Innovative Methods, User-Friendly Tools, Coding, and Design Approaches in People-Oriented Programming

Steve Goschnick
Swinburne University of Technology, Australia

A volume in the Advances in Computer and Electrical Engineering (ACEE) Book Series

Published in the United States of America by
IGI Global
Engineering Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue
Hershey PA, USA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

Library of Congress Cataloging-in-Publication Data

Names: Goschnick, Steve, 1956- editor.
Title: Innovative methods, user-friendly tools, coding, and design approaches in people-oriented programming / Steve Goschnick, editor.
Description: Hershey, PA : Engineering Science Reference, an imprint of IGI Global, [2018] | Includes bibliographical references and index.
Identifiers: LCCN 2017059192| ISBN 9781522559696 (hardcover) | ISBN 9781522559702 (ebook)
Subjects: LCSH: Human-computer interaction. | User interfaces (Computer systems) | Electronic data processing.
Classification: LCC QA76.9.H85 .I5765 2018 | DDC 004.01/9--dc23 LC record available at https://lccn.loc.gov/2017059192

This book is published in the IGI Global book series Advances in Computer and Electrical Engineering (ACEE) (ISSN: 2327-039X; eISSN: 2327-0403)

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Advances in Computer and Electrical Engineering (ACEE) Book Series

ISSN:2327-039X
EISSN:2327-0403

Editor-in-Chief: Srikanta Patnaik, SOA University, India

MISSION

The fields of computer engineering and electrical engineering encompass a broad range of interdisciplinary topics allowing for expansive research developments across multiple fields. Research in these areas continues to develop and become increasingly important as computer and electrical systems have become an integral part of everyday life.

The **Advances in Computer and Electrical Engineering (ACEE) Book Series** aims to publish research on diverse topics pertaining to computer engineering and electrical engineering. **ACEE** encourages scholarly discourse on the latest applications, tools, and methodologies being implemented in the field for the design and development of computer and electrical systems.

COVERAGE

- Programming
- Computer Architecture
- Analog Electronics
- Microprocessor Design
- Sensor Technologies
- Electrical Power Conversion
- Digital Electronics
- Computer Hardware
- Optical Electronics
- Computer science

Table of Contents

Detailed Table of Contents

Foreword

I welcome the opportunity to introduce you to this volume, a representation of the varied contributions that have appeared in the *International Journal of People Oriented Programming* (IJPOP), a journal devoted to ideas concerning the generation of user-fashioned software products. Steve Goschnick, the editor of this eclectic collection of articles, co-taught with me the inaugural *Interactive System Design* subject offered in the late 1990s to 'hard core' computer scientists at the University of Melbourne. That subject and its various descendants benefited greatly from Steve's extensive ICT industry experience and his enthusiasm as an educator, and that same spirit is evident in the choice of the articles comprising this volume.

Steve has a longstanding passion for making coding and the associated activities more accessible and this volume includes many articles illustrating the remarkable progress that has been made internationally in bringing such ambitions to reality. As viewed by designers of programming environments such as described in Section 1, coding is a type of literacy. By this it is meant that practicing the discipline of coding is not just about writing programs, but can help in organising one's thinking more generally and so improve one's ability to express ideas. But the articles in Section 1 are not simply a contemporary view of novice programming languages and environments – they are written by leaders in the field who have generously shared their expert reflections and lessons learned.

Indeed, following the spirit of POP, the authors of this collection are not just spectators or commentators, but are themselves do-ers, drawn from varied disciplinary backgrounds. The four sections of the volume contain a broad sweep of styles and cover a range of topics relevant to the growing field of supporting People Oriented Programming: the art of programming language design for novice coders; the craft of requirements gathering in support of software engineering practice relevant to the design of individualisable socio-technical systems; an interesting excursion through a cultural studies perspective on identity; and material concerning the design of personalised learning environments.

A reader can use this book as a starting point from which to plunge into specific topics, or just to get a taste of diverse perspectives on what it will take to further progress the field of People Oriented Programming.

Liz Sonenberg
The University of Melbourne, Australia
January 2018

Liz Sonenberg *is a Pro Vice-Chancellor at The University of Melbourne, Australia, and also Professor of Information Systems in the Melbourne School of Engineering. In her Pro Vice-Chancellor role, Professor Sonenberg holds specific responsibilities for planning of university-wide research infrastructure, and for oversight of the University's digital and data strategies. Liz Sonenberg's research expertise is in computational science, with the integrating theme of her research being the conceptualisation and construction of adaptive, distributed, intelligent information systems.*

Preface

People from all walks of life use modern interactive systems in a multitude of ways across their lives, and the rate at which such devices and apps are created with evolved features and functionality, is placing new and urgent challenges upon the creators of these same devices, systems and apps. How do you anticipate what people need or want? How do you design them for ways of operation yet to be discovered or envisaged? How do you improve peoples' experience in each of their unique situations? These are among the most pressing design issues of our times. Furthermore, there are complexities in contemporary life that exacerbate these problems for designers and creators, including:

- A heightened need for flexibility.
- Constant consideration of the context of the user.
- Dynamic environments that include heightened degrees of uncertainty.
- Incomplete information about the tasks or activity at hand.
- A growing diversity in the users themselves and their needs as the uptake of technology continues into the far cultural reaches of the planet.
- A proliferation of sensors upon the user and within their environment, both near and far (e.g. satellite).
- Immersion of the user within multiple parallel social worlds including personal, family, recreational and work-related social groupings, not to mention civic engagements at the local, national and global levels.

The People-Oriented Programming paradigm is envisaged to address this multifaceted design and development problem. It has a particular strength over conventional paradigms within domestic and non-work settings, where people spend a significant amount of their time, often on non-work-related tasks or goals, and less results-orientated activities such as play and contemplation. However, it is an equally adept paradigm when supporting or enhancing goal-oriented activities. Under the descriptive umbrella of People-Oriented Programming, there are a diverse range of new and innovative methods, new requirements gathering and design tools, user-friendly coding environments and new programming languages aimed at introducing programming to everyone that wishes to write their own code, often with online social communities of people that help each other.

BACKGROUND TO THE POP PARADIGM

People-Oriented Programming (POP) is a relatively new design paradigm for developing individual-oriented, software-based applications and systems, with a rudimentary sketch of what the paradigm should encompass, outlined in (Goschnick, 2009). It has four *elements* of activity, the first three of which call upon the individual user themselves:

1. An individual user of the system or application is the central focus of the endeavour, with customisations directed specifically to fit their requirements, in what von Hippel and Katz (2002) described as a 'market of one' – emphasising that the user needs to be heavily involved in servicing the customisation. Conversely, this increasingly empowered individual, although a member of multiple communities both online and the more traditional, is progressively becoming a global citizen.
2. Highly usable and well-engineered development tools are employed by the end-user so that the individual in question can design and then code, configure, make or mashup solutions to their own problems and situations, either afresh or customised from solutions to similar problems or needs, usually drawing upon a community of like-minded people.
3. Self-ethnography (Arnold 2004; Goschnick & Graham, 2005) is employed to some degree including the use of sensors, to define and refine the individual's requirements and clarify the context. This is being enhanced with ongoing improvements and innovations in new methods for requirements gathering, analysis and design of applications for individuals and around an individual's situatedness.
4. Cognitive models are drawn from two disciplines: the Agent-Oriented (AO) paradigm, and Cognitive Task Modelling (from HCI) – both for the analysis and design of applications, and also in creating new end-user development tools and methods, that make the second and third elements immediately above increasingly viable to all who want to participate in POP.

Although the third element initially represented a big hurdle for POP to be viable, Goschnick (ibid) drew attention to the practice of modding by users of games in the likes of World of Warcraft, as a demonstration that tools of the required usability and effectiveness were possible, and indeed, were becoming more commonly available. That same year Minecraft - the popular 3D creative game - was released on the public with very fast uptake by 5-year-olds and up, accruing tens of millions of users in just a few years (currently it has in excess of 100 million paid downloads). Such a rapid uptake for a paid-for app is uncommon, attesting to the demand for user-friendly software that end-users of virtually all ages can use by themselves to create things that previously required (in that case) game developer level skills. The impact that Minecraft was having on young creativity was further underlined by the willingness of Microsoft Inc. to pay US$2.5 billion for the controlling company, Mojang AB, in 2014.

COMPARATIVE TIMELINES OF THE OBJECT-ORIENTED AND AGENT-ORIENTED PARADIGMS

To get something of a handle on what the future holds for the development and consolidation of POP, it is worth looking at parallels in other much older paradigms of software and system development. In the early years of the millennium the Agent-Oriented paradigm of software development came into focus. It happened after nearly a decade of research and development, coming from different players across the world, and it didn't come in an orderly or pre-defined manner. A new software design and development paradigm doesn't come into existence in a neat chronological sequence like: Theory, Tools, Methodology, Framework, Applications. Some AO researchers developed tools first (e.g. dMARS) which they eventually realized were AO paradigm tools, as others started with theory first and only then developed AO tools. While others began with a pressing application first (e.g. the intelligent software in a robot; the software for airport flight control systems). The results in one area of AO research were used as feedback to improve the other areas which were not a specific focus - e.g. Work on analysis and design, provided feedback to agent theory and tools, and fed-forward into improved implementation efforts in problem domain areas (applications), and both informal methods and more rigorous full-blown methodologies. So, there were early AO architectures that now have supporting methodologies, early tools that have now grown into sophisticated frameworks, and early AO applications that guided theory, tools and methodologies. Not so much chaotic as vigorous innovation by pockets of researchers spread around the world, each informed periodically by the developments of the others, each realizing at different times that they were indeed toiling away on one or a few aspects of what was a whole new paradigm of software design and development.

The same was true of the Object-Oriented Paradigm (OOP). For example, OOP languages were widely available in the 1980s, including C++, Objective C and

Smalltalk-80 (itself based on Simula-67, circa 1967); and yet the Unified Modelling Language (UML) used predominently in the analysis and design of OOP programs and systems, only culminated in a 1997 release of UML V1 of that agreed upon standard, bringing together the proponents of three earlier competing methodologies which themselves evolved separately in the early 1990s.

TIMELINE OF THE PEOPLE-ORIENTED PARADIGM

This same non-chronological manifestation of a paradigm, over a significant number of years, can be seen with the People-Oriented Programming paradigm. Researchers and developers are working on different aspects of POP, spread around the world, often without realizing they are putting in place, the pieces that will eventually reach some agreed upon paradigm, an enterprise much larger than the corner they are working on – which will eventually be called POP, or some other name with a very similar intent and coverage of human endeavour.

This book is a contribution to that realisation and the ongoing discussion. Each of the chapters makes a solid and unique contribution to this discussion. As what happened with the OOP paradigm the POP languages, such as Scratch and Greenfoot, are well ahead of the other aspects (methodologies, theories, analysis and design notations) of the paradigm. In this sense, the evolution of the POP paradigm is more like that of the OOP paradigm, than the AO paradigm. The AO paradigm had great trouble coming up with AO programming language – most of the AO frameworks and toolkits were based on Object-Oriented languages, e.g. dMars was based on C++, while other AO programming environments like JACK and JADE were heavily Java-oriented, as the underlying language. Whereas in the OOP paradigm, the languages came first.

Nonetheless, there are significant contributions to the methods, theories and analysis and design tools of POP, right here in this volume. And as with the AO paradigm which took decades for any firm consensus of what constitutes AO, the POP paradigm is in its earliest days and a widely agreed definition of it, is probably far off in the future. This book, drawing as it does from the most pertinent papers published in the International Journal of People-Oriented Programming (IJPOP), updated with new research and insights, taking into consideration the latest developments in their respective topics and areas of expertise, makes a significant contribution to the embryonic but impactful POP paradigm.

SECTION 1: PROGRAMMING ENVIRONMENTS FOR PEOPLE-ORIENTED PROGRAMMING

Highly usable and well-engineered development tools are required for People-Oriented Programming so that the end-user can design, code, configure, make or mashup solutions to their own problems, either afresh or customised from solutions to similar problems or needs. This section is made up of four chapters that together present the state-of-the-art of the languages of the People-Oriented Programming paradigm. There has been a push to bring computer programming languages to a much wider cohort of people, for several decades, with Basic and Logo signposting very early attempts. However, there has been something of a breakthrough with the block-based languages from about 2007, onward. New programming environments such as Scratch (Chapter 1) and Greenfoot (Chapter 2) on the web, Alice for 3D coding, and ScratchJr (Scratch Junior) on tablets, allow the effective teaching of coding concepts and the writing of code, to children as young as 5 or 6 Note: ScratchJr is realistically aimed at 5 to 7 year olds with an iPad or Android tablet.

These new language environments have coincided with the introduction of coding into the school curriculum in numerous countries, including the UK, Australia, and parts of the US. E.g. Judith Good (Chapter 1) notes that the English national curriculum introduced statutory computing programmes of study for all educational key stages, from age 5 to 16 in 2013. Estonia introduced coding into primary schools in 2012. How these young coders transition to the current traditional textural languages of the software engineering and computing world as undergraduates and beyond, is one subject of Michael Kölling's excellent work in Chapter 2. The new language environments have also coincided with the rise of the Maker movement, which most often brings hardware and software together in creating brand new artifacts though personal invention. New inexpensive single board computers (e.g. Raspberry Pi; many variants of Arduino) have arrived, that take the student and the hobbyist from coding to the hardware and sensor world, and back again. In Chapter 3 Antonio Rizzo and colleagues present their work on UAPPI, a platform that extends the App Inventor block-based language for Android, together with hardware that combines the processing functionality of Arduino and Raspberry Pi on a single board, into an IoT (Internet of Things) platform. They also use a Machine Learning toolkit from Google to do some surprisingly sophisticated things in their student-based case studies, for example "*a light source is controlled by the blink of the eyes and the (intensity of a) smile*". Chapter 4 brings the focus away from the new block-based languages and back upon natural language for computational purposes, succinctly titled *From Natural Language to Programming Language*.

Chapter 1

Professor Judith Good re-examines five user-friendly coding environments for novice programmers – Scratch, Alice, Looking Glass, Greenfoot and Flip – a full 6 years after she first published a state-of-the-art investigation of the same five environments in the International Journal of People-Oriented Programming in 2011. All have evolved substantially in that time, in a number of different ways. She identifies a number of exciting trends, including the substantial advances in these programming environments for novices, with regard to the programming languages inherent in them, the sophistication of the user interfaces, the growth of the communities around them, and a recognition of the importance of the innovative collaborative options that coders themselves are pioneering, in this new type of creative social network (e.g. both Scratch and Greenfoot have online supportive communities of users, that are able to share code and much more). She highlights the significant expansion of resources around these coding environments, for both learners and educators.

Professor Good looks at the process one goes through in creating a program in each of the five environments, and discusses the special features that mark out each from the others. Each coding environment has gone on a specific journey in terms of: how the developers have seen their target user base, what they think are the most important features such an environment should have to support and encourage novice coders in their creative endeavours and progress, and so on. Growth has been exponent in the uptake of these environments, for example, Scratch now has in excess of 20 million projects online.

Alice now has a vast gallery of ready-to-use 3D objects from the SIM II game, made available to coders for their own purposes thanks to Electronic Arts. The Alice approach takes a *game-first* or *animation-first* approach to motivating the novice coder. Typical of many creative tools where the user is a prosumer of new virual artifacts, Alice 3 has two significant editors within the one environment: a Code Editor complete with drag and drop controls, and a Scene Editor, where the user 3D world is envisaged, created and populated with all manner of objects. There is significant support for students to transition to the Java language, for those wishing to move beyond/out-of block-language coding, into a regular IDE, in this case Netbeans.

Looking Glass was designed with the aim addressing the large gender disparity in undergraduate computer science. It is specifically aimed at helping middle school girls (aged 11-14 years) to create stories in a 3D world, based on the rationale they will be gain motivation in tackling the programming part of the exercise. The Looking Glass IDE (Integrated Development Environment) is itself based on Alice 3, and so it also has both a Scene Editor and a Code Editor.

Greenfoot is covered too, but much more so in Chapter 2, by its originator. Flip is a block-based language for constructing game dialogues in a dungeon-and-dragon style role-playing game (an RPG). There is much innovation here, and Flip is an outcome of Good's own applied research. Flip demonstrates much innovation, for example, there is a translation feature from the routines coded in blocks, into natural language, that has proven to be a signifcant communication aide in discussions amongst small teams of coders, such as debugging their logic via the English equivalent.

After a comparison of the 5 environments there is an extensive section of reflection, drawing upon both the chapter's coverage and a significant career down the path of bringing programming and computational thinking to everybody. For example, whether the current textual vs block-based coding discussion is the right level of investigation that researchers should be concentrating upon. Professor Good identifies these new POP tools as ideal environments for studying collaboration. However, she also emphasies that the collaborative e-nature of learning and advancement in environments like Scratch, shouldn't supplant the cognitive – that these tools need to continue to support and encourage external representations of individual cognitive activity and understanding.

Chapter 2

In this chapter titled 'Blue, BlueJ, Greenfoot: Designing Educational Programming Environments', Professor Michael Kölling begins with a brief history of the programming languages and development environments that were most focused on people learning to code, which began almost as early as programming itself. Professor Kölling is a pioneer in providing environments for learning to code. He is in the unique and valuable position of having been centrally involved in designing and developing three programming environments for teaching programming to novices, over a period of more than twenty years. The involvement includes cultivating and servicing input from communities of learners, and teachers, for ongoing improvements to both BlueJ and Greenfoot. BlueJ, introduced in the year 1999, was aimed at 1st year undergraduate students, teaching them 'objects first' in the Java language. Kölling soon realised that many 1st year undergraduate students were learning their first programming language well before beginning their tertiary studies. He then designed Greenfoot to address the needs of students and teachers at mid-level high school.

Kölling outlines that history, often in relation to other environments, notably Scratch and Alice. Significantly, he distils his hard-won advice gained in the design decisions he took with the three environments, and in supporting the large user communities of two of them. The advice is particularly well suited to current or would-be designers of new learning environments for coding. It is equally well suited to those wanting to choose between existing environments in teaching programming/ coding to a particular target user base.

His sub-headings within the section The Lessons Learned, are good to repeat here as the briefest of a summary: Visualization, interaction, simplicity; Know your target group; Do one thing well; Say no (to features requests beyond the target group); Beware feature creep; Do not be afraid to make decisions; Availability; Support material matters (e.g. "*A software system—no matter how well designed—does not, by itself, make an impact in education. What actually drives adoption is an ecosystem of support material*"); The importance of motivation (e.g. "*We want to create motivation, show how a computer works, instill computational thinking, discuss problem solving, show the main aspects and limits of programming. In short, it is about concepts, not syntax*"); Taking control (and the learner taking ownership); Visualization of program execution; The power of community; Teacher communities.

His own endeavours over the extended period of time that has elapsed since the turn of the millennium, have seen the target user group for such environments go from first year university programming novices to high school student novices. Further, he has observed the block-based languages successfully targeting primary school kids as young as seven.

Greenfoot was written to address the high school cohort, whereas BlueJ had been envisaged for first year university students who had already chosen their career direction. In high school, everyone was being subjected to coding, and so *motivation* became the key design criteria.

In moving from a class diagram interface in BlueJ, Greenfoot went for interactive objects: "*we arrived at a well-known destination: micro-worlds. … we had come full-circle… But most importantly, Greenfoot was not (just) a micro-world, but a micro-world meta framework*." That is, any number of micro-worlds, and objects in a world, could be created by users, all within a fully developed coding environment (editor, debugger, runtime system, graphics library, and so on).

There is now a new cohort of students that go from block-based environments for their first foray into coding, then learn a text-based syntax language later on if they need to take programming more seriously. Some of the new tools do both styles of coding in the one environment, demonstrating a two-way approach to coding, sometimes in dual side-by-side windows.

Towards the latter part of the chapter Professor Kölling introduces his own new work on Greenfoot 3, with a technique he calls *frame-based editing* that combines aspects of both block and text code editing in a single merged approach. This frame-based editing approach is brought home with a new language called Stride, although Java can still be used instead. Indeed, the syntax of Stride is something of a subset of that of Java. The new version aims to enable novices who are first introduced to coding in the new environment, to be able to continue on in it, taking their coding as far as they want to go, without the necessity of a transition.

This is a valuable chapter from many perspectives, particularly with regard to those learning, teaching and researching in and around the high school cohort of people learning to code. However, its greatest value is perhaps the set of generic lessons and insights drawn from the experience of envisaging, developing and supporting two very significant and innovative coding environments in BlueJ and GreenFoot. There are tools available for the developers of new block-based languages, principally Blockly, a JavaScript code base from a joint Google and MIT project – the same people who brought App Inventor to Android, discussed in the next chapter. All sorts of block-based environments based on Blockly have appeared in the last year or two, including the Code.org site, which runs the well-subscribed to Hour-of-Code, a school-focused event held each December. If you are thinking of joining the few research teams that currently supply a block-based coding environment, and in so doing, adding to the toolsets available for the People-Oriented Programming paradigm, then this chapter is a must read.

Chapter 3

Titled 'UAPPI: A Platform for Extending App Inventor Towards the Worlds of IoT and Machine Learning', this chapter by Professor Antonio Rizzo and colleagues, is complementary to the previous two chapters, in that it is about another block-based language, namely App Inventor, but an extended version that gives new coders access to the new frontier of the Internet of Things (IoT) via an UDOO board. The UDOO board allows microcontroller-managed sensors and actuators to be accessed from code (note: It has two ARM processors integrated on the one board – one much like the Raspberry Pi single-board computer but which instead runs Android apps, while the other is capable of running Arduino like programs).

Apps have been a great motivator for novices wanting to learn to code, and App Inventor for Android has arguably been the most successful approach to date, in terms of novices programming apps for mobile devices. The authors present an extension to App Inventor called UAPPI (short for UDOO App Inventor), that adds some new functional blocks to the block-based language, to deal with the new hardware – beyond what an Android mobile phone normally has. This has been possible since the original source code of App Inventor is open source. The chapter shows how they use App Inventor and the UDOO hardware to connect it to all sorts of sensors, all from within the widely-used block-based language of the App Inventor environment (i.e. there were over 6 million registered users of App Inventor who had created over 21 million apps by Dec 2017). UDOO App Inventor empowers novices further, by giving them the means for direct experience in coding sensors and firing up actuators in the real world beyond the touchscreen of their everyday mobile devices.

Three case studies are presented using UAPPI. The researchers followed the Research through Design (RtD) approach. RtD in Human-Computer Interaction (HCI) is a practice focused on improving the current state-of-the-art by enabling new behaviours and interactions between technology and humans. Participants are pro-active makers of a world they desire to live in. In the more sophisticated case they used UdooVision components, that integrate the Google Vision framework, which is able to find objects in photos and video, in real-time by use of on-device vision technology. The Google mobile face API finds human faces in photos, tracks positions of *facial landmarks* (eyes, nose, mouth) providing information about the state of those facial features, e.g. Are the subject's eyes open? Are they smiling? This allowed them to control an LED light with a smile, or by simply blinking their eyes. The three application scenarios presenting in this chapter, are examples of interaction with everyday objects and related activities, that were transformed by merging physical objects with digital technology. All from a block-based coding language, highly accessible to novice programmers.

Chapter 4

The title of this chapter is 'From Natural Language to Programming Language' by Associate Professor Dinghao Wu and colleague Xiao Liu. Natural language programming is one of the current research directions being pursued, as an easier way for people with limited programming experience to compose solutions to practical problems, where a programming task is currently required. After analysis of research on the language features used by non-programmers in describing their problem-solving approaches, the authors proposed and built a general framework that synthesizes programming languages from natural language descriptions, with considerable success.

They present three case studies, the first of which aimed at lowering the bar for novice programmers or children, to draw graphs. They prototyped a domain-specific *program synthesis* system called Programming in Eliza (PiE), to draw graphs, taking as input, conversations in English with a computer. PiE is capable of synthesizing programs in the LOGO programming language to draw graphs, taking input from the English conversation between Eliza and the user.

Their second case study, called Natural Shell, is about turning natural language describing a task, into a shell script that does the deed. Shell scripting is widely used for automating tasks at the Operating System level - sometimes tasks involving multiple applications from different sources or vendors. Despite the popularity of shell scripting, it remains difficult to use for both beginners and experts alike, but for different reasons. Beginners are confronted by a barrage of cryptic commands and concepts, while experts struggle with the often incompatible syntaxes across different systems, made worse by seemingly similar syntax, that can be used for some completely different tasks. The authors solution is a system they call *Natural Shell* that transforms the natural language describing a task we want done on our computer, into the appropriate shell scripts that make it happen. It is a multi-step process. Firstly, it translates the natural language of the user into an intermediate, system-agnostic script called Uni-shell. In the remaining steps, Natural Shell transforms the Uni-shell script into the appropriate underlying native script, executes it, then reports details about the resulting action. In an affirming and reinforced-learning manner, the user then gets to see all the steps involved: the natural language they entered, the generated Uni-shell script, the converted native script, together with the system-reported results. Their system was applied to tasks that together require some 50 common shell commands.

The third case study aimed to lower the bar for network administration training and reduce the configuration complexity, so they proposed EasyACL, a tool that synthesizes ACL configuration commands for different platforms directly from natural language descriptions. EasyACL can port the synthesized commands to different platforms, namely, Cisco and Juniper.

Natural Shell and EasyACL both provide an intermediate phase, where the sytem provides a formal interpretation of what the end-user was requesting in natural conversation. The user can thereby quickly see if the system really interpreted what they really meant. In addition to ensuring that the user understands the meaning of tasks undertaken in a programming language, the use of a natural language translation into an intermediate language that is understood by both the user and the system, also reduces the time it takes for the user to learn and do appropriate tasks. Furthermore, the intermediate representation is platform-independent, so the commands can be ported to any platforms with same functionalities, e.g., Bash, Csh and Winbat for Natural Shell, and various Cisco and Juniper systems for EasyACL. The conversational dialogue allows the user to transition from English to programming syntax at his or her own pace – if they want to go beyond natural language dialogue.

In user testing of the resultant applications from the two latter case studies, reportage that it (the intermediate language) was very useful in successfully troubleshooting problems, was a common response. This is an interesting parallel with a finding by Professor Good back in Chapter 1, where the Flip language had a translation from block-based code to plain English feature, where it also greatly helped in team communication and debugging. This reinforces one of the author's conclusions here, that such systems can significantly help end-users learn technical programming languages, not just replace them with natural language.

There are great efficiencies in programming languages with regards to specifying and communicating issues around complex tasks, that should not be under-valued going into a future with more People-Oriented Programming.

As our user interfaces in general move further towards natural language and voice input, solutions such as this framework by Wu and Liu are becoming increasingly important to People-Oriented Programming. At the significant vendor level we already have intelligent agent products such as Apple's Siri, Google Assistant (and now Google Now), Amazon's Alexa, Microsoft's Cortana and Wolfram Alpha - that extensively draw upon data, services and knowledge *out on* the Internet. However, it is at the personal level, that we tend to organise our own data, and where we service those requests and tasks, asked of us by others. We ourselves are often a part of a complex human-agent system, increasingly supported by algorithms and AI, inputting our own individual specialised knowledge into the system. The authors of this chapter are addressing issues that are physically closer to our end of the deal, as the human in these human-agent collaborations of which we are all steadily becoming a part. Solutions on our desktops and our laptops and tablets, and in our phones, which are not always connected to a network. They are also applying their natural language framework to a number of divergent cases, where programming language and scripting presents a significant hurdle to people, to cases where formalised languages were previously a requirement.

SECTION 2: NEW METHODS FOR THE PEOPLE-ORIENTED PROGRAMMING PARADIGM

As with the programming languages and environments covered in the last section, a similar rate of progress has been made to help along with the third element of People-Oriented Programming: employing self-ethnography, which may include the use of sensors (informational probes), to define and refine the individual's requirements and clarify the context. This is being enhanced with ongoing improvements and innovations in new methods for: requirements gathering, analysis, through to the design of applications - for individuals and around an individual's situatedness.

Sophisticated sensors in smartphones (e.g. NASA has sent them into orbit as cheap satellites) and smart watches, and various sensors in other wearable technology and in the home, are leading to collections of vast amounts of data about an individual, and about the immediate environment of the individual. The Internet of Things (IoT) is just one strand of research addressing these new sensor technologies. POP research is interested in how real-time *data* can be turned into highly useful *information* about the individual for the individual, but also for greater society. I.e. the empowered individual is increasingly becoming a global citizen - one such example is the Citizen Science projects aimed at helping prevent further loss of endangered species (Preece, 2017). Self-ethnography - such as the keeping of diaries, photos and video, including those in social media - and quantifying information from arrays of sensors on and about the user in their environment, is quickly helping to bridge a long-recognised information gap between ethnographers capturing user requirements, and software engineers developing a system based on their interpretation of what a 'typical' end-user wants and needs. That the users themselves may become coders is helping to bridge the gap between the software that an individual needs, and those able to write/create it. The following chapters cover a range of people-oriented methods that can bridge that gap successfully.

The first chapter in this section by Graham and Rouncefield, covers the variants of *cultural probes* that are eminently suitable for self-ethnography as a *requirements method* for POP. They present significant insights and concluding recommendations about how such data can and should be used. In the second chapter, I showcase a *design method* suitable for a significant class of applications that end-users could develop themselves, given the tools. Chapter 7 outlines a significant body of work by an eminent group of collaborative professors, that also calls upon cultural probes for requirements gathering, but then develop an innovative *design method* incorporating models from the agent-oriented paradigm (goals and roles), and combines these with *quality goals* and *functional goals*. Chapter 8 builds on the initial work in Chapter 7, in a digital media design context, adding *emotional goals* to the quality goals and functional goals, evolving the initial design method into a full people-oriented *development methodology* that has been successfully used in teams of as many as 300 digital media students "right down to teams as small as one".

Chapter 5

In this chapter Connor Graham and Mark Rouncefield explore the use of Cultural Probes as a method in People-Oriented Programming. They investigate Cultural Probes that have previously been used by ethnographers and designers, to gain fresh perspectives in areas of technology usage, that are very sensitive and often domestic in nature. They begin with a literature review of the different branches of probes since the seminal paper on cultural probes by Gaver et al in 1999. They discuss the variety of 'x' probes available: empathy probes, digital cultural probes, mobile probes, urban probes, value probes, domestic probes and cognitive probes – which have common features, including: they capture some biographical account of individual people; they give visibility to the things in a person's life; they focus on people being experts in their own lives. Two other types of probes are detailed in the literature: technology probes, and informational probes. The first involved exploring the potential for new technologies to support communication for diverse, distributed and multi-generational families. While information probes were first deployed to help understand and assist people and their carers in residential care centres for: elderly people living at home; and hostels for former psychiatric patients.

The usefulness of ethnographic approaches was being questioned in the late 1990s, in terms of both general applicability and design relevance, regarding the observations and documentation coming from them. Cultural Probes were one response to such criticisms. These probes provide a way in which we can see everyday activities as social actions embedded within some larger socially organised domain. The various ways that the day-to-day activities of people are documented, provides access to the ways in which people understand and conduct their lives.

Furthermore, insights gained on individuals' lives are closely coupled with insights about design, particularly the design of new things and new solutions. Some see this as a sign that we have entered a 'post-methodology' era where diversity is key and fitness for purpose and flexibility are critical; and that a multitude of probes are available to help achieve this. Whereas Graham and Rouncefield see that culture probes actually "*constitute a 'people-oriented' method if not a methodology*."

Probes allow the movement away from laboratory confined studies with generalised outcomes, to context-aware and people-oriented observation, down at the individual level, and how we interact socially with those people closest to us in work, leisure and home-based situations. We can build up a unique picture of personhood, of self, with the help of such probes. Interestingly, the study that Graham and Rouncefield embarked on, can be seen as a middle-ground between laboratory and study of individuals – they chose to study a community residential care setting for former psychiatric patients, with both information probes and technology probes. One site was staffed with shift workers around the clock, while the second nearby site supported semi-independent living with support staff as needed. This gave them access to a site where domestic lives are already heavily observed, routinized and facilitated, removing the novelty of introducing technology and overseers, while still being able to observe the workflow and patterns of individual lives via their new probes. The second site, one kilometer away, allowed them a more normalized arrangement.

Many of the insights they gained are applicable to the wider population. Think of parents as a team of two, responsible for their children and themselves 24/7, as well as situations where neighbours and relatives look out for elderly citizens in their midst. Also, the most generic of insights were gained. For example, technology probes required a reactive and adaptive participation, while the informational probes encouraged investigative and reflective participation. Both forms of participation made available descriptive data that acted as a resource for design: one that was more temporal and thematic, while the other, more spatial and episodic.

Another observation they make is that the work required to maintain the technical order of domestic technologies, has now filtered down to the individual, to non-expert household members. This can range from what is described as system administration work, to more broadly, the "*digital plumbing*" or "*the actual work of installing digital* *technologies in a setting*" that ensures newly installed technologies "*fit with a whole constellation of pre-existing organisational arrangements, both physical and social*". Beyond administration, home users are also buying, upgrading, customising and even making new domestic software and hardware solutions that "*contribute to the socio-technical order of their households*" … "*the individual person is increasingly subject to and responsible for the work of maintaining their digital personhood*". This discussion of probes as a method has implications beyond my second element of POP above regarding coding. They note that: "*programming-like activities have entered, even sneaked, into many people's everyday lives. There is a raft of everyday, unnoticed work that people do to configure, administer, tailor and even create and publish through computing systems*". Which suggests that the revolution we are seeing in block-based coding amongst current school students, for example, may well flow into the domestic space in many of those areas of *digital plumbing*. Allowing "*home users (to) also upgrade, tailor, purchase and even develop new domestic software and hardware solutions that contribute to the socio-technical order of their households*". There is already something of this evident in the early adopters of block-based coding within the maker movement.

They explore other opportunities that probes present for People-Oriented Programming as a people-oriented method. For example, that 'probing' can include automatically collected datasets of different scales and specificity, which can happen in parallel with people's domestication of a technology. However, the pertinent questions that arise are what can one (ethically) do with such data and how we can make sense of it. The methods the authors discuss in this chapter, give appropriate perspective to these questions, so that they can be more easily answered in a given situation. For example, the demonstrated information flows, shine light on the pathways that people make around and through different technologies. And beyond the insights gained, some probes have the potential to highlight, record and even replay to the individual concerned, how they did it 'last time' in a perfect memory sense.

On the increasing amount of automation in the collection of data about an individual, the authors outline questions regarding responsibility, visibility, performance and expertise that are essential to ethically understand, and to further develop the potential of new forms of probing for People-Oriented Programming. That the kinds of efforts made by some researchers in the spirit of 'total capture' of a life, autonomously, at least for People-Oriented Programming, seems to miss the point entirely. "*What we choose to collect through Probes is informed and selective and helps drive how we 'make sense' through selection, extraction and presentation of the data we collect. In addition, what people effortfully collect and choose to put on display (e.g. with Informational Probes) is of some importance to us*", and if such 'work' is outsourced to autonomous probes, gone too is that effort that made it so.

Through the probes used, the researcher may acknowledge 'the user' not only as a content producer but also as a cultural commentator, a diarist, a photographer, and so on, in a place in the real world, as a unique person who exhibits their identity and self, in and through the physical and digital worlds. We return to these last most important subjects about Identity and Self, in depth in Chapter 9 titled 'Formation and Control of Identity in a Social Media World'.

Chapter 6

This chapter is titled 'A Design Method for People-Oriented Programming: Automating Design of Declarative Language Mashups on the Raspberry Pi'. In it, I detail a method that was conceived as a People-Oriented Programming method, from the ground up. From a coding point of view, it applies to declarative languages rather than procedural languages. While the logic languages such as Prolog are very much less used than mainstream procedural and object-oriented languages, the widely used and easily understood SQL language is declarative in nature. SQL has a *function point count* (a metric used to measure the expressiveness of programming languages, based on large databases of past projects) which is in the top 3 languages of most comparisons done (e.g. http://www.qsm.com/resources/function-point-languages-table; http://www.ifpug.org/Conference%20Proceedings/ISMA3-2008/ISMA2008-14-Jones-using-function-point-metrics-for-software-economic-studies.pdf). The SQL language is ideally suited to first-time coders for certain types of applications, an opportunity that has not been exploited well in the movement to bring coding to novices. One aspect responsible for that lack of adaption is that good information design skills greatly reduce the complexity of the SQL needed to solve a particular set of problems. The TANDEM method detailed in this chapter significantly eases the need for those information design skills. By applying a series of rudimentary steps, a novice is able to find an acceptably good way to structure domain level data. Having some measure (a starting data set) of the domain data assumes a previous requirements gathering exercise. For a People-Oriented Programming approach to requirements gathering, the range of cultural probes discussed in the previous chapter, offer an eminently suitable choice for gathering the data needed for a starting point with TANDEM. Other traditional requirements methods are also a possible precursor to using TANDEM.

The application developed in the case study within the chapter, does so on a Raspberry Pi miniature computer, fast becoming a useful POP platform, used heavily in the UK primary and secondary education sectors (Banks Gatenby, 2017), and gaining a significant footprint in those school sectors in other countries, and also in the maker movement market in general.

The case study describes the TANDEM design method and demonstrates the use of it in detail, calling upon it to design a mashup application of services and local data stores that solves the so-called *movie-cinema problem*. An implementation of the newly designed movie-cinema app is then built within the DigitalFriend, an end-user oriented IDE that fits into the second element of POP defined above. The DigitalFriend example uses an inbuilt declarative logic language called CoLoG, but it can equally use SQL scripts. The early part of TANDEM uses a simple task analysis technique, describing all the *doing* activities in the proposed system, always beginning with a verb, e.g. *Reserve* seats. *Read* a book. These verb-first activities from Task Analysis, are familiar to all Facebook users, which are discussed below in Chapter 9. Furthermore, a significant part of the TANDEM design method is then automated within the development tool itself. This automation removes the most skilled task required by TANDEM of the end-user: the automation of the process of data normalisation. The automation applies data normalisation to the initial model of components and data sources that feed into the mashup. The presentation here relies on some understanding of data normalization, so a simple example is presented. After this demonstrated example of the method and the implementation, a discussion ensues on the applicability of a model achievable by end-users using TANDEM coupled with the automated normalization process built into the IDE, versus using a top-down model by an experienced information analyst. In conclusion, the TANDEM method combined with the automation as demonstrated, does empower an end-user to a significant degree in achieving a workable mashup of distributed services and local data stores and feeds. Such a powerful combination of methods and tools will help the Raspberry Pi become a significant people-oriented programming platform, beyond just a platform for teaching novices to code. Furthermore, the TANDEM method does have broader applicability to designing a broad class of logic programs, complementing the use of collected patterns in logic programs.

With 15 million units of the Raspberry Pi having sold (by August 2017) it is not only a formidable common platform for makers and for teaching school kids something of coding and computer science, but is also a fully functional Linux machine with desktop level tools, together with low level interfaces to all sorts of sensors and physical world devices to be controlled. Therefore, it is not surprising that the Raspberry Pi has also raised the interest of many researchers as a platform for building sociotechnical IoT systems for end-users (see Chapter 3 for an IoT application upon a *similar* low-cost computer board). It is with this in mind that the DigitalFriend was ported to the Raspberry Pi. Using the DigitalFriend, end-users are able to build mashups of REST and SOAP web services, together with local resources and processes, and other information sources including data streams from sensors and devices, into newly envisaged and often personalised POP applications.

Chapter 7

Professors Paay, Sterling, Pedell, Vetere and Howard have assembled an important body of work over 8 or 9 years that bridges a previous gap in the understanding between ethnographers, interaction designers and software engineers, whenever they came together to design and create socio-technical systems. They were early adopters of cultural probes (discussed back in Chapter 5), and have also adapted models from the Agent-Oriented paradigm - in which several methodologies use goal models that are tied to role models - in the requirements, analysis and design phases of system development. They take their agent models from those detailed in (Sterling & Taveter, 2009). While ethnographic data is often ‘messy’, the models used by software engineers are conversely, systematic and well organised. This can lead to either, a regular clash between those different ways of working, or to a productive team with complementary strengths via a shared understanding across the whole process. The authors have developed a method which promotes such a shared understanding, very effectively, during the whole time the team are together. In this chapter they consolidate their findings and present them with some clarity.

To uncover the social requirements for a new system a design team must identify non-work-focussed and non-goal-oriented interactions that happen between people. Cultural probes are particularly good for collecting data about personal and intimate aspects of people’s lives. Data generated from cultural probes by the user themselves, can include photographs, postcards, drawings, diary and scrapbook entries, and more, which can be supplemented with contextual interviews to enrich understanding. With such ethnographic field data, the problem is how to derive usable and useful system requirements while retaining the insights and inspirations collected from the alternative views. Mulling over this material, an interdisciplinary team will often uncover and generate ambiguity in the system requirements.

Traditionally, software engineers do away with ambiguity before the design stage, by forcing choices early on during the analysis phase. However, ambiguity can be seen as an opportunity rather than a problem, particularly when developing social systems, as it allows people to interpret situations for themselves, and thereby establish a deeper understanding of what is needed. The authors use high level *quality goals* to capture such ambiguity and carry it through the whole design process. Quality goals are non-functional goals used to encapsulate social aspects of the context of use, bringing them into the models. They include *emotional goals* within the quality goals, such as: *having fun*, and *feeling secure*. Quality goals become a vehicle to carry subtle nuances of social interactions from the field data through to the final system design. The authors turn quality goals into quality attributes that in turn both enrich and constrain the goal and role models from the agent methods.

They present their method in a case study, the *Secret Touch* system, with quality goals representing information about intimate interactions that occur between couples. The study is an extension of their earlier Mediating Intimacy project: a study focused on supporting intimates (i.e. life partners, parents and younger children, adult offspring with dependent parents, and so on) with both co-present or separated by distance. The *Secret Touch* scenario was chosen for the design and implementation of a prototype of a socio-technical device for mediating intimacy in couples.

They have since extended their experience with the methodology through modelling a range of more complex socio-technical situations: guiding the building of software systems to encourage grandparents and grandchildren to have fun over the internet; in the mental health area, the development and pilot study of a novel digital intervention to promote personal recovery in people with persisting psychotic disorders; and the design of an emergency alarm monitor systems to help older people wishing to continue living at home.

These quality goal and role models, together with an innovative notation have been incorporated into an agile development methodology, that is now routinely used in teaching interaction design and software engineering at tertiary education level - which is detailed in the next chapter.

Feedback from the members of the interdisciplinary design team confirm that the role and goal models, together with the innovative notation, are easily understand by all team members, which is a major contributing factor of the success of the method. For that reason, there is every chance that the study subjects and actual users of these systems could also easily understand the method. Combine that with the fact that the cultural probes used in requirement gathering, were self-administered by the study subjects in this project, and what you have here is a method that is very well placed to play a bigger role in People-Oriented Programming into the future.

Chapter 8

Where the previous chapter presents a case study with an interdisciplinary design team made up of software engineers, ethnographers and interaction designers, research in this chapter involves the same starting method, but within Digital Media Design courses where the projects are generally about the design and creation of video games, and the student team members are a younger and more diverse group. As such, for the method it is a good further proving ground as a People-Oriented Programming method, demonstrating that it should be well placed in the future of POP. Particularly as there is some evolution of the method through the work presented here, involving human emotions, that has been folded back into the usage of it in general.

A striking difference in the domain and the digital design discipline, as compared to the software engineering and interaction disciplines, comes up early in the chapter:

Realising emotional factors is fundamental to design disciplines, from industrial and communication design to architecture and fashion design ... The consideration of emotions is important in the development of many sociotechnical systems. If a computer game does not feel fun, we will not play it; if an ecommerce website does not feel trustworthy we will not purchase from it; and if a social networking application does not feel engaging we will not use it. We describe these as the emotional goals of the system, which we define as goals that aim to affect people's emotional state or wellbeing… Digital media design methodologies have a tradition in realising emotional goals by using tools and processes such as: emotional colour scripts, flowcharts, rich pictures and mood boards… Designers model emotion and communicate the feel to clients and to each other using this visual, and in some cases, audio and textural language.

However, Marshall recognised the need for an inter-disciplinary design approach to solve the increasingly complex digital technologies being used and the diverse collaborations taking place in creating new digital media applications. The agent-oriented models (Sterling & Taveter, 2009) that were being applied to sociotechnical systems by the Paay et al. the authors of Chapter 7, fitted the need that Marshall had personally identified. He incorporated emotional goals and gave them equal importance, to the quality and functional goals duo, that Paay et al. had used to hold ambiguity in the minds of all team members for as long as possible in the design process. By advocating this new category of goal – *the emotional goal* (and introducing a new icon to the visual notation, in the shape of a heart) – the method has been evolved considerably, improving the way that sociotechnical systems are designed and created by inter-disciplinary teams, large and small.

Indeed, in the case study in this chapter - Aspergion Galaxy, a video game (a learning resource) for secondary school students that promotes *respect* for people with Asperger's Syndrome - Marshall applied the method to a team of 160 student developers over two semesters, and got a significant result. Conversely, the method has been used successfully by individuals. Marshall concludes with some statistics and results around the success of the method, with regard to teaching new digital media design:

Between 2011 and (late) 2017 the process has been repeated in over 200 separate projects, ranging in scale from 1 to approximately 300 developers, and at academic levels including 3rd year, Honours, Masters, Doctoral, Post-doctoral students, *Industry-based and Research projects. Outcomes have included the development of computer games, websites, animations … software applications, systems and services, advertising campaigns, branding and physical products.*

Student feedback consistently reported that the goal models were intuitive and easy to use, and they provided common, well-understood artefacts around which progress can be communicated to all, and also helped to focus students on the primary project objectives throughout the life of the project.

From a marketing perspective, what the author's research is reporting at the end of this chapter, is a strategic advantage no less, in the way his university is teaching Digital Design, based squarely upon this new method that incorporates emotional, quality and functional goals, in equal measure. A method that has all the hallmarks of a good *POP methodology*, both scalable to large and diverse groups of people, and that still works right down to teams as small as *one*.

SECTION 3: THEORY IN THE PEOPLE-ORIENTED PROGRAMMING PARADIGM

As previously highlighted, a new software design and development paradigm doesn't come into existence in a neat chronological order like: *Theory, Tools, Methodology, Framework, Applications*. And so it is with the *theory* part of People-Oriented Programming. However, there have been advances in the area of theory coming from disparate disciplines, that do sit well within the embryotic POP paradigm. The two chapters in this Section hold important aspects of how relevant theoretical considerations from diverse disciplines have already impacted the other areas of POP, namely, the theory behind some of the tools and the methods described in Section 2. The focus in Chapter 9 upon the nature, formation and control of *Identity*, is timely, as there is something of a groundswell lately, aimed at harnessing the multitude of smartphone-empowered individuals, into groupings that both think and shoulder some responsibility at a global level, but act locally within their immediate communities. Two such initiatives are mentioned in the chapter. The first is from the social media platform industry itself, with the founder of Facebook, Mark Zuckerberg, flagging the intentions of his significant development resource, to provide the infrastructure for building a global community (Zuckerberg, 2017). While the second, mentioned earlier, is the various existing citizen science projects, such as those cited by Jennifer Preece in (Preece, 2017). This groundswell is very relevant to POP as it is a paradigm about empowering the individual, and the subject of Identity is right at the core of 'individual'.

There is a case study in Chapter 10 that thoroughly documents the complete process in specifying and designing an emergence alarm system for elderly people, who wish to continue living independently at home for as long as possible. It is a safety critical system. There is people-oriented method here within an overall software engineering methodology. The chapter also brings into focus, both the original and the ongoing role of software engineering in the People-Oriented Programming paradigm.

Chapter 9

This chapter by Dr Christine Yunn-Yu Sun et al. is titled 'Formation and Control of Identity: In a Social Media World'. It largely comes from the Cultural Studies discipline, but then that investigation of identity is reinforced with parallel and complementary theory from the *Psychology of Sub-selves* which is presented by way of a case study. The chapter begins by exploring the construction of identity in online communities and websites for social purposes, and its consequences in terms of how one's online identity may be utilized to such an extent that one's real-world identity is either enforced or eroded. It does so by investigating the very nature of Identify, coming predominantly from a Cultural Studies and philosophical view, although it also cites some related findings and advances in Computing and Information Systems (IS) research. In section: *Identity as a Self-Produced Continuity and the Media* - the authors investigate the concept of identity in the real world. They then investigate it in the online world in the section: *Online Identity, Its Construction and Circulation*. The section titled: *Social Media, Identity and the Meaning of Self* - examines how an individual positions herself, including who she associates with and why she flags this positioning, to others. For context, it looks at several mainstream social media platforms used, when doing so. This includes an overview of Facebook's recently divulged plan to build a global community, and compares that to certain citizen science projects that also often have global-level goals.

Interestingly, the biodiversity data collected by individual in these communities, is typically "*collected using smartphones (which) can include photos, comments, numerical data, video, and sound, together with metadata (e.g., time, date, and geolocation logging)*" – i.e. crowd-sourcing people with their increasingly sophisticated smartphones and attached/nearby sensors, as *information probes*, discussed by Graham & Rouncefield at length back in Chapter 5 on cultural probes as a POP method. That chapter was focused on self-ethnography. The biodiversity citizen science usage of these same probes, often scales to *ethnography* at the global level.

The section: *A Case Study in Representation and Control of Identity and Privacy* - presents an innovative approach to dealing with those issues and actions that revolve around the control and disclosure of identity, as detailed in earlier sections. Initial emphasis is on *knowing oneself well* through the interests and roles in one's life. Conceptual models of example POP software and an instantiated digital-self built using the software, are shown, as a pro-active tool that helps a person manage identity as their life evolves and as they interact in the online world, both through social media platforms and the greater Internet.

The central argument that runs across this chapter is two-fold, firstly, it promotes and demonstrates an initial shift in focus from the management of online identity to the nature and significance of *Identity* itself, whose construction may be conceptualized as a process of sense making and strengthening. Then, armed with a better understanding of identity, one can focus back upon the management of it more effectively, with a view to the individual taking more control of their own identity, particularly with regard to privacy. They emphasise the importance of this with: "*At stake is mass trust in the service providers of these new social media and search platforms. Also at stake is the last interior privacy firewall of Self*'. They further emphasise the importance of this at the global level, as the technology space (most notably but not limited to, social media, search engines, and mobile apps) is increasingly transitioning us all into members of a global community, one that needs to be functional not dysfunctional.

In several chapters of this book including this one, there is a recognition that the increasingly empowered individual, while a member of multiple communities both online and in the physical world, is progressively becoming a global citizen, or in this case: forming or reinforcing something of a global identity. This chapter demonstrates that the People-Oriented Programming tools and methods have a significant part to play in this ongoing empowering process, and that *Identity* research from multiple disciplines, including - cultural and philosophical studies, psychology, computing and information systems - is an important theatre for the *theory* that will continue to play a part in the POP paradigm.

Chapter 10

In this chapter by Leon Sterling, Antonio Lopez-Lorca, and Maheswaree Kissoon-Curumsing, a case study is used to show how they incorporate *emotions* as a first-class entity in the requirements and modelling phase of a software engineered system. The case study is one of emergency alarms for elder people who continue to live independently in their own homes, with the help of an on-call emergency system, rather than move into retirement villages or similarly staffed facilities. They point out that their emotional goal modelling technique is particularly well suited to most socio-technical and domestic systems. In the domain of the case study, previous alarm systems tried by participants in the study, involved two devices: a pendant or wristband to be warned at all times, with a button on it to push in emergencies to alert a call centre that assistance is required (e.g. a fall they cannot get up from); and a wellbeing checking device that the elderly person is required to press within a fixed time period each day, to indicate to a carer/relative that all is going well. The design of these traditional alarms in no way included the emotions of people involved. As such, many elderly people viewed wearing the pendant as a stigma and sign of old age, while the well-being check-in, they saw as an infringement of their independence.

The authors demonstrate, by the inclusion of emotional goals as a design criteria, how much better they are able to gather requirements, analyse the problem, and design a more appropriate solution. For example, the wellbeing checking device they come up with, is not a button, but a touchscreen connected to the relative or carer, on which they receive photos each day (e.g. of a grandchild, say), and that by simply touching or commenting on it, it becomes an implicit wellbeing check-in that is positively accepted into their lifestyle.

They use goal models (drawing quality goals and emotional goals from the requirements gathering phase), and identify the various roles played in using such a system. They draw upon several deep veins of research for their methodology: role and goal models from the agent-oriented software engineering field; and viewpoint analysis from requirements engineering. They also use Scenarios with the various stakeholders in a project, to flush out the required functionalities, and pick-up any missing details from the role and goal models – much as scenarios are used in Human Computer Interface (HCI) to analyse and design systems.

The real innovation in this approach is in identifying emotional goals and placing them on an equal footing with other quality metrics such as performance and reliability of software engineering systems: "*It can be argued that emotional goals are different from traditional quality goals such as performance and reliability, because they are properties of the user rather than the system. Emotional goals are subtle, ambiguous, and difficult to measure*." This is in line with the approach that Marshall takes in Chapter 8 in the digital media design area, which lead to the notation used for emotional goals here.

While their methodology is within the field of software engineering, typically where a team of people including stakeholders: determine the requirements for a new system, analyse, design and then build/or have it built. There is people-oriented method within an overall software engineering methodology, by way of the emotional goals being included: "*Emotional goals are linked to roles, which represent stakeholders in the system, and specify a desired state of emotion or* *wellbeing of an agent playing those roles. Emotional goals represent how people feel, so are a property of people, not of the system.*"

One of the observations during the case study, is just how easily the non-technical stakeholders in the project, readily took up and used the emotional goal models, to converse and explain their particular viewpoints in the project. Furthermore, the use of role models from the Agent-Oriented paradigm were also readily identified with and taken up by both technical and non-technical stakeholders.

Their modelling and methodology is extensive, as can be clearly seen in this chapter, with 34 figures representing the models used to bring the new system in the case study into existence. However, the Scenarios figures are all textural and easily understood. Many of the visual models are equally as easy to comprehend. Nonetheless, what is evident in this chapter is a tendency for software engineering to lead to over specification of a system, from a people-oriented method point of view – something also recognised in Chapter 7.

One way that complexity and heavy specification is alleviated, is through the development of new software development *tools*, and indeed several of the authors of this chapter and Chapter 7 are involved in just that: improving their methodology by developing new tools and refining the visual notations used. At some stage in the near future, their approach is likely to become a fully people-oriented *methodology*. They also encourage other developers of tools and applications, to fold emotional goals into their existing processes, to improve their outcomes. We only have to look back at the four chapters in Section 1, to see that it has been innovation in the tools themselves since about 2007 forward, specifically in the integrated programming environments (IDE) rather than in the syntax of programming languages, that has put coding within range of most school children.

This chapter could have been placed in Section 2 given the significant amount of people-oriented method within it, but it is doubly useful here in this section on *theory*, as it gives some good backgrounding to the agent-oriented goal and role models used, and also shows how and why emotional goals have been folded into a software engineering methodology. It also reminds us that many applications should and will remain in the software engineering realm. Emergency alarms for elderly people wanting to continue living independently at home for as long as possible, is a safety critical situation, and not something you want tinkered into existence. And, as is emphasised in the fourth element of the People Oriented Programming paradigm at the outset, the highly usable end-user development tools that are required for the paradigm to succeed, also need to be well-engineered.

SECTION 4: PERSONALIZED LEARNING ENVIRONMENTS AND THE PEOPLE-ORIENTED PROGRAMMING PARADIGM

We saw in Section 1 a lot of the exciting developments in POP is in new programming environments for coding, and these coding environments are often much more than programming editors. Several of them including Scratch and Greenfoot, are prime examples of learning environments with large communities of likeminded people. In the recent book *Lifelong Kindergarten: Cultivating Creativity through Projects, Passion, Peers, and Play* (Resnick, 2017) the author and the prime-mover behind Scratch, Mitchel Resnick, outlines his reasons for being a strong advocate for teaching all young people to become fluent in coding: *"Most people won't grow up to become professional programmers or computer scientists, but learning to code fluently is valuable for everyone. Becoming fluent, whether with writing or coding, helps you to develop your thinking, develop your voice, and develop your identity."* While coding is only one aspect of learning in the modern school curriculum, there is considerable scope for coding activities and computational thinking in many other subjects that make up a comprehensive curriculum. For these reasons we include two chapters on Personalised Learning Environments (PLE) in this volume.

In the Editorial Preface by the guest editors for a special issue on Personalised Learning, Judith Good and Ben de Boulay (2014) characterized personalized learning learning in the following way:

Personalised learning, or the tailoring of curriculum, pedagogical approaches and the learning environment to each individual, provides a greater focus on the individual than any earlier approach to learning. In doing so, it has been concerned with adapting learning experiences and resources based on a wide variety of factors, including personal learning history, current location and context, available interactive devices, cultural preference, personality, learning preferences and gender, amongst others. A dominant source of personalisation derives from cognitive differences between individuals, whether in terms of their approaches to learning or, more commonly, what knowledge has already been acquired, and what skills have so far been mastered.

Additionally, new generation learners often like to customise their learning content and learning spaces/locales. They will personalise their interactions in the learning process, expressing themselves with their own user generated content, as much as the technology allows. And given fewer constraints than tradition learning has afforded them, they will learn whatever, wherever and whenever they desire, usually intermingled with other non-learning activities.

Personalised Learning Environments (PLEs) include technical environments which encourage and support personal learning. In a sense, several of the coding environments covered in Section 1 could be described at some level as PLEs. For example Scratch now has in excess of 20 million online projects, with commenters, collaborators and followers, within which a person learning to code, can call upon and borrow from, *whenever* and from *wherever* they desire.

However, the two chapters that follow are more focused on generic PLEs – those in which support for an individual learner of any subject (*whatever*) can be facilitated.

In Chapter 11 titled 'Intertwining E-Learning Technologies and Pedagogies at the System Design Stage: To Support Personalized Learning', George Weichhart and Chris Stary explore the issue of how a learning management systems (LMS) might be evolved so that it could explicitly incorporate different pedagogical models, depending on the individual student and teacher (or mentor). In doing so, however, they are careful to propose a model where the pedagogical aspects of the LMS remain decoupled from its technical aspects, so that different pedagogies might be applied in different circumstances, or a particular pedagogy shared across different e-learning platforms and tools. This chapter outlines many of the technical requirements that would enable interoperability across the different system components needed to support interchangeable pedagogy models.

In Chapter 12, I make the case that the future of learning environments lies in the merging of the better aspects of Learning Management Systems (LMS), with those popularised in Social Networking platforms, to enable the personalization of the individual learning experience. The resulting system is a PLE (Personal Learning Environment). After examining the details of a particularly flexible LMS, followed by the investigation of several key data structures behind the Facebook social networking platform, this chapter then demonstrates how such a merging can be done at the conceptual schema level, and presents a subset list of the novel features that it then enables.

It is unclear how LMSs and PLEs will evolve as larger percentages of current and future young students become fluent in coding in one of more of the languages covered in Section 1. However, the glimpses we see in the collaborative learning communities, such as those within Scratch and Greenfoot, are inspiring. There is even potential for those learning communities to use their knowledge gained within those empowering environments, to expand their projects and tutorials to envelop subjects in the wider curriculum of their immediate educational organisations. If the maker movement is anything to go by, should young coders in the midst of that movement, move their innovative focus towards the local curricula, beyond just coding and foundational computational thinking, the synergy arising from a pairing of the People-Oriented Programming paradigm (POP) and newly evolving

Personalised Learning Environments, could transform secondary education at the very least into a learning hothouse.

As noted earlier in the introduction of Chapter 9 with its focused upon identity, the large corporate social network platforms, as well as the niche social websites (and apps) such as the biodiversity citizen science communities, are transitioning us all into an increasingly global community, one that needs to be more functional rather than less. Empowering individuals and raising global consciousness, are it seems, two sides of the one coin. Just as Preece (2017) emphasised a limited area of endeavour (in terms of one's full identity) targeted at making a positive difference and acting locally - in that case saving species and maintaining as much biodiversity as possible by contributing in some small way to the management of the biosphere – the pairing up of POP with personalized learning environments (PLE), aimed at local curricula and local civic problems, but drawing on an online community of like-minded, collaborative peers and mentors as needed, would be an admirable application domain for developing the future of the People-Oriented Programming paradigm. Advanced PLE platforms operated by the safe sets of hands that belong to currently committed educationalists, is a good and familiar place, domain and cohort for researchers and academics in the POP space, to concentrate much/more of their future efforts.

Steve Goschnick
Swinburne University of Technology, Australia

REFERENCES

Banks Gatenby, A. (2017). Developing Critical Understanding of Computing With the Raspberry Pi. *International Journal of People-Oriented Programming*, *6*(2), 1–19.

Good, J. & de Boulay. (2014). Guest Editorial Preface. *Special Issue on Personalised Learning. International Journal of People-Oriented Programming*, *3*(2).

Goschnick, S. (2009). People-Oriented Programming: from Agent-Oriented Analysis to the Design of Interactive Systems. *Proceedings, HCI International*. 10.1007/978-3-642-02574-7_93

Graham, C., & Goschnick, S. (2006). Augmenting Interaction and Cognition using Agent Architectures and Technology Inspired by Psychology and Social Worlds. *Universal Access in the Information Society*, *4*(3), 204–222. doi:10.100710209-005-0012-x

Preece, J. (2017, March-April). How two billion smartphone users can save species! *Interaction*, *24*(2), 26–33. doi:10.1145/3043702

Resnick, M. (2017). *Lifelong Kindergarten: Cultivating Creativity through Projects, Passion, Peers, and Play*. Cambridge, MA: MIT Press.

Sterling, L., & Taveter, K. (2009). *The Art of Agent-Oriented Modelling*. Cambridge, MA: MIT Press.

Section 1

Programming Environments for People–Oriented Programming

Highly usable and well-engineered development tools are required for People-Oriented Programming so that the end-user can design, code, configure, make or mashup solutions to their own problems, either afresh or customised from solutions to similar problems or needs. This section is made up of four chapters that together present the state-of-the-art of the programming languages and coding environments of the People-Oriented Programming paradigm.