. Expert functional programmers may be most adept at looking at a functional program and 'instantly' inferring what abstract input-output relationship it prescribes. Their expertise is of the same kind that applies to skilful programmers using any paradigm: they can configure code in such a way that its behaviour is intelligible and amenable to adaptation in meaningful ways, they can readily conceive program code that can realise a given behaviour. Adopting an EM perspective involves taking proper account of the experiential character of programming. Whatever programming paradigm is involved, the expert programmer has to be able to recognise the connection between 'looking at the code' and 'imagining a behaviour', and this involves correlating two independent experiences. As Victor remarks in [21], this process of correlation is easier to interpret in experiential terms if it can be represented using explicit visualisation as a relation between states rather than behaviours (cf. the way in which the movement of a character in a game is animated using his environment by first creating a time-independent representation [21]). The shift in emphasis towards learning through interaction with artefacts that this entails naturally leads Victor to invoke Papert's constructionist stance (cf. [17]).

More about construals

The character and effectiveness of environments that use visualisation and interaction to help the programmer appreciate the relationship between program code and the behaviour it prescribes depends critically on the programming paradigm, programming language and specific details of the program code. A program devised by a novice may prescribe the correct behaviour, but do so in a way that is hard to understand and difficult to adapt. Were we to experiment by changing the values of the variables within such a program in the spirit of Victor's LP environment, the effects might be hard even for the programming expert to predict. Declarative programming techniques were first introduced because of the perceived difficulty of imagining the behaviour of procedural programs and relating this to meaningful behaviour in the application domain. This difficulty cannot be attributed solely to bad programming practice. For instance, some of the most ingenious and efficient algorithms achieve their results in ways that render them brittle and hard to understand. This characteristic of procedural approaches is not easy to reconcile with the fact that Papert advocated the procedural language Logo as a vehicle of constructionism [17,10,2].

More insight into this problematic aspect of traditional programming can be gained by looking more closely at what studying the experiential aspects of programming activity from an EM perspective entails. The key concepts in EM are *observables*, *dependencies* and *agency*. An **observable** is something to which the maker can ascribe an identity and current status. A **dependency** is a predictable synchronisation of changes to observable values that is perceived by the maker. The current state of an EM construal is determined by the current status of its observables and changes to this state are effected by redefining the status or values of observables or reconfiguring the

dependencies to which they are subject. All such changes of state are attributed to **agents** – these include the maker and agents whose interventions are automated.

These concepts can be interpreted with reference to Figure 1, as it applies to the development of conventional programs within environments of the kind mentioned above. Within such an environment, the observables would include syntactic characteristics of the program code, such as the variables and their names, the code layout and the components of the program structure in so far as these could be stably identified, and the attributes – size, colour, style – of the font in which the code was presented. They would also include the status of any of the interface widgets for manipulating the code, and features of any diagrammatic representations that assist in the visualisation of behaviour. An example of a dependency might be the relationship between the position of a slider and the value of variable as in Victor's LP environment [22].

The correlation between the construal and its referent depicted in Figure 1 is between the observables, dependencies and agency in the construal and the referent. In effect, there should be metaphorical counterparts for the patterns of observables, dependencies and agency encountered in the referent within the construal. It is typically quite difficult to account for the relationship between a procedural program and the behaviour it prescribes in these terms. Features exemplified in Victor's LP environment can be helpful in this respect, but a more radical shift in perspective is ideally required.

Empirical Modelling

The interpretation of Figure 1 takes on an entirely different character if we substitute a spreadsheet for a program (cf. Granger's observations in [13]). In a spreadsheet, cells and their values are 'live' observables with which the maker can interact 'as-of-now', and the definitions of values in formulae express dependencies latent in 'as-of-now' interaction. In these respects, the cells of a spreadsheet typically have direct counterparts in the external world to which they relate in a live fashion amenable to experimental validation. By contrast, the correspondence between variables in program code and meaningful observables in the application domain typically has a nominal abstract quality and may be sketchy or tenuous. As the use of spreadsheets as an instrument for *what-if* analysis confirms, there is a very direct correlation between interaction with a spreadsheet and interaction in the world. The very formulation of the spreadsheet reflects what we understand as sensible interaction in the application domain. And whereas it is hard to predict whether changing a variable in a procedural program has a familiar, plausible or absurd real-world interpretation – or indeed admits any interpretation at all, possible ways of redefining the cells of a spreadsheet afford just such nuances of meaning.

EM is a practice in which the focus is on making construals using specially-adapted principles and tools [25]. The main tool developed for EM has been the EDEN interpreter. This allows the values of observables of diverse kinds to be freely specified by definitions similar in character to the definitions of cell values in a spreadsheet. A family of definitions framed in this way serves as an interactive environment that corresponds to an external referent in the manner that is depicted in Figure 1. Many varieties of agent interaction can then be realised by redefining observables. Such interactions are typically first implemented manually through the direct intervention of the maker. Program-like behaviours can be identified by refining the family of definitions and exploring patterns of agent interaction that can admit appropriate meaningful interpretations. In this respect, making a construal is a more general activity than writing a program. It has the virtue of enabling the user to frame the manual and automated agency involved in implementing the program with explicit attention to which observables are changed and what dependencies these observables are subject to. This is consistent with the need in general in programming reactive systems to engineer the reliable interaction between agents that is taken for granted when programming a conventional computer [8].

For a full discussion of EM, the interested reader may refer to [4] and related publications on the EM website [1-11]. A noughts-and-crosses construal from which a suite of programs for playing games with a family resemblance to noughts-and-crosses, developed using EM principles and implemented using EDEN, was presented as an illustrative example of EM in the 6th PPIG conference in 1994 [9]. A version of this construal that has been re-implemented using JS-EDEN, an online version of EDEN currently under development, can be accessed via the JS-EDEN interpreter [26].

The Ghost of Programming Yet to Come

In which the Ghost of Programming Yet to Come champions music as the inspiration for blends of human and computer agency beyond the user model, connects large-scale software integration with a paradigm for personalised music-teaching, and notes the virtues of 'making construals' as a better instrument than 'programming' for making music.

Computing and music-making

When viewing programming activity as resembling performance, it is natural to make analogies between writing programs and composing music in an improvisatory style. Bret Victor makes this analogy when discussing learnable programming [22]: *"An essential aspect of a painter's canvas and a musical instrument is the immediacy with which the artist gets something there to react to. A canvas or sketchbook serves as an 'external imagination', where an artist can grow an idea from birth to maturity by continuously reacting to what's in front of him."*

Such an analogy invites further analysis and critical evaluation. Nothing in the abstract theory of computation refers explicitly to the idea that a machine offers experiences to the programmer in the way that a musical instrument *of its essence* does. As commentators on Victor's LP essay have observed, the concept of responding to the visual feedback that the program affords makes a great deal of sense for a JavaScript program that specifies an animated image on a webpage, but does not apply so naturally to typical systems programming tasks. The element of performance in Victor's manipulation of the JavaScript code that is so much in evidence in his *Inventing on Principle* talk [21] relies crucially both on the character of the programming environment and the precise nature of the interactive transformations he performs. As Victor himself makes clear when criticising the environments for learning programming developed by the Khan Academy [15], the principle of changing parameters in program code and observing the consequences makes sense only when used in a context that is suitably engineered and annotated. Without imposing such a context, the practice of editing program code and 'seeing what happens' typically leads to confusion and incoherence. The notion that certain principles are being respected in this empirical approach to developing programs is implicit in Victor's claim that LP is an archetype for constructionist learning in Papert's sense.

Central to Victor's LP is the idea that a programmer should be able to understand the correlation between program code and the behaviour it effects in a manner that can be traced directly to their immediate experience. As has been argued in many EM publications [2,4,5,10], processes of construction that rely on establishing relationships that are directly experienced are more appropriately conceived as 'making construals' rather than 'writing programs'. Substituting "making a construal" (as described in a previous section) for "writing a program' resolves those discrepancies that make the analogy between creating artefacts by computer and playing a musical instrument problematic (cf. [3]). Construals are essentially concerned with exploiting the computer as a source of interactive experience; they establish a playground for agency that subsumes the roles of learners, developers, users etc and allow these to be blended and distinguished at the maker's discretion; their construction reflects the way in which what can be directly experienced evolves through the refinement of representations and acquisition of skills on the part of human agents. In these respects, substituting 'making construals' for 'writing programs' liberates far richer visions for human agency within computing environments than is enfranchised by the notion of 'computational thinking' alone.

Software architecture and principles for music teaching

Further insight into the parallels between computing and music can be gained from the third author's experience first as a software engineer specialising in software integration and subsequently as a music teacher.

In the development of software solutions, a major challenge is to introduce software into an existing environment in such a way that it will integrate with many different applications on many platforms with diverse configuration. For large software systems, this presents problems that cannot be resolved simply at the specification level; the complexity of the interactions between software components and the diversity of the environments is such that a systematic iterative process of implementation, testing

and refinement has to be followed. The Google developer Rob Pike's keynote talk about the programming language Go at SPLASH 2012 [18] highlights the pragmatic nature of the issues that are topical in this context (cf. slide 11, which lists concerns that are not addressed by language *features*). For instance, a key goal in designing Go and its development environment was to reduce the compilation time for systems so as to make the iterative process of deployment more efficient.

At this high level, software engineering can be identified as the deployment of a 'system' in an 'environment'. By analogy, teaching an instrument can be seen as imparting a skill set to the learner. The skill set can be seen as the counterpart of the system, and each learner as a specific environment. Within the skill set, each skill has similar status to a component of the system. The dependencies between skills guide the way in which content should be delivered to the learner in much the same way that dependencies between components of a system guide its architecture and design.

Just as a system is more than its set of components, and serves some overall functional aims, the skill set to be imparted to the learner is directed at achieving specific goals, such as being able to play a particular piece, or selection of pieces within a particular idiom. In implementing a system, designing the set of components that will be deployed for minimum viable product is the starting point, and the entire architecture of the system is also understood, developed, and tested for each environment as time goes by. In a similar spirit, it is appropriate to identify a basic skill set that is needed to master a particular repertoire, and to consider what is involved in teaching this to each individual learner.

This establishes a parallel between the role of a software engineer in system architecture and design and the role of the instrumental teacher in planning the delivery of content.

Once the system architecture has been designed, the implementation and installation of the software can begin. A particular coder then takes responsibility for the implementation of a component. The instrumental teacher takes on a role resembling that of a coder when devising ways in which to impart a specific skill – for example, framing a plan to teach reading the notes using a mnemonic such as 'FACE corresponds to the spaces in the treble clef' to a student for the first time. In the same way that a coder develops a build process that sets up a component for a specific environment, the instrumental teacher has to devise a plan that takes account of the particular characteristics of the learner.

Once the skills have been deployed and debugged, a specific objective such as a performance of a piece can be addressed. Preparing for performance is itself an iterative development process. The coder attempts to execute the component on the environment, debugs the experience and then refines the implementation. In a similar spirit, the learner performs music in accordance with the guidance of the teacher, then refines the performance with the help of the teacher in response to the experience.

The bugs that arise in the software engineering process take several forms, each of which has a counterpart in the instrumental teaching context:

- the design of the component as a concept might itself be problematic – using mnemonics to learn to read music frequently introduces new bugs to the environment, and it is also often something that the student has covered before. There will be an existing installation of 'using mnemonics to read music' with which this installation might conflict. A decision needs to be taken about whether to try to integrate the two versions or to teach the same thing using an entirely different design e.g. reading music using no mnemonics;
- the implementation of the component is problematic e.g. once the software has been deployed it appears to function correctly at first glance, but intermittent problems emerge, revealing underlying conditions in the student environment that were not addressed in the design and implementation of the component that need to be debugged; the component must then be revised and redeployed, or replaced with a different approach;
- the delivery of the component is problematic e.g. the component is well designed for the student environment, but it was installed at a time when other crucial software was not running properly, or not running at all, or there was not enough processing power available. For example, the student was busy running another program about something else when the topic was introduced, or visual observation was required for the topic to be absorbed but their eyes were too tired;

- the environment requires pre-requisites to be installed before the component can be deployed e.g. the student needs to be familiar with the patterns and shapes of a stave and a clef before a technique for labelling parts of it can be deployed.

Perspectives on human and computer agency

A pivotal question for PPIG is the extent to which human agency can be conceived as machine-like. Some schools of thought take the extreme view that all human activity admits a rule-based account, including activities such as music-making and composition that are deemed to exhibit creativity [12].

Likening a learner to a computer environment may appear to be endorsing the idea that teaching music is like programming a machine. Certainly, a performer can ostensibly behave like a machine. Learners can be taught to "play Mozart" etc by being trained in the appropriate repertoire of skills (playing scales and arpeggios, learning classical harmonic progressions etc) required for this purpose. There are no doubt schools of instrumental teaching that could with some justification be seen as *programming* musicians to perform in a particular way.

The analogy between 'learner' and 'computer environment' here has a different significance. The qualities of computer environments are too difficult to predict and impossible to preconceive for the challenge of effective cross-platform software integration to be met by aligning all environments to a universal norm. The individual characteristics of environments have first to be discovered through empirical testing, and adapted only where necessary or desirable. A good integration strategy then exploits the distinctive pre-existing features of the environment itself. In this same spirit, music teaching can enable students to integrate new knowledge and skills with those they already have.

The peculiar blend of human and computer agency that making a construal affords is in fact well-suited to supporting teaching of this nature. Making construals gives computer support for sense-making that can in principle be applied to explore the particular individual characteristics of learners, and to identify their pre-existing skills, limitations, knowledge and misconceptions. It can also be used to train students in basic musical skills. For instance, a construal devised by the third author in connection with teaching students to read different clefs at the keyboard comprises a stave in which the bottom, middle and top lines are highlighted in bold . For each clef there is a direct correspondence (a "dependency") between pairs of adjacent bold lines on the stave and clusters of consecutive notes that lie under the five fingers on the keyboard. Such a construal hints at the subtlety of the dependency relationships that become second nature to the accomplished pianist, which may (for instance) connect the shape of the hand with patterns of notes on the page, or the lateral movement of the hand over the keyboard with up-and-down trajectories of the melodic line.

As the open-ended nature of the noughts-and-crosses construal discussed in [9] illustrates, there can be great subtlety in the human agency that underpins what appear to be simple tasks. In contexts where this agency is inconceivably rich and multifarious, making construals is more appropriate than programming as a means to enhance human agency precisely because of its essentially provisional and pragmatic character. For instance, the quality of the agency involved in playing a musical instrument is much more than acquiring a repertoire of reliable machine-like skills and learning to apprehend specific dependencies. A fine instrumentalist is far more versatile than a machine, has characteristics that are highly individual that reflect distinctive physical attributes (e.g. size of hand), strategies for performing (e.g. sight-reading vs memorising vs improvising), and capacities to interpret different modes of representation (cf. reading from solfa notation, guitar tablature or a figured bass). It is also self-evident here that the representation of music, the characteristics of the instrument itself, and the cultural context in which it is presented is critically important: cf. the practical impossibility of playing from a MIDI file or emulating quarter tones on a piano.

Teaching that promotes such an exalted view of musicianship has *giving the learner the capacity to make their own construals* as its goal. The end products to which such teaching aspires are motivation and enjoyment rather than accomplishment alone. These qualities, so difficult to sustain in a rule-based goal-oriented culture, play an invaluable role in giving meaning to our musical and human life.

Concluding remarks

The second author first ventured to pitch the ideas that developed into EM at the first PPIG. These ideas have since matured out of all recognition, especially through the contributions made by many generations of research and undergraduate project students. Particularly relevant to PPIG is the fact that the philosophical stance that best suits EM is ‘radical empiricism’, as conceived by William James, the ‘father’ of American psychology, at the beginning of the last century [14,1]. It is particularly significant that the direct associations between ‘experience of the program code’ and ‘experience of the behaviour to which it refers’ to which Victor aspires in [22] can be construed in Jamesian terms as empirically given “conjunctive relations”. This decisively shifts the focus in viewing an interactive computer artefact from the abstract formal semantics derived from the program text to the meaning as it is established and evolves through direct experience of skilful interaction. This perspective, so crucial for the musician [3], is also most congenial to the culture of PPIG.

Looking to the past, it is apparent that what we understand by and expect of 'programming' has evolved to a point where serious reconsideration is required. The role of programming languages in particular needs to be reappraised, as the practices that surround constructing programs as texts are ill-suited to the purposes to which 'programming' is being recruited.

Looking to the present, there is – in Empirical Modelling – a better conceptual framework in which to view the exceptionally rich ways in which computing is being applied. This goes beyond targeting different varieties of use, or modes of development, of software that are predicated on delivering specific functionalities by exploiting established machine-like agencies. The computing activity that is associated with programming in the narrow sense now embraces support for crafting human roles beyond those of users, programmers and developers in conjunction with the agency that formerly was assumed to be given by computing platforms.

Looking to the future, computing activity must increasingly acknowledge the personal agenda of individuals in the same spirit that musical instruments and skills have to be adapted to the personal characteristics of learners, performers and composers. Turing's model of 'a mind following rules' has its place in this setting, but cannot be invoked as a way of developing and validating such rules.

This is not to deny the power and potential of a computationalist stance. Beyond question, activity that is in the spirit of programming in its authentic meaning has a secure role and glorious future prospects in contemporary culture. There is no knowing what can be achieved through the application of computational thinking backed by technological advances and algorithmic ingenuity. No doubt we shall be surprised by future applications for rule-based goal-oriented activities and new contexts in which human agents can act in the role of users.

Nevertheless: our humanity is more than can be expressed in such terms. As has been argued by Jaron Lanier [16], the pervasive status of computing and our limited conceptual grasp of its nature are in danger of imposing a computational thinking paradigm upon areas of life where it is out of place, with damaging social and psychological effects. We need a framework for thinking about computing that acknowledges the scope for more exalted forms of human agency than ‘using a computer program’. No one is better placed to respond to this challenge than the PPIG community.

Acknowledgment

We are indebted to Steve Russ for helpful comments and to Tim Monks, Nick Pope and Joe Butler for their work on the design and implementation of the JS-EDEN interpreter.

References

[1] Beynon, W. M. (2005).. Radical Empiricism, Empirical Modelling and the nature of knowing. In (ed. Itiel E Dror) Cognitive Technologies and the Pragmatics of Cognition: Special Issue of Pragmatics and Cognition, 13:3, December 2005, 615-646.

[2] Beynon, W. M. (2007). "Computing technology for learning – in need of a radical new conception." Educational Technology and Society 10.1: 94-106.

[3] Beynon, Meurig. (2011). From formalism to experience: a Jamesian perspective on music, computing and consciousness. Chapter 9 in Music and Consciousness: Philosophical, Psychological, and Cultural Perspectives (ed. David and Eric Clarke), OUP, 157-178

[4] Beynon, Meurig. (2012). Modelling with experience: construal and construction for software. Chapter 9 in Ways of Thinking, Ways of Seeing (ed. Chris Bissell and Chris Dillon), Automation, Collaboration, & E-Services Series 1. Springer-Verlag, January 2012, 197-228

[5] Beynon, Meurig (2012). "Realising software development as a lived experience." Proceedings of the ACM international symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, October 21-25, Tucson, Arizona, USA, 229-244

[6] Beynon, Meurig. (2013). Turing's approach to modelling states of mind. In Alan Turing – His Work and Impact (eds. S Barry Cooper and Jan van Leeuwen), Elsevier, 85-91.

[7] Beynon, W.M., Boyatt, R. and Chan, Z. E. (2008) Intuition in Software Development Revisited. In Proc. 20th Annual Psychology of Programming Interest Group Conference, Lancaster University.

[8] Beynon,W.M., Boyatt, R.C. and Russ, S.B. (2006). Rethinking Programming. In Proceedings IEEE Third International Conference on Information Technology: New Generations (ITNG 2006), April 10-12, 2006, Las Vegas, Nevada, USA 2006, 149-154.

[9] Beynon, W.M. and Joy, M.S. (1994). Computer Programming for Noughts-and-Crosses: New Frontiers. Proc. PPIG'94, Open University, 27-37.

[10].Beynon, W.M. and Roe, C.P. (2004). Computer support for constructionism in context. In Proc. of ICALT'04, Joensuu, Finland, August 2004, 216-220.

[11] Beynon, W.M. and Russ, S.B (1992). The Interpretation of States: a New Foundation for Computation?. Proc. PPIG'92, Loughborough.

[12] Cope, D. (2005). Computer Models of Musical Creativity. Cambridge, MA: MIT Press.

[13] Granger, C. (2014). Towards A Better Programming. Available from http://www.chris-granger.com/2014/03/27/toward-a-better-programming/

[14] James, William. (1996/1912). Essays in Radical Empiricism, London: Bison Books.

[15] Computer Programming, Khan Academy. Available at http://www.khanacademy.org/compiuting/cs

[16] Lanier, J. (2010) You are not a gadget. Penguin Books.

[17] Papert, S. (1980). Mindstorms: Children, computers, and powerful ideas. Basic Books, Inc.

[18] Pike, Rob (2012). Go At Google, Available from http://talks.golang.org/2012/splash.slide

[19] Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., Kafai, Y. (2009). Scratch: programming for all. Communications of the ACM, 52(11), 60-67.

[20] Turing, A.M. (1936). On Computable Numbers, with an Application to the Entscheidungsproblem, Proc Lond Math Soc (2), 42, 230-265

[21] Victor, Bret. "Inventing on principle." Invited Talk at Canadian University Software Engineering Conference (CUSEC). 2012.

[22] Victor, Bret. "Learnable Programming." Available at http://worrydream.com/LearnableProgramming/

[23] Wing, Jeannette M. (2006). "Computational thinking." Comm. of the ACM 49.3: 33-35.

[24] W3Schools Online Web Tutorials. Available at http://www.w3schools.com

[25] The Empirical Modelling website. Available at http://www.dcs.warwick.ac.uk/modelling

[26] The JS-EDEN interpreter. Available at jseden.dcs.warwick.ac.uk/master