Fifty years of the Psychology of Programming

Alan F. Blackwell , Marian Petre , Luke Church

PII: S1071-5819(19)30079-5
DOI: https://doi.org/10.1016/j.ijhcs.2019.06.009
Reference: YIJHC 2335

To appear in: *International Journal of Human-Computer Studies*

Received date: 1 February 2019
Revised date: 11 June 2019
Accepted date: 13 June 2019

Fifty years of the Psychology of Programming

Alan F. Blackwell[a], Marian Petre[b] and Luke Church[a]

a The Department of Computer Science and Technology, University of Cambridge, United Kingdom

b Centre for Research in Computing, The Open University, United Kingdom

Abstract This paper reflects on the evolution (past, present and future) of the 'psychology of programming' over the 50 year period of this anniversary issue. The International Journal of Human-Computer Studies (IJHCS) has been a key venue for much seminal work in this field, including its first foundations, and we review the changing research concerns seen in publications over these five decades. We relate this thematic evolution to research taking place over the same period within more specialist communities, especially the Psychology of Programming Interest Group (PPIG), the Empirical Studies of Programming series (ESP), and the ongoing community in Visual Languages and Human-Centric Computing (VL/HCC). Many other communities have interacted with psychology of programming, both influenced by research published within the specialist groups, and in turn influencing research priorities. We end with an overview of the core theories that have been developed over this period, as an introductory resource for new researchers, and also with the authors' own analysis of key priorities for future research.

1 Introduction

This paper is a historical reflection on the field broadly described as psychology of programming (PP), as it has developed over the 50 years of the International Journal of Man-Machine Studies (IJMMS), subsequently International Journal of Human-Computer Studies (IJHCS). We relate this thematic evolution to research taking place over the same period within more specialist communities, especially the Psychology of Programming Interest Group (PPIG), the Empirical Studies of Programming series (ESP), and the ongoing community in Visual Languages and Human-Centric Computing (VL/HCC). The definition of the scope (for this reflection) of psychology of programming, and the situation of psychology of programming within a number of academic and professional communities, are key questions that we consider in some detail below. This reflection is also to some extent, and unavoidably, personal. We have been invited to make the contribution to this special issue because the three of us have each acted as leaders, mentors, programme chairs and keynote speakers within this community over many years. We have particular loyalty to the Psychology of Programming Interest Group (PPIG), which is the longest-standing specialist venue for work in this area, although there are other influential groups that have come and gone, or evolved, over the 50-year timescale of this anniversary issue. We also wish to note at the outset that the three of us all owe a huge debt to Thomas Green, who has personally supervised and collaborated with all of us, was the senior founding partner of PPIG, and continues to make profound and central contributions. Significantly, Thomas was also one of the authors of the first paper published in IJMMS on this topic (Sime, Green & Guest 1973). We would like to dedicate this review to Thomas, in recognition of his academic service and research legacy.

1.1 Scope

As programming languages and technologies change, any review of psychology of programming over these years needs to consider what kinds of software technology and activity should be considered to fall within the definition of "programming". Edge cases might include command line interaction (which can resemble "code"), spreadsheet construction (which does not resemble code, but can be used to create many business applications previously requiring programming languages and skills), or in a previous era VCR programming (which is called "programming", but doesn't involve code generation).

This question has, of course, been considered within the research community itself, in a paper by one of us (Blackwell, 2002a, later elaborated in 2002b) explicitly addressing the question "What is programming" for the PPIG audience. The fundamental argument in that paper is to define programming by contrast to direct manipulation. In direct manipulation interfaces (Shneiderman 1983), user actions have immediate effects that are incremental and reversible. By contrast, in programming tasks, the user makes changes to the system that will have effects at some time in the future, often in ways that will not be reversible. The cognitive demands of that situation are different from those of direct manipulation, in a way that makes the psychology of programming fundamentally different from psychological issues that apply to direct manipulation. If the consequences of an action will take place in the future, this automatically introduces questions of notation, debugging, specification and so on, even in simple cases such as VCR "programming" (thus explaining the casual use of the term). Emerging technologies such as Internet of Things and Artificial Intelligence will continue to introduce usability challenges where users have to act like programmers, even when they have no relevant experience or training - the class of interaction that is described as end-user programming, end-user development, or even end-user software engineering (Ko et al., 2011).

This definition of programming, while theoretically motivated and justified as described later in this paper, is undoubtedly more broad than might be expected from a focus (for example) on professional software development or computer science students. As a result, the more recent decades of HCI research have included a number of topics that share the characteristics of programming, but where neither the users nor the researchers would necessarily use this word. A classic example would be the study of spreadsheet use, which is a core topic in psychology of programming, despite the fact that many users (and many programming language researchers) might not have realised that a spreadsheet is also a programming language. This breadth of scope does have implications for our historical review. We have endeavoured to include many of these programming-like situations in our analysis, but when the work is published by authors who are not aware of the relationship, it is quite possible that some potentially relevant pieces of research have escaped our notice, through lack of relevant keywords in titles and abstracts.

2 Five decades of psychology of programming

To orient our discussion, we have reviewed the archive of published papers in IJMMS/IJHCS, dividing it into five decades in order to identify and discuss the long-term trends and developments in research through this period. The following passages discuss each of these decades in relation to the research that appeared in the IJHCS/IJMMS journal. Later parts of this paper broaden our scope of enquiry to relate these developments to other communities and publication venues that bring more specialist perspectives to the psychology of programming.

2.1 The 1970s (1969-1978): the cognitive work of the programmer

The first paper to directly address the psychology of programming in IJMMS was "Psychological evaluation of two conditional constructions used in computer languages" by Sime, Green and Guest (1973). Widely recognised as a classic, this was also the first paper in the index of those selected for re-publication to mark the 30th anniversary issue of IJMMS, 20 years before our current anniversary celebration. Over the whole of the decade 1969-1978, 14 papers in the journal focused on the cognitive processes of programming, and on opportunities to improve performance through changes to language design. These explored other specific language features, as well as the newly-favoured paradigm of structured programming. Experimental studies investigated errors and debugging, often contrasting novices and experts in order to gain understanding of the mental representations and strategies that were involved in expertise, as discussed further in the section on theoretical developments later in this review. From the outset, there was already interest in understanding how people without programming training might approach programming tasks (Miller 1974), in order either to make languages more natural, or to support programming by non-professionals.

The research methods and theoretical concerns of these early papers reflected the increasing interest in cognitive psychology through the 1960s and 70s, which aimed to achieve a deeper understanding of human learning and problem solving through the use of controlled experiments involving formally-structured problem-solving tasks such as logic puzzles. The computational theory of mind being developed also reflected early achievements of artificial intelligence, where general theories of machine learning and automated problem solving were among the core ambitions. The problems of the software industry therefore presented a valuable opportunity for applying such research to important practical problems, while advancing the emerging agenda of cognitive science that attracted many researchers with an appreciation of both psychology and programming (e.g. Brooks 1977).

During this period, other early studies of the programming profession, such as Fred Brooks' classic The Mythical Man-Month (1975), were demonstrating the ways in which both individual skill and organisational structure were critical to the success of software projects. Psychological research, including work psychology and organisational psychology (Weinberg 1971), offered the potential to gain improved understanding of the problems experienced in such projects, and perhaps evaluate alternative approaches to addressing them (Weinberg & Schulman 1974).

2.2 The 1980s (1979-1988): cognitive models at scale

After the initial foundations were laid in the 1970s, work presented in the 1980s greatly expanded the literature in psychology of programming, with 46 IJMMS papers devoted to the topic. Cognitive models were elaborated and refined, paying particular attention to the skills required by the professional (and student) programmer in order to understand and reason about more complex programs and data structures. Controlled experiments still compared and evaluated design options for specific language features (e.g. conditionals, control structures such as iteration and recursion, or data structures such as arrays and linked lists), exploring these in terms of both semantics (e.g. keyword choice) and syntax (e.g. delimiters and indentation). Program comprehension continued as a dominant theme (e.g., Brooks 1983), including the articulation of the role of 'beacons' in understanding larger code bases (Wiedenbeck 1986). More explicit attention was paid to problem understanding and decomposition, and increasing attention was paid to mental models and problem representation (e.g., du Boulay, O'Shea and Monk 1981). Moving on from the dominant paradigm of structured programming, the first empirical studies emerged to compare procedural and declarative paradigms (Gilmore and Green 1984), along with first steps toward the non-functional (and non-cognitive) attributes of source code described as "program aesthetics" (Leventhal 1988).

Similar themes were evident in the early ESP proceedings, for example with Rist (1986) addressing cognitive models of programs (and the emergence of programming plans - slightly anticipating their appearance in IJHCS) based on studies of novices and experts. Letovsky et al. (1987) identified key behaviours in code inspection: design reconstruction, mental simulation, and document cross-checking. Pennington (1987) added to the literature on comprehension strategies in programming with evidence that cross-referencing the domain world and the program/solution world facilitated comprehension. Spohrer and Soloway (1986) analysed bugs that novice programmers make while solving introductory programming problems, identifying high-frequency bugs and some of the underlying problems that contribute to them. Lewis and Olson (1987) considered barriers to end-user programmers, and identified two tactics for making programming easier for them.

During this period, the founding meeting of SIGCHI took place at Gaithersburg in 1982, where it is notable that 10 of the 79 papers presented at the conference related to human factors in software development or programming. In contrast, at the recent CHI meeting in 2018, while the total number of papers related to programming is about the same (11), this is now only a tiny fraction of the 670 papers in the CHI proceedings.

2.3 The 1990s (1989-1998): professional skills and processes

In the 1990s, IJMMS included another 53 papers addressing psychology of programming, extending attention to the skills that were essential in larger projects, where programmers would need to gather information about code contributed by others, or re-use code that they had not written themselves (e.g. Bellamy and Carroll 1992). Consideration of larger software projects motivated increased attention to the cognitive tasks of problem decomposition, and problem solving strategies framed in terms of programming plans (e.g. Davies 1990), but also more rigorous investigations of cost-effectiveness and efficiency. Studies during this decade extended to many other parts of the software development lifecycle, including requirements analysis, debugging, tracing and maintenance, and the team coordination required in larger projects.

Many of these activities involved specialised notations rather than conventional source code, which drew attention to the properties of such notations that had impact on usability. Increased pluralism of language use, both in education and professional use, motivated comparisons of languages, at both the syntactic level (comparing visual to textual syntax) and further semantic paradigms (now focused on comparison of object-oriented to procedural). Attention also shifted for the first time toward the increasing use of spreadsheets, which could be used to implement many business information systems that would previously have required construction and compilation of source-code.

Broad awareness of differing levels of productivity between professional programmers, combined with the demand for new recruits and trainees, increased attention to individual differences between programmers, including the need to evaluate or predict programming skill at recruitment time. Expert/novice comparisons continued to offer ways of understanding professional skill, and to guide educational initiatives, including empirical evaluation of specific teaching strategies.

Again, similar themes were echoed in the ESP proceedings. Notably, there was increased attention to larger software projects and team behaviour, with significant papers by Flor and Hutchins (1991) on team programming during software maintenance, and by von Mayrhauser and Vans (1997) on debugging of large-scale software. Language pluralism was reflected in ESP as well, with attention to comparison of different programming notations. Both Green et al. (1991) and Moher et al. (1993) , reported comparisons of the comprehensibility of textual and graphical programs, with Green et al. making a case against 'superlativism' arguments, in favour of a 'match-mismatch' position that takes account of information accessibility for a given task.

2.4 The 2000s (1999-2008): the social enterprise of code

The number of papers devoted to psychology of programming in the 2000s was fewer than in the previous decade, with 33 papers in IJHCS addressing these themes - including special issues on the topics of empirical studies of programming (replacing the planned contents of a final ESP workshop after it had been cancelled), empirical studies of software engineering, and a large special issue on social and collaborative aspects of software development.

During this period, research continued on the cognitive tasks that were essential in larger projects, such as comprehension and maintenance activity, but far more attention was paid to the organisational problems in large project teams, considering team coordination and personal dynamics, organisational psychology, and approaches to engaging with users outside the software development team. The rise of agile software development methods, where programming would be far more closely integrated into the organisation, became a clear focus (e.g. Sharp & Robinson 2008). The commercial discovery of "pair programming" introduced not only a need for research into the interaction between programming pairs (e.g. Lui & Chan 2006), but also a research opportunity through the ability to capture and analyse natural vocalisation of programming knowledge and strategy (e.g. Bryant et al 2008).

There continued to be comparative studies of alternative programming languages, now extending beyond the vogue for object-oriented programming to include logic and functional programming paradigms, as well as evaluation of visual language syntax and of the visual notations and diagrams used in other phases of the software development process. Some insights from such research were also explored in relation to novice programmers, for example to inform teaching practice using algorithm animations or other representations.

2.5 The 2010s (2009-2018): code in new contexts

In the final decade of this 50-year survey, only 10 papers on the psychology of programming have appeared in IJHCS. As we discuss later in this paper, the reduced number of such papers in general HCI venues is largely because many specialised communities now exist to carry forward the research agendas established in previous decades - including the large communities of researchers in empirical software engineering and computer science education, each of which now have substantial bodies of research addressing the questions of team coordination, individual differences and training, that are so essential to professional software development work.

In addition, many of the communities focused on the development of new programming languages and software engineering tools have also incorporated psychology of programming theory and research methods into their own research, and we discuss below the resulting bodies of research in human-centric computing, software engineering, and programming language communities. The psychology of programming community itself, including both the PPIG meetings and publications in IJHCS, now often focus on new emerging contexts where programming has become relevant, beyond the well-established concerns of professional software development and education (e.g. Bellucci et al 2019).

In-depth analysis of software engineering processes outside the standard business model, including open source communities, language learning in a social context, and pair programming, continues. End-user business tools such as spreadsheets are considered as exemplars of other ways of programming (e.g. Kankuzi & Sajaniemi 2016). And in this period, small-scale programming has also become increasingly prevalent in people's homes as a consequence of the Internet of Things. These new contexts introduce new cognitive and social concerns, such as the gender dynamics of task-sharing within the home, and the implications of programming tasks within a discretionary situation where people might choose not to do the task at all, or delegate it to others (Blackwell, Rode & Toye 2009). More diverse application contexts and domains also require more diverse approaches to research and system design, with questions from previous decades explored and applied via new methodologies supplementing traditional controlled experiments.

2.6 Reflection on the five decades - the reorientation of HCI

Looking back over the period of time addressed by this anniversary issue, in HCI research there has been broad recognition of three waves or paradigms during this period (Bødker 2015). In broad terms, these waves can be characterised by: in the first wave, cognitive psychology and human factors; in the second wave, social interaction within work settings; and in the third wave, a focus on everyday life and culture. It is apparent from the review of each decade above that the research questions addressed in relation to psychology of programming have also been influenced by broader trends in HCI. Early research in psychology of programming focused on the individual programmer, and sought to construct cognitive models of the processes involved in programming, in a manner that was typical of first-wave HCI. Toward the middle of our historical survey, psychology of programming research broadened into software engineering and paid more attention to the social contexts and collaborative processes that are necessary within organisations and design teams. In recent years, there has been increased consideration of programming within a cultural context, as an artistic practice, or a craft or hobby that is carried out at home. None of these tendencies has involved wholesale reorientation of the field, just as there are still researchers active in each of the three paradigms of HCI. Current research in psychology of programming, for example at the PPIG conference, includes projects reflecting each of the three waves of HCI.

3 Intersecting communities

We have been able to review five decades of research in this journal in a relatively straightforward manner, but one of the challenges in understanding what has been addressed in the psychology of programming *in toto* (and of software design and development more widely) is that the literature is so widely dispersed.

Psychology of programming is a particular focus within HCI, meaning that theory, evidence, and people move freely between these communities (although our pragmatic definition of 'programming' includes only a specific sub-set of HCI studies). Nevertheless, many of the topics addressed within the pragmatic scope of our own survey are also addressed within interaction design - and there are particular points at which interaction design and interface development intersect with programming (e.g. Blandford et al. 1998). On one hand, PPIG (and the previously active community of ESP) may be considered a core psychology of programming research community, because they are motivated by an interest in what the complex endeavor of software development reveals about human cognition and collaboration - and how psychology can inform and assist software development. On the other hand, there are in fact multiple research communities (each with their own core concerns) where psychology of programming might be discussed and research published.

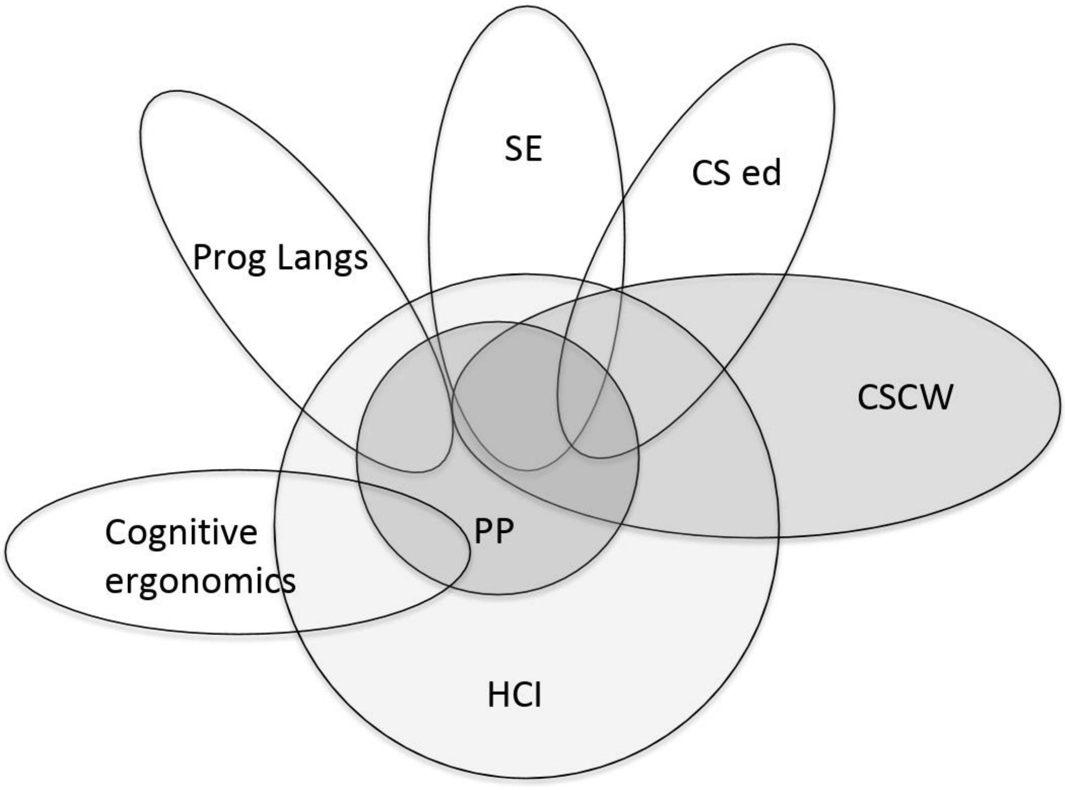

Figure 1: Research communities intersecting with psychology of programming (PP): cognitive ergonomics, programming languages, software engineering (SE), computer science education (CS ed), computer-supported cooperative work (CSCW), human-computer interaction (HCI)

Figure 1 above (and see Appendix 1 with more comprehensive coverage and detail), identifies some of these cognate communities, and the diverse venues in which psychology of programming literature might therefore be found: software engineering, programming languages, computer-supported collaborative work (CSCW), computer science education (CSEd), cognitive ergonomics, and of course HCI more broadly. The table in Appendix 1 is by no means complete, and we see many other intersections even in our own research, including design studies, information systems, information visualisation, knowledge engineering for expert systems and knowledge-based systems, cognitive modelling for intelligent tutoring systems, data modelling, database design and queries, supervisory control systems, interfaces for musical expression, creativity and cognition, semiotics, diagrams, and so on. Such intersections between communities and concerns enable mutual influence between them.

As an example, attention to 'programming in the small' (i.e., coding, reading, debugging, and modifying small, independent programs) was a core focus of early psychology of programming, and has continued to receive attention throughout the half-century, despite other perspectives and concerns emerging to overtake its predominance in the psychology of programming discourse. But this specific topic has by no means declined, in the volume of research that is carried out globally. Much of the current work on 'programming in the small' has shifted into the CS education research communities, where teaching programming, and latterly 'computational thinking' (a policy formulation that appears very closely associated with theoretical developments from the early days of our review), are a focal concern.

These intersections – and their importance – have long been recognized and encouraged by the psychology of programming community. For example, the Psychology of Programming Interest Group not only includes researchers who are also members of these other communities, but also deliberately invites keynote speakers from other domains (software engineering, scientific software, cognitive ergonomics…) and other contexts (industry, primary education, craft) to its annual workshop, specifically in order to promote cross-fertilisation and new perspectives.

Another example of such an 'intersection' is around program comprehension, involving communities with two quite different perspectives. The International Workshop on Program Comprehension (IWPC) started from a software maintenance perspective. Psychology of programming - and its key venues PPIG and ESP - started from a cognitive perspective. The two gradually cross-fertilised. Over time, attention to program comprehension has expanded into software comprehension more broadly, and in broader contexts – not just 'code comprehension' but comprehension of software systems and their contexts. This parallels the increasing scope of software, and the shift from programs to 'systems of systems', as a result of which the psychology of programming community has responded with increasing attention to software maintenance and software/code re-use.

As with CS education, there has been a gradual infiltration of human and psychological perspectives into the software engineering community, around the same time as psychology of programming expanded from its initial focus on programming into the broader concerns of software engineering. CHASE, the Workshop on Cooperative and Human Aspects of Software Engineering co-located with ICSE, which began in 2011, was one reflection of this intersection. We discuss several other such points of intersection later in this review.

4 The Psychology of Programming Interest Group

4.1 A Distinctive Community

One particularly enduring research context for the study of programming from a human perspective has been the Psychology of Programming Interest Group. While the field of programming language design can have a tendency to prefer mathematical idealisation and selective publication as ways to advance the field, more human-centric contributions to programming language research offer a moderating influence supporting cultural change and identification of new priorities. PPIG is one example of where this happens. Unlike conferences whose oversight bodies impose targets for the proportion of submitted papers that must be rejected (the implicit assumption being that good science is dependent on high rejection ratios), PPIG has strongly resisted this assumption.

The practice of the PPIG workshop for many years has been that, even where a research manuscript demonstrates some lack of understanding, it is more productive to invite the author to participate in conversation rather than excluding them from the community. The result has been impressive breadth of enquiry and cross-fertilisation, such that it is not unusual for a paper on the verge of being rejected to lead within a few years to the author becoming conference chair. In the words of Paul Mulholland, himself a wise and supportive past chair of PPIG, “we have a strict criterion for publication – you submit it and we publish it!”

In recent years PPIG has experimented with methods of trying to give more exposure to the insights that contained within reviews, using a system inspired by the London Review of Books (and recapitulating earlier experiments such as the *Journal of Interactive Media in Education* (JIME)) to offer the reviewers the opportunity to share parts of their reviews more publically in the proceedings. This process, emphasizing that the purpose of PPIG reviews is conversation forming rather than that quality control, has directly contributed to other workshops, for example in directly influencing the method used by the Salon des Refuses workshop series at the <Programming> conference of publishing critical responses by program committee members to each submission. It has gone on to have an influence in the design of the PX and LIVE workshops.

It is interesting to speculate whether, in an era of transformation through open access publishing and social media, PPIG represents an unsustainably romantic model of old-fashioned scientific community, or perhaps a distinctive model for the science of the future.

4.2 The Resulting Research

It is interesting to look at the consequence of structuring the research community in this way, and consider the ways in which this unconventional approach to research discourse has influenced the study of programming. Church and Mărășoiu (2016) surveyed in detail the 400 research papers published at PPIG between 1992 and 2015, using the 56 papers from PLATEAU (the Workshop on Evaluation and Usability of Programming Languages and Tools) between 2009 and 2015 as contrast material. Each paper was categorised within an open coding scheme, looking at the types of programmers being studied (end users vs novices vs professionals), empirical methods (e.g. eye tracking), analytical perspectives and theoretical frames applied (e.g. cognitive dimensions) and aspects of programming (e.g. comprehension), technologies (e.g. spreadsheets), and languages (e.g. Java).

The work showed that PPIG as a community has engaged in much more extended study of novices (64%) and end-user programmers (13%) compared to the PLATEAU workshop (35% and 5% respective). The rest of the analysis shows a very wide variety of research interests, varied in methods (Figure 2, taken from Figure 7 in the original), topics of study and technologies.

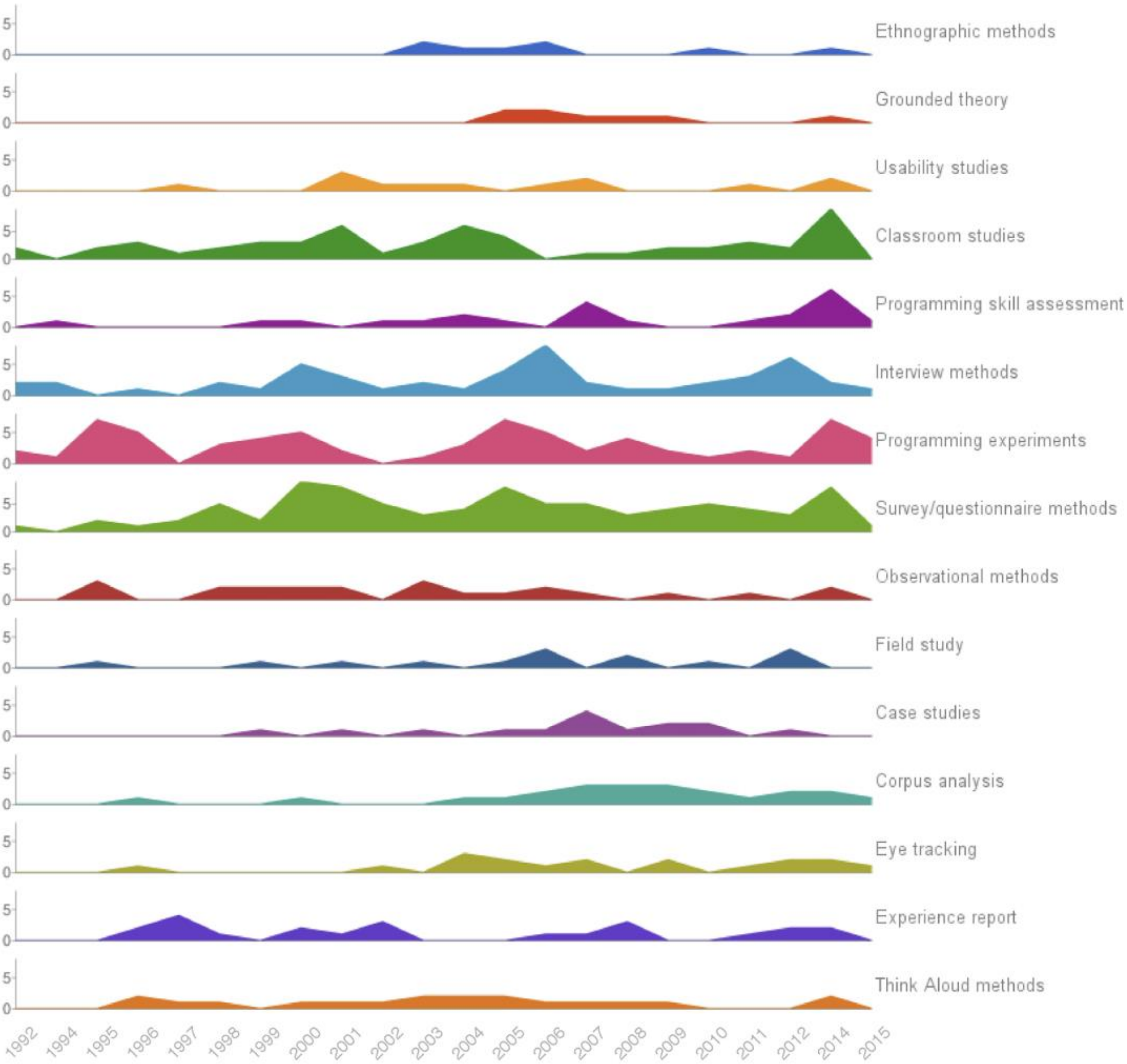

Figure 2: Empirical techniques as PPIG over time. From Church and Mărășoiu (2016), Figure 7. Used with permission.

The breadth of the methods is apparent, and whilst new approaches such as ethnography or grounded theory are introduced, they augment rather than displace existing approaches. The topics of interest follow a similar pattern of retention and expansion. Indeed, a similar accumulation of interests has applied to all of the aspects listed above, with the sole exception of programming languages, where for example, the study of Pascal and Prolog have declined and been displaced by Java and Visual Basic.

This analysis also reveals some interesting quirks. By comparison to PLATEAU, the PPIG community has been more interested in variables, and less interested in algorithms and concurrency. As Blackwell and Morrison (2010) point out, this may be related to the relatively greater focus of PPIG on end-user and novice programmers. Whilst this focus has resulted in novel research contributions, it has tended to result in the community overlooking the study of relevant new technologies, such as machine learning, or of new organisational work practices such as the study of development operations.

Nevertheless, this attitude of openness and encouragement of new ideas has resulted in a community - and hence research - of substantial methodological diversity, enabling it to see the area of study from a broader perspective than is often possible after an extended period, when there may be a tendency to normalise and establish standard methods.

4.3 A note about ESP

We have discussed the intersecting communities that share research concerns with psychology of programming, as characterised in the archive of IJMMS/IJHCS, and also in the very specific community of PPIG. Some of these communities have been far more influential than others, and in particular the early boom in psychology of programming research was closely associated with the series of workshops on Empirical Studies of Programmers (ESP). Many of the figures making seminal contributions to the psychology of programming, including many of those mentioned in our review of the five decades of this journal, also published foundational (and widely cited) papers in the proceedings of ESP. In preparing this review, we have also reviewed the ESP archives, in order to inform discussion of overall trends and theoretical contributions.

In contrast to ESP and other more formal publication venues, the PPIG workshop format and community has provided a supportive environment for speculative, risky, critical and alternative perspectives that frequently run counter to conventional thinking in programming language and software engineering research. A venue of this kind, while undoubtedly valuable for nurturing young researchers and immature ideas, is not likely to attract large audiences or prestigious citation metrics. Any research field also benefits from venues where more mature research can be published in an archival forms. In the early years of PPIG, ESP played this role. More recently, this function has been taken by larger selective conferences such as CHI and VL/HCC, or journals such as the Journal of Visual Languages and Computing, and of course IJHCS.

5 When psychology of programming meets programming languages

A singular demonstration of the changing priority accorded to human factors in programming language research can be seen in the series of IEEE Symposia on Visual Languages from 1988 to 2000 (following an inaugural workshop in 1987). The focus on visual programming languages - specifying program behaviour with diagrams rather than text - followed routine use of diagrams by professional programmers (especially the flowcharts ubiquitous in the 1970s) and systems analysts (in the various approaches to business-oriented planning of data files and processing). However programming languages themselves, from FORTRAN and COBOL on, used teletype characters, with manual translation from diagrammatic designs to text that could be processed by compilers. This textual nature of programming "language" had not been inevitable - indeed the first formal proposal by Goldstine and Von Neumann (1947) was purely diagrammatic. However the economics of interactive graphical displays, despite the early celebrity of Sutherland's Sketchpad (1963/2003) meant that mass-market deployment of graphical user interfaces had only become feasible in the mid 1980s.

In the following 13 years of the IEEE Symposia on Visual Languages until the series ended in 2000, the research presented focused primarily on description of specific languages (129 papers), and on theoretical or engineering frameworks for classes of visual language (125 papers). In addition to these topics that might be regarded as core to visual programming languages, the VL symposia also paid significant attention to visualisation of software and data (62 papers), and other aspects of graphical user interface design more generally (67 papers). Other regular themes included discussion of diagrammatic modelling languages and notations (38 papers), algorithms for parsing, layout or rendering of visual languages (30 papers), and other types of interactive drawing tool (28 papers). Some specialist topics have remained a constant minority interest, including tools for the construction or generation of user interfaces (14 papers) and the use of inference methods for end-user programming by example (11 papers).

However, it is remarkable how few of the papers published in this period of the series paid attention to theories of human factors that justified the change from textual to visual language, or empirical studies that attempted to measure or evaluate the benefits of that change, with only one or two papers each year dealing with those topics. As noted by Blackwell in a 1996 paper, subsequently recognised with an award for most influential paper after 20 years (Blackwell 1996), research in the field relied mainly on theories of pop psychology, folk wisdom, or personal subjective intuition to justify its core technical concerns.

In 2001, the Visual Language research community made a radical change, restructuring as a series of linked symposia on topics in "Human-Centric Computing" (HCC). For the following three years, members of the community proposed symposium topics that were likely to be of interest under this theme. Of the ten symposia that followed, the majority continued the established core interests of the Visual Languages community, with symposia on Visual Languages and Formal Methods, Visual/Multimedia Approaches to Programming and Software Engineering, Visual/Multimedia Programming and Software Engineering, Visual/Multimedia Languages, Visual Languages and Formal Methods, and Visual/Multimedia Software Engineering. But a substantial change in direction was signposted in four of the ten symposium topics, which reflected core interests of the Psychology of Programming community: two Symposia on End-User Programming in 2001 and 2002, a Symposium on Empirical Studies of Programmers (explicitly referencing and reviving the name of the ESP series) in 2002, and a Symposium on End-User and Domain-Specific Programming in 2003.

In the most recent 13 years of the VL/HCC conference series, the changed orientation of the community is quite marked. Over the same duration that the earlier VL conference series had reported 17 empirical studies of programming activity (e.g. Pandey & Burnett 1993), papers at VL/HCC have included 146 primarily empirical studies - and in fact most other papers at the conference also have some empirical element, even where they are devoted to a more conventional concern (for example, a presentation of a new visual language will often include small empirical studies of its usability, or investigation of requirements in its proposed context of use).

The series has also moved from a specific focus on languages with diagrammatic syntax to a broad interest in software engineering tools, including tools for use with conventional text languages, and extensions to visual environments that focus on mainstream questions in software engineering, such as requirements analysis, design patterns, API use, code reuse, debugging, testing, maintenance etc. Over the period of analysis, 98 papers addressed these broad software engineering issues.

The third largest category (73 papers) in this period was dedicated to educational projects, including some design of specifically educational languages or visualisation tools, but also theories of education or learning cognition.

The two periods before and after the HCC transition saw a substantial change in the level of interest in spreadsheets, with 6 papers in the VL series dedicated to spreadsheets, while there were 45 papers in the VL/HCC period. There is, of course, still considerable attention to the traditional concerns of the VL field, although at levels rather reduced by comparison to the early years of the conference. 50 papers were devoted to new visual languages (versus 129 in the early period), and 21 to general purpose models or frameworks (versus 125 in the early period). Several topics continue to be of interest at about the same level - algorithms (20 vs 30), diagrammatic modelling languages (44 vs 38), and visualisation (46 vs 62).

It is also noticeable how the theoretical basis of the empirical studies has shifted away from a universalist conception of human skill, founded in cognitive psychology, toward attention to gender, age, personality and socioeconomic factors. There is also a significant shift from the individual activity of programming toward questions of collaboration both in software engineering, and collaborative problem solving more generally. These changes in emphasis and interest mirror the general tendencies in mainstream HCI research, often characterised as the second and third waves in HCI.

6 Established theory in psychology of programming

Many areas of HCI develop by responding to new technical trends in computer science, exploring the properties of new classes of interactive system through inductive empirical research, rather than by constructing experimental prototypes specifically to test theoretical predictions. The resulting long-term trend has been to create a field that is methodologically and theoretically diverse, but does not construct or evolve a universal 'theory of HCI'. This situation has been described as a 'big hole' in HCI research (Kostakos 2015). To some extent the same is true of psychology of programming, which has responded over these 50 years to specific contemporary trends, including spreadsheets, object-oriented languages, visual languages, agile development and so on. Responsiveness to relevant commercial and technical developments is to be commended, but our survey has also demonstrated a number of more established theoretical perspectives, suitable for application in continued research into programming.

These include the fundamental characterisation of programming as an information processing activity, as pioneered by Ruven Brooks (1977). These models of cognitive activity as information processing architecture were able to draw on generic model components established in cognitive psychology, such as Baddeley's model of working memory, or Johnson-Laird's theory of representations in problem solving. Programming as a complex activity involving problem-solving and planning was able to build on models of goal-directed planning from artificial intelligence, and AI-based theories of knowledge representation, for example in the work of Soloway & Ehrlich (1984), Rist (1986) and Hoc (1979, 1981).

Models from cognitive science also informed major theories of comprehension and learning in programming, such as Susan Wiedenbeck's observation of the way that visual search employed "beacons" within source code to structure and orient information processing in relation to a mental representation (1986). Theories of learning built on cognitive models of learning by analogy, and also on systematic description of non-expert technical understanding drawing on cognitive anthropology (Gentner & Stevens 1993). A classic paper from Ben du Boulay et al (1981) observed that people learning to program had to have some knowledge of how the program would be interpreted through what they see in the language, but that this model in itself necessarily hid further complexity - a "black box within a glass box". These principles have continued to inform design of educational environments and teaching strategies, for example Chee's (1993) analysis of how structural analogy can be used to understand program behaviour.

An unfortunate consequence of increasingly well-characterised models of human cognition during programming was the implication that programming might eventually be treated as a routine activity (whether routinely executed, or a skill that would be acquired through routine instruction). Apparently universal claims about the cognitive essence of programming could easily become conflated with claims of advocacy by the proponents of particular programming languages or paradigms over others, on the grounds that they would be more "natural". This conclusion was not consistent with the earliest findings in the field - including the 1973 paper by Sime, Green and Guest, which had observed precisely the opposite - that the best form of representation for a particular problem depended on the structure of the problem.

The need to recognise, account for, and design for, diversity in programming problems led to revisitation of those early experiments in order to counter the claims of "superlativism" by those who imagined that any particular language or language feature would be universally superior (Green et al. 1991). Green and his collaborators (including authors of this review) commenced a painstaking revision of the field, identifying those design features of programming tools that were cognitively relevant, and relating them to the particular kinds of tasks for which they were beneficial or not (with associated tradeoffs). This "cognitive dimensions of notations" framework (Green & Petre 1996) has become the most widely cited work in the field, and also the most widely cited publication in the computer science journal where it appeared (Blackwell, 2006). As documented elsewhere, the theoretical account of notation use, and design methods within which this account can be employed for formative and summative critique, have resulted in a substantial legacy for all the research into psychology of programming that it drew on, regardless of the fact that the "cognitive" part of this theory is not nearly so relevant as the implicit theory of design that it embodies.

Further theoretical developments have continued in the wake of the turn to notation, but these have focused more on accounting for specific patterns of individual difference, describing the reasons for diversity of behaviour rather than universal knowledge or strategies. One of these theoretical contributions has been the focus on "self-efficacy" among learners, which causes some groups to be disadvantaged through social expectations of their competence. Pursued with great energy by Margaret Burnett, following initial exploration with her student Laura Beckwith (Beckwith et al. 2006), this theory of programming behaviour has broadened into a campaign for "Gender HCI" that addresses questions of expectation and inclusion throughout the design process.

Analysis of the diversity that can be observed among notation users, in combination with broader consideration of end-user programming tasks, led to a behavioural economics model of first steps in programming, described as "attention investment" (Blackwell 2002). Complementing the theoretical accounts of search, comprehension and planning behaviour, the attention investment model offers understanding and design strategies relevant to the fundamental distinction between direct manipulation (with immediate feedback) and programming (planning for the future, as expressed through some shared notation).

7 Future Agenda

We close this reflective review with some crystal ball-gazing, contributed independently by each of the authors to indicate some diversity of opinion in our own interests.

7.1 Alternative views of programming

In addition to the study of established practices in programming and software development, the psychology of programming community plays an important role in nurturing more experimental approaches. These have the potential to move beyond established practices, and redefine what we understand programming to be. Two examples of this are the development of the Live Coding research community, and the creation of the “attention investment” model of interactive tasks that have important cognitive similarities to programming.

Live Coding is an artistic practice involving improvised programming in front of an audience, often to produce synthesised music or generative video art. It is associated with a community of researchers and arts practitioners, originally identified with the “TOPLAP” manifesto, and now focused around the annual International Conference on Live Coding (ICLC). Although the origins of the live coding movement were in the digital media programmes of art schools, two of the TOPLAP manifesto authors made early connections to psychology of programming. Nick Collins’ PPIG paper on "the programming language as a musical instrument" (Blackwell & Collins 2005) was one of the earliest academic publications on the topic, while Alex McLean, developer of the TidalCycles language, presented much of his development research at PPIG (2010, 2011). Other popular live coding languages such as Sam Aaron’s Sonic Pi (Aaron et al 2016) and Thor Magnusson’s ixi lang (Magnusson 2011) have built on engagement with the psychology of programming, and PPIG 2012 hosted a panel in which several of these early leaders gave improvised performances, one of the first times that the community had engaged directly with computing research. Although live coding has always been an artist-led practice, the psychology of programming community provided academic support leading to a subsequent Dagstuhl symposium on the topic (Blackwell et al 2014), and then the establishment of the ICLC series.

Understanding programming as an improvised and creative experience offers new insights to other practices of programming, and alternatives to conventional engineering processes. In a parallel research agenda, the end-user software engineering community explores the way that other kinds of user interaction also involve programming-like activity. The attention investment model applied insights from psychology of programming to observe that not all user interfaces correspond to the mainstream model of direct manipulation, where actions have effects that are immediately observable and reversible. When we consider simple programmable devices such as heating controls, the cognitive challenge of carrying out actions now for effects at a future time can clearly be understood in relation to more complex programs, since they involve abstract tasks such as specification of requirements, reading and writing some kind of notation, and debugging mismatches between these. Such abstract activities require greater investment of user attention than direct manipulation tasks, and applying insights from the psychology of programming offers new insight for many kinds of domestic automation, internet of things scripting, and efficient use of web browser extensions, mashup tools, email agents and so on.

Over the 50 years considered in this historical reflection, increasingly ubiquitous computing hardware has transformed the cultural significance of the computer, and also of programming. Initially associated with speculative scientific research and advanced military technology, by the foundation of IJMMS in 1969 the computer had become a mainstream tool of engineering and business professionals. In contrast to the idiosyncrasies of laboratory apparatus, increasingly professional computing required that programming become more efficient and predictably managed. The human factors and organisational concerns of first- and second-wave HCI offered business benefits and addressed educational policies that focused on professional skill and productivity.

However recent years have seen a further shift in the cultural significance of computers, which is starting to lead to a transformation in understanding the human experience of code. Although professional software development and skills are still important business and technical capacities, code has also become a cultural object, accessible to everyday experience, open to play and creative subversion. All children in wealthy countries are now routinely exposed to code, through playful environments such as Scratch, Alice or Sonic Pi. Multimedia artists routinely hack code with Arduinos and Raspberry Pis, or write code on-stage at hackathons and algoraves. These new and informal styles of programming are enabled by interactive and domain specific languages, and by the tools and development environments that also enable agile software development and iterative design methods.

These new informal, experimental and everyday practices of coding treat code as a kind of craft material, and discuss programming skill and experience in relation to other craft practices in which malleable materials are explored toward creative ends (Bergstrom & Blackwell 2016). The 29th annual meeting of PPIG in 2018 was convened together with the Art Worker's Guild of London, a respected establishment of the 19th century English Arts and Crafts movement. Discussions between these communities highlighted the ways in which programming, like other forms of craft knowledge, is both embodied and socially situated, further broadening interpretation of the word "psychology" to extend well beyond the cognitive and computational concerns of human factors engineers. As with previous waves of influence from psychology of programming to mainstream software engineering and programming language research, we may find that computer science itself becomes more oriented toward acknowledgement of the craft perspective (Blackwell 2018).

7.2 Software design and modelling

As part of the 'third wave', studies of program design, and of the dialogue between problem analysis and solution design, have arisen alongside the many studies of code generation, comprehension, debugging, and maintenance. For example, Curtis, Krasner and Iscoe (1988) conducted seminal field studies at MCC on software design for large systems. Another example is Sonnentag's (1998) analysis of professional software design processes. In IJHCS/IJMMS, there has been a smattering of papers addressing system design (e.g., Guindon, 1990), expert reasoning about program design (e.g., Petre & Blackwell 1999) , and the relationship between domain analysis and solution analysis and construction (e.g., Maiden & Hare, 1998). Yet, although the PP community has been considering expertise, problem-solving, and mental representations throughout this fifty years, it has not yet brought software design and *design thinking* – the conceptual backdrop to 'programming' - fully into the spotlight. Much of the theorizing about plans, for example, was largely addressing routine design, solving familiar problems. 'Design thinking', solving 'wicked' problems, and reasoning more broadly about software systems and systems-of-systems has not received the sort of attention that has been devoted to, say, program comprehension. Similarly, despite the attention to programming languages and language paradigms, there has been less attention to modelling and model-driven development. The community has not yet fully engaged with design thinking, and has not yet expressed a strong cognitive perspective on design and modelling.

7.3 Engagement with programming language/software engineering development

The nature of the communities that study the psychology of programming offer an overlap between those who study non-traditional domains of programming such as live coding for music production, and those who work in commercial and professional development contexts.

This overlap has resulted in an easier interchange of ideas between the professional and end user domains than would otherwise have been possible. A notable example is Clarke's (2001) application of Cognitive Dimensions of Notations to the design of languages (Clarke 2006) and APIs (Stylos et al. 2001) at Microsoft. However the flow of ideas is by no means one-directional; Clarke's reflections on the needs of industry at a panel discussion at PPIG in 2016 contributed to a resurgent discussion about the methods that are appropriate for studying and making claims about the nature of programming interaction.

In a manner reminiscent of the early critique of the empirical study of programming by (Sheil, 1981) and (Brooks, 1980), there has been a call for an increase in the objectivity of the methods used to study the design of programming languages, with a focus on the use of randomised control trials. This call is often accompanied by specific criticism of the methodologies and individuals in the psychology of programming community (e.g. Stefik and Hanenberg, 2017).

Perhaps from the long institutional memory of the community of psychology of programming, as the place where the application of many of the experimental techniques to programming was first tried, there is scepticism (Lewis, 2017) about the appropriateness of underpinning a complex design process with this naive empiricism.

This periodic reminder, that as Clarke put it, the study of the psychology of programming provides 'tools for thinking with', rather than a search for a universal language has been an enduring contribution to both the academic field and its commercial applications.

7.4 PP and Artificial Intelligence

At the time we write, a sentiment often expressed in public promotion of artificial intelligence[1] is that the revolution promised by AI will result in the disappearance of programming - that we will no long program machines, but will "teach" them, perhaps even as we teach children. In this case, research investment for future computing might be better to prioritise attention to AI, rather than investing further in programming, if it will become obsolete in the AI future. How appropriate does such a research strategy appear, in the light of the research that we have surveyed?

One consideration is that the learning algorithms themselves will still need to be programmed by somebody, as will the infrastructure for data transfer, interfacing, communications, mechanical actuation and so on. The notations that are used to express the architecture and processing logic of machine learning infrastructure have often been established with little concern for broader human-centric issues, since the focus has been on mathematical expressiveness and efficiency. What is the notation in which the interfaces and infrastructure of AI should be programmed, and is there any danger that assumptions about AI research might disadvantage or exclude some communities as a result?

However a more fundamental question is whether "teaching" (as performed by labelling training data sets) really will be different from "programming". This returns to the definitional issue "what is programming" introduced earlier in this paper. Will the person who is labelling training data need to consider the future effect of the labels that they are assigning? Many current problems of bias and explanation in AI systems appear to result from a disconnect between labelling and execution.

The classical term in psychology of programming and end-user programming research for specifying system behaviour through use of training examples has been "programming by example". At the current time, AI researchers use the phrases "supervised learning" (programming is done by labelling a set of training data) "semi-supervised learning" (the programmer works by explicitly selecting cases from a larger training data set) or "active learning" (programming is done more interactively, with the system consulting the programmer to act as an oracle who can decide hard cases). In the programming language literature, the compiler-side interpretation of training examples is associated with probabilistic "program

[1] e.g., Chris Bishop, Director of Microsoft Research Cambridge, lecturing on AI to a 200-anniversary meeting dedicated to the "Futures of the Sciences" on 22 March 2019, or Andrew Blake, Director of Samsung AI Research, in the Darwin College Lecture Series on 1 February 2019

synthesis”. From a PP perspective, the people acting as oracles or labellers are still programming (contributing to definitions of system behaviour), simply using different tools and notations.

This is not the first time that AI methods have promised to revolutionise programming. Before the current boom in statistical machine learning methods, previous AI booms were associated with enthusiasm for declarative languages - knowledge representation languages, logic languages, or specification languages. At that time, it was considered that programs would be automatically generated by processing those notations. In fact, the idea that computers might be able to program themselves based on more convenient human-readable specifications dates back even before AI, when early FORTRAN compilers (FORmula TRANslators) were created to “automatically” construct programs that had been expressed directly as mathematical formulae. Unfortunately, with FORTRAN, as with every formal language since, the formal language always turns out to be harder to write than was anticipated, meaning that the human activity of writing the specification still seems like programming. Unfortunately, as semiotician Umberto Eco has demonstrated persuasively, human history has been associated with the "search for the perfect language” since before classical times (Eco 1995). Eco traces a direct line from biblical and medieval scholars to the knowledge representation languages of the AI era, showing that the desire to resolve human problems through better language is a desire both perennial and doomed to failure.

All of the methods above recapitulate human factors problems of earlier specification languages that they require perfect languages (and perfect humans) in order to achieve perfect specifications, whether this is done by labelling examples or by declarative specification. Sadly, the perfection of humanity that might lead to the obsolescence of programming does not appear to be imminent, despite successive generations of AI research. Problems of bias, explanation and fragility of machine learning systems that are based on human data labels are already becoming apparent. Humans training such systems need to understand how the models are working and why they are making the judgements they do - activities that could benefit from the application of prior research in program visualisation and comprehension. Similarly, the construction of the architectures is often defined by notations, both textual and visual, that could benefit from the application of the principles established within the PP communities.

The lesson from all these generations of programming language and compiler technologies has been consistent: although we can compile program specifications from one form to another, the relevant human factors relate to the notation that the human must use to provide the specification. In some cases (logic languages, specification languages), it turns out that the notation is more specialised and harder to use than a conventional program would have been, meaning that the dream of automated “AI” program synthesis is of practical use only to specialised communities for whom that notation is particularly convenient for one reason or another.

However it is not only with respect to usability that there are productive lessons. The attention investment approach focussed not only on the direct usability properties of a notation, but also the judgements that users made about about the risks and returns of programming. This characterisation of the actual and perceived risk has many implications for the design of AI systems, often overlooked. It is not sufficient to have a system that does the right thing, it must also be clear to the end-user programmer that it is *going* to do the right thing, and there must be ways of managing the possibilities of misbehaviour. Otherwise, satisficing users will opt to ignore the possibilities of automation, be it expressed by example, or by imperative description, and to perform the tasks manually.

Although not yet a major focus, the methodological diversity of the PP community allows perspectives to be established that would not otherwise have emerged in either AI or programming language venues. For example, an overtly political critique of the commercial context of software development might appear outside the scope of human factors or cognitive ergonomics of programming. However, it is very much a part of the social context of AI - and offers a way of critiquing the socio-technical implications of future programming systems that integrate AI methods into software development. This is both directly useful to the growing critique of AI in society, but also perhaps acts as a template for how a human centered study of AI (maybe AI/HCC) might be established.

Our recommendation is that the whole of the current boom in AI research, including intelligent user interfaces, mixed initiative systems, autonomous vehicles, ethical AI, explainability and data transparency, and every other domain in which AI researchers have recognised the need for human engagement, would benefit further from insights in the psychology of programming. Notations are everywhere. Direct manipulation is easy but slow. If we desire automation (we do), and if we want to control it (we do), then we will need programming, and the psychology of programming, whatever happens in AI.

8 Conclusion

Over the past 50 years, psychology of programming research has resulted in contributions to a wide range of fields, both by introducing new theoretical focuses, and by acting as an advocate of human-centric priorities within programming language research. In addition to the particular research outcomes that might be expected from the name of the field, this reflective community is also oriented toward meta-level contributions, resulting from a distinctive social character that encourages diversity rather than convergence. As a consequence, psychology of programming has generated a surprisingly broad range of perspectives regarding human engagement with the activity of programming. Few of the resulting theories have been definitive, and on the whole, more ideas have been added rather than existing ones overturned. Many questions have not been conclusively resolved, but this seems to be the point of the discipline. Indeed, programming itself has become so diverse, that any attempt to hold to the priorities of 50 years ago would have resulted in a set of concerns that may have been well-focused, but almost certainly no longer relevant. Instead, we find that many areas of computer use in business and in everyday life now share important characteristics of programming, where the theories, research methods and strategies of psychology of programming are a valuable supplement to conventional questions of usability.

9 References

Aaron, S., Blackwell, A.F. and Burnard, P. (2016). The development of Sonic Pi and its use in educational partnerships: co-creating pedagogies for learning computer programming. Journal of Music, Technology and Education 9(1), 75-94.

Baddeley, A. D., and Hitch, G. (1974). Working memory. In *Psychology of Learning and Motivation*, 8, pp. 47-89. Academic Press.

Beckwith, L., Kissinger, C., Burnett, M., Wiedenbeck, S. , Lawrance, J., Blackwell, A., and Cook, C. (2006). Tinkering and gender in end-user programmers' debugging. ACM Conference on Human Factors in Computing Systems, pp. 231-240.

Bellamy, R.K.E., and Carroll, J.M. (1992) Re-structuring the programmer's task. *International Journal of Man-Machine Studies*, 37(4), pp. 503-527.

Bellucci, A., Vianello, A., Florack, Y., Micallef, L. and Jacucci, G. (2019). Augmenting objects at home through programmable sensor tokens: A design journey. *International Journal of Human-Computer Studies* 122, 211-231.

Bergstrom, I., and Blackwell, A.F. (2016). The Practices of Programming. In *Proceedings of IEEE Visual Languages and Human-Centric Computing (VL/HCC)* 2016.

Blackwell, A.F. (1996). Metacognitive theories of visual programming: What do we think we are doing? In Proceedings IEEE Symposium on Visual Languages, pp. 240-246.

Blackwell, A.F. (2002a). What is programming? In Proceedings of PPIG 2002, pp. 204-218.

Blackwell, A.F. (2002b). First steps in programming: A rationale for Attention Investment models. In Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments, pp. 2-10.

Blackwell, A.F. (2006). Ten years of cognitive dimensions in visual languages and computing. *Journal of Visual Languages and Computing*, 17(4), pp. 285-287.

Blackwell, A.F. (2018). A craft practice of programming language research. In Proceedings of PPIG 2018.

Blackwell, A., and Collins, N. (2005). The Programming Language as a Musical Instrument. In Proceedings of PPIG 2005, pp. 120-130.

Blackwell, A.F. and Morrison, C. (2010). A logical mind, not a programming mind: Psychology of a professional end-user. PPIG 2010, pp. 175-184.

Blackwell, A.F., Rode, J.A. and Toye, E.F. (2009). How do we program the home? Gender, attention investment, and the psychology of programming at home. International Journal of Human Computer Studies 67, 324-341.

Blackwell, A.F., McLean, A., Noble, J. and Rohrhuber, J. (2014). Collaboration and learning through live coding. Dagstuhl Reports 3(9), 130-168. Edited in cooperation with Jochen Arne Otto.

Blandford, A.E., Buckingham Shum, S.J., Young, R.M. (1998). Training software engineers in a novel usability evaluation technique. *International Journal of Human Computer Studies*, 49(3), pp. 245-279.

Bødker, S. (2015). Third-wave HCI, 10 years later---participation and sharing. *Interactions*, 22(5), pp. 24-31.

Brooks, F.P. (1975). The Mythical Man-Month. Addison-Wesley.

Brooks, R. (1977). Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies*, 9(6), 737-751.

Brooks, R. E. (1980). Studying programmer behavior experimentally: The problems of proper methodology. Communications of the ACM, 23(4), 207-213.

Bryant, S., Romero, P. and du Boulay, B. (2008). Pair programming and the mysterious role of the navigator. International Journal of Human-Computer Studies 66(7), 519-529

du Boulay, B., O'Shea, T., & Monk, J. (1981). The black box inside the glass box: presenting computing concepts to novices. *International Journal of Man-Machine Studies*, 14(3), pp. 237-249.

Chee, Y.S. (1993). Applying Gentner's theory of analogy to the teaching of computer programming. *International Journal of Man-Machine Studies*, 38(3), pp. 347-368.

Church, L. and Mărășoiu, M. (2016). A fox not a hedgehog: What does PPIG know? In Proceedings of PPIG 2016.

Clarke, S. (2006) Evaluating a new programming language. In Proceedings of PPIG 2006, pp. 131-139.

Clarke, S., Söderberg, E., and Luff, M. (2016) Panel discussion: PPIG in the wild - what should we be studying?

Collins, N., McLean, A., Rohrhuber, J., & Ward, A. (2003). Live coding in laptop performance. *Organised Sound*, 8(3), 321-330.

Curtis, B., Krasner, H., & Iscoe, N. (1988). A field study of the software design process for large teams. Com*munications of the ACM*, 31(11), pp. 1268-1287.

Davies, S.P. (1990). The nature and development of programming plans. *International Journal of Man-Machine Studies*, 32(4), pp. 461-481.

Eco, U. (1995). *The Search for the Perfect Language*. Oxford: Blackwell.

Ernst, G. W., & Newell, A. (1969). *GPS: A case study in generality and problem solving.* Academic Press. ISBN 9780122410505.

Flor, N. V., & Hutchins, E. L. (1991). A case study of team programming during perfective software maintenance. *Empirical Studies of Programmers: Fourth Workshop*, p. 36-56. Intellect Books.

Gentner, D., & Stevens, A. L. (Eds.) (1983). *Mental Models.* Psychology Press.

Gilmore, D. J. Green, T. R. G. (1984). Comprehension and recall of miniature programs. *International Journal of Man–Machine Studies,* 21, 31–48.

Goldstine, H. H. & Von Neumann, J. (1947). Planning and coding of problems for an electronic computing instrument. Report on the Mathematical and Logical aspects of an Electronic Computing Instrument Part II, Vol. 1-3. IAS ECP List of reports, 1946-57 nos. 4,8,11. Institute for Advanced Study, Princeton, New Jersey

Green, T. R. G. (1977). Conditional program statements and their comprehensibility to professional programmers. *Journal of Occupational Psychology*, 50(2), pp. 93-109.

Green, T. R., Petre, M., & Bellamy, R. K. E. (1991). Comprehensibility of visual and textual programs: A test of superlativism against the'match-mismatch'conjecture. *Empirical Studies of Programmers: Fourth Workshop,* pp, 121-146. Intellect Books.

Green, T. R. G., & Petre, M. (1996). Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing*, 7(2), pp. 131-174.

Guindon, R. (1990). Knowledge exploited by experts during software system design. *International Journal of Man-Machine Studies*, 33(3), pp. 279-304.

Hoc, J.M. (1979). Le probleme de la planification dans la construction d'un programme informatique. *Le Travail Humain*, 42(2), pp. 245-260.

Hoc, J. M. (1981). Planning and direction of problem solving in structured programming: An empirical comparison between two methods. *International Journal of Man-Machine Studies*, 15(4), pp. 363-383.

Kankuzi, B. and Sajaniemi, J. (2016). A mental model perspective for tool development and paradigm shift in spreadsheets. International Journal of Human-Computer Studies 86, 149-163.

Ko, A.J., Abraham, R., Beckwith, L., Blackwell, A.F., Burnett, M., Erwig, M., Lawrence, J., Lieberman, H., Myers, B., Rosson, M.-B., Rothermel, G., Scaffidi, C., Shaw, M., and Wiedenbeck, S. (2011). The state of the art in end-user software engineering. *ACM Computing Surveys*, 43(3), Article 21.

Kostakos, V. (2015). The big hole in HCI research. *Interactions*, 22(2), pp. 48-51.

Letovsky, Stanley, Pinto, Jeannine, Lampert, Robin, Soloway, Elliot (1987). A cognitive analysis of a code inspection. In: Olson, Gary M., Sheppard, Sylvia B., Soloway, Elliot (eds.) *Empirical Studies of Programmers: Second Workshop*, pp. 231-247. Ablex Publishing Corp.

Leventhal, L.M. (1988). Experience of programming beauty: some patterns of programming aesthetics. *International Journal of Man-Machine Studies*, 28(5), pp. 525-550.

Lewis, C., & Olson, G. (1987). Can principles of cognition lower the barriers to programming?. *Empirical Studies of Programmers: Second Workshop*, pp. 248-263. Ablex Publishing Corp.

Lewis, C. (2017) Methods in user oriented design of programming languages. In Proc. PPIG 2017 Psychology of Programming Annual Conference, Delft, Netherlands, 1-3 July 2017.

Lui, K.M. and Chan, K.C.C. (2006). Pair programming productivity: Novice–novice vs. expert–expert, International Journal of Human-Computer Studies 64(9), 915-925

Magnusson, T. (2011). Algorithms as scores: Coding live music. *Leonardo Music Journal,* 21, 19-23.

Maiden, N.A.M., and Hare, M. (1998). Problem domain categories in requirements engineering. *International Journal of Human Computer Studies*, 49(3), pp. 281-304.

von Mayrhauser, A., & Vans, A. M. (1997). Program understanding behavior during debugging of large scale software. *Empirical Studies of Programmers: Seventh Workshop*, pp. 157-179. ACM.

Miller, L.A. (1974). Programming by non-programmers. *International Journal of Man-Machine Studies*, 6(2), pp. 237-260.

Moher, T. G., Mak, D., Blumenthal, B., & Leventhal, L. (1993). Comparing the comprehensibility of textual and graphical programs. *Empirical Studies of Programmers: Fifth Workshop*, pp. 137-161. Ablex, Norwood, NJ.

Pandey, R. K. and Burnett, M.M. (1993). Is it easier to write matrix manipulation programs visually or textually? An empirical study. In *Proceedings IEEE Symposium on Visual Languages*, pp. 344-351.

Pennington, N. (1987). Comprehension strategies in programming. Empirical Studies of Programmers: Second Workshop, pp. 100-113. Ablex Publishing Corp.

Petre, M., and Blackwell, A.F. (1999). Mental imagery in program design and visual programming. *International Journal of Human Computer Studies*, 51(1), pp. 7-30.

Rist, R. S. (1986). Plans in programming: definition, demonstration, and development. *Empirical Studies of Programmers* (pp. 28-47). Ablex, Norwood, NJ.

Sharp, H. and Robinson, H. (2008). Collaboration and co-ordination in mature eXtreme programming teams. International Journal of Human-Computer Studies 66(7), 506-518,

Sheil, B. A. (1981). The psychological study of programming. *ACM Computing Surveys (CSUR)*, *13*(1), 101-120.

Shneiderman, B. (1983). Direct manipulation: A step beyond programming languages. *Computer*, 8, pp. 57-69.

Sime, M. E., Arblaster, A. T., & Green, T. R. G. (1977). Reducing programming errors in nested conditionals by prescribing a writing procedure. *International Journal of Man-Machine Studies*, 9(1), pp. 119-126.

Sime, M. E., Green, T. R. G., & Guest, D. J. (1973). Psychological evaluation of two conditional constructions used in computer languages. *International Journal of Man-Machine Studies*, 5(1), pp. 105-113.

Simon, H. A. (1969). *The Sciences of the Artificial*. Cambridge, MA: MIT Press.

Stefik, A. and Hanenberg, S.. (2017), "Methodological Irregularities in Programming-Language Research," in *Computer*, vol. 50, no. 8, pp. 60-63.

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, (5), pp. 595-609.

Sonnentag, S. (1998). Expertise in professional software design: A process study. *Journal of Applied Psychology,* 83 (5), pp. 703-715.

Sonnentag, S., Niesson, C., & Volmer, J. (2006). Expertise in software design. In: *Cambridge Handbook of Expertise and Expert Performance.* Cambridge: Cambridge University Press, pp. 373-387.

Spohrer, J. G., & Soloway, E. (1986). Analyzing the high frequency bugs in novice programs. Empirical Studies of Programmers: First Workshop, pp. 230-251. Ablex Publishing Corp.

Stylos, J., Clarke, S., and Myers, B. (2001) Evaluating a new programming language. In Proceedings of PPIG 2001, paper 22

Sutherland, I.E. (1963/2003). Sketchpad, A Man-Machine Graphical Communication System. PhD Thesis at Massachusetts Institute of Technology, online version and editors' introduction by A.F. Blackwell & K. Rodden. Technical Report 574. Cambridge University Computer Laboratory

Weinberg, G. M. (1971). *The Psychology of Computer Programming.* New York: Van Nostrand Reinhold.

Weinberg, G. M., & Schulman, E. L. (1974). Goals and performance in computer programming. *Human Factors*, 16(1), pp. 70-77.

Wiedenbeck, S. (1986). Beacons in computer program comprehension. *International Journal of Man-Machine Studies*, 25(6), pp. 697-709.

10 Appendix 1

psychology of programming	end-user development	software engineering	programming languages	CSCW	CS Education	HCI
PPIG – Psychology of Programming Interest Group, **ESP** – Empirical Studies of Programmers Workshop, **VL-HCC** – IEEE Symposium on Visual Languages and Human-Centered Computing	**IS-EUD** - International Symposium on End-User Development, **EuSPRIG** - European Spreadsheet Risk Interest Group Conference, **(EUSES Consortium** - end-user software engineering technologies for enabling End Users to Shape Effective Software)	**TSE** – IEEE Transactions on Software Engineering, **TOSEM** – ACM Transactions on Software Engineering and Methodology, **JSS** – Journal of Systems and Software, **ESE** – Empirical Software Engineering, **SSM** – Software and Systems Modelling, **ICSE** – IEEE International Conference on Software Engineering, **FSE** – Foundations of Software Engineering, **EASE** – Evaluation and Assessment in Software Engineering conference, **Models** – ACM International Conference on Model Driven Engineering Languages and Systems, **CHASE** – IEEE/ACM International Workshop on Cooperative and Human Aspects of Software Engineering (at ICSE), **IEEE Software**,	**SIGPLan** – ACM special interest group on programming languages, **SPLASH** - ACM SIGPLAN conference on Systems, Programming, Languages and Applications, including **OOPSLA** (Object-oriented Programming, Systems, Languages, and Applications), **Onward!**, **DLS** (the Dynamic Languages Symposium) **PLoP** - Pattern Languages of Programming (at SPLASH) **PLATEAU** - Evaluation and Usability of Programming Languages and Tools (at SPLASH), **PX** - Programming Experience Workshop (at <Programming>) **SdR** - Salon des Refuse (at <Programming>) **JCL/JVLC** - Journal of Computer Languages,	**CSCW journal** – Computer Supported Cooperative Work: the Journal of Collaborative Computing and Work Practices **CSCW conference** - ACM Conference on Computer-Supported Cooperative Work and Social Computing **IWSC** - International Workshop on Social Computing	**SIGCSE** – ACM Special Interest Group on Computer Science Education Symposium, **CSE** -Computer Science Education, **ACM Inroads** **ToCE** - ACM Transactions on Computing Education (TOCE) (formerly JERIC - ACM Journal on Educational Resources in Computing), **FiE** – IEEE Frontiers in Education, **ITiCSE** – ACM Conference on Innovation and Technology in Computer Science Education, **ICER** - ACM International Computing Education Research Conference, **CompEd** - SIGCSE Global Computing Education Conference, **ACE** - Australasian Computing Education,	**CHI** – ACM Conference on Human Factors in Computing Systems, **IwC** – Interacting with Computers, **ToCHI** – ACM Transactions on Computer-Human Interaction, **Human Factors**, Interactions, **BIT** – Behavior and Information Technology, **UbiComp** – ACM Conference on Pervasive and Ubiquitous Computing, **DIS** – ACM conference on Designing Interactive Systems, **UIST** - User Interface Software and Technology. **Interact** British HCI Conference, **IUI** – Intelligent User Interfaces, **TEI** - ACM International Conference on Tangible, Embedded and Embodied

		Journal of Software Maintenance and Evolution, **ICPC** (formerly IWPC, WPC) - IEEE International Conference on Program Comprehension, **ICSME** (formerly CSM) International Conference on Software Maintenance and Evolution, **Agile** conference, **XP –** Agile Alliance International Conference on Agile Software and Systems Development	incorporating the Journal of Visual Languages and Computing			Interaction