Paper index by session

Session 1

Code as Art, Art as Code: On the Use of Poetry and Paintings in Programming Education
Felienne Hermans

Methods in user oriented design of programming languages
Clayton Lewis

A Systematic Literature Review of Cognitive Dimensions
Mohammad Hadhrawi, Luke Church & Alan Blackwell

Session 2

Using Cognitive Dimensions Questionnaire to Evaluate the Usability of Security APIs
(Work in Progress)
Chamila Wijayarathna, Nalin A. G. Arachchilage & Jill Slay

Digital Makers: Ownership, Personalisation and Purpose
Chris Martin and Janet Hughes

Programming Education to Preschoolers: Reflections and Observations from a Field Study
Alaaeddin Swidan & Felienne Hermans

Scores & Scripts — a Bestiary of Intents
Luke Church & Joana Chicau

Session 3

Tica: An environment for exploring tangible vs. screen-based programming
John Wilkie & Judith Good

Finding Patterns in Visualizations of Programs
Catalin-Florian Perticas, Bipin Indurkhya, Razvan-Valentin Florian & Lehel Csato

Dynamic Translation of Spreadsheet Formulas to Problem Domain Narratives
Bennett Kankuzi

Session 4

Towards Webcam-based Eye Tracking in the Eclipse IDE
Sebastian Lohmeier

The role of the cognitive style in improving the learning to program
Margherita Pasini, Ugo Solitro, Margherita Brondino & Daniela Raccanello
Towards an IDE to Support Programming as Problem Solving  
*Nicholas Nelson, Anita Sarma & André van der Hoek*

On the Nature of Programmer Expertise  
*Chris Parnin, Janet Siegmund & Norman Peitek*

**Session 5**

Investigating high-achieving students’ code-writing abilities through the SOLO taxonomy  
*Ayman Qahmash, Mike Joy & Adam Boddison*

H.I.D.E.: A Virtual Reality Debugging Environment  
*Nicolas Slack & Kate Howland*

Naming Guidelines for Professional Programmers  
*Peter Hilton & Felienne Hermans*

**Session 6**

User experiences in a visual analytics business  
*Mariana Marasoiu & Alan Blackwell*

Sprego, End-user Programming in Spreadsheets  
*Maria Csernoch & Biró Piroska*

If What We Made Were Real  
*Antranig Basman*

**Doctoral Consortium**

Developing a Systematic Approach to Evaluate the Usability of Security APIs  
*Chamila Wijayarathna*

Applying Asynchronous Gaze Sharing and Social Navigation for Problem Solving Principles in an Programming Environment  
*Fabian Deitelhoff*

Problem-Solving Applications in Developer Environments  
*Nicholas Nelson*

Block-based languages for professionals  
*Robert Holwerda*
If What We Made Were Real
   Antranig Basman

Scores & Scripts — a Bestiary of Intents
   Luke Church & Joana Chicau

Sprego, End-user Programming in Spreadsheets
   Maria Csernoch & Biró Piro ska

Applying Asynchronous Gaze Sharing and Social Navigation for
Problem Solving Principles in an Programming Environment
   Fabian Deitelhoff

A Systematic Literature Review of Cognitive Dimensions
   Mohammad Hadhrawi, Luke Church & Alan Blackwell

Code as Art, Art as Code: On the Use of Poetry and Paintings in Programming Education
   Felienne Hermans

Naming Guidelines for Professional Programmers
   Peter Hilton & Felienne Hermans

Block-based languages for professionals
   Robert Holwerda

Dynamic Translation of Spreadsheet Formulas to Problem Domain Narratives
   Bennett Kankuzi

Methods in user oriented design of programming languages
   Clayton Lewis

Towards Webcam-based Eye Tracking in the Eclipse IDE
   Sebastian Lohmeier

User experiences in a visual analytics business
   Mariana Marasoiu & Alan Blackwell

Digital Makers: Ownership, Personalisation and Purpose
   Chris Martin and Janet Hughes

Problem-Solving Applications in Developer Environments
   Nicholas Nelson

Towards an IDE to Support Programming as Problem Solving
   Nicholas Nelson, Anita Sarma & André van der Hoek
On the Nature of Programmer Expertise

Chris Parnin, Janet Siegmund & Norman Peitek

The role of the cognitive style in improving the learning to program

Margherita Pasini, Ugo Solitro, Margherita Brondino & Daniela Raccanello

Finding Patterns in Visualizations of Programs

Catalin-Florian Perticas, Bipin Indurkhya, Razvan-Valentin Florian & Lehel Csato

Investigating high-achieving students’ code-writing abilities through the SOLO taxonomy

Ayman Qahmash, Mike Joy & Adam Boddison

H.I.D.E.: A Virtual Reality Debugging Environment

Nicolas Slack & Kate Howland

Programming Education to Preschoolers: Reflections and Observations from a Field Study

Alaaeddin Swidan & Felienne Hermans

Developing a Systematic Approach to Evaluate the Usability of Security APIs

Chamila Wijayarathna

Using Cognitive Dimensions Questionnaire to Evaluate the Usability of Security APIs
(Work in Progress)

Chamila Wijayarathna, Nalin A. G. Arachchilage & Jill Slay

Tica: An environment for exploring tangible vs. screen-based programming

John Wilkie & Judith Good
Code as Art - Art as Code: 
On the Use of Poetry and Paintings in Programming Education

Felienne Hermans
Delft University of Technology
Software Engineering Research Group
f.f.j.hermans@tudelft.nl

Abstract
Programming is often taught by having students do practical programming exercises. From simple string reversal to search tree, the examples and methods of teaching mimic the life of a professional programmer in a sense. This leads to young children developing an idea of what programming is. We found that children under 12 already have clear preconceptions of what programming is for.

Can we design educational materials to battle this notion? Can we teach programming by using less traditional forms or viewing programming? In this paper we describe a four part course called Code as Art - Art as Code. It uses poems and paintings to teach novices and experienced programmers to see source code differently. In the first two lessons, participants practice viewing source code as a poem or as a painting (Code as Art). In the other two, they use source code to generate poems and paintings (Art as Code). We describe the scientific and creative rationale behind both and describe our experiences in teaching each of the four parts.

1. Introduction
Programming education is in fashion: in many countries around the world, programming is mandatory now in the UK, Australia and the US. This is of course, wonderful news for the lovers of programming! In the future, more children will be able to enjoy our field of profession. However, while teaching children programming, we also teach them what programming is used for and useful for. For example, it seems that the choice of example to use for programming lessons is often an afterthought, rather than a deliberate choice. Code.org for example, while a lovely platform used by many children (they report over 280 million children served) mainly consists of lessons where children create games of control robots. You could argue that is a relatively narrow view of what programming can do.

This narrow view is not without direct consequences for children learning to program. On two separate occasions we gave a lecture for about 70 children, aged 8 to 12. In the first lecture, we just asked the children to name one thing you can create with software. The answers were quite shocking! About 80% of children named robots, web sites or games. Only a few more creative answers were given, like ‘everything’ or ‘cake’. In the second lecture, the results were similar, but we added a second question in which we asked children to be as creative as possible. Some did come up with many great things in the second question, like pets, an operating system, or a new nature to replace the existing nature when we as humanity have destroyed it. However, somewhat more realistic answers like art or music were not given. This made us wonder if it is possible to teach programming in different ways? Can we help children and adults think about what source code is in a different way? And can we design educational materials to diminish the impact of the traditional examples of programming like robots and games?

We have therefor developed a four part course called Code as Art - Art as Code, in which we help novices and experienced programmers to integrate their views of programming and art in order to broaden existing views. The course consists of four different lessons, which can be taught separately, or as a whole. In the course poems and visual artworks are used as art forms in two different roles. In Code as Art, art is the lens through which participants view code, while in Art as Code art is the subject to create with code.
The course has two parts, each with two lessons, resulting in these four lessons:

1. Code as Art: Poems
2. Code as Art: Paintings
3. Art as Code: Poems
4. Art as Code: Paintings

In the lessons 1 and 2, together forming the Code as Art part, participants are encouraged to view source code not as code, but as an artwork. This means its features can be studied, like we study art. How does this line of code sound? Do these lines rhyme? These are questions answered in lesson 1. In lesson 2 participants focus on the metaphor of visual art rather than poetry, and think about questions like How does the code look, and How does it make me feel?.

Lessons 3 and 4 as in a sense the opposite of lessons 1 and 2: we use code as a means of creating art. This is often called ‘computational creativity’ (Newell, Shaw, & Simon, 1959). Computational creativity and is a vibrant field where mainly artists and scientists participate. While many developers pride themselves on having ‘pet projects’ these usually do not include creating artworks with code. We would love for people and especially children to learn about the role of the computer as a creative partner, who can generate ideas, and with which one can even exchange ideas. Therefore both these lessons are aimed at novice programmers, but they could also very well be ran with professional developers.

Both the Code as Art and the Art as Code part start with poems\(^1\), since their textual form is somewhat similar to source code, resulting in the fact that some concepts like metrum and rhyme are easily transferred. Visual art is a nice subsequently deepening step, since it is more traditionally seen as ‘art’ and different art styles like surrealism or cubism are relatively well known among developers and novice programmers.

All four parts could be taught separately, as a whole of four, or in combinations on topic, such as only lessons 1 and 3, forming Code as Art - Art as Code: Poems. Over the past year, we have experimented with different formats, combinations and audiences, which we will describe in this paper. Overall we have found that professional programmers enjoy viewing their work in a different way, and that novices are surprised and excited by the idea of creating art.

2. Lesson 1: Code as Art: Poetry
2.1. Setup
In the first lesson, participants focus on elements of poetry that can be recognized in source code. The lesson consists of two parts that are traditionally seen as the building blocks of poetry: rhyme and rhythm.

2.1.1. Rhyme
The rhyme exercise is concerned with the sounds of code. We ask participants to vocalise what a line of code sounds like? For some lines of code, determining the ending sound might be quite simple, like this one:

```
Listing 1 – Sounds like: middle is length of m divided by two
middle = len(m) // 2
```

But how does the line below sound? You could say it is ‘m from middle’, but you might as well say it sounds like ‘m from the middle to the end’, as that is the meaning of this line of code in Python. Or even ‘take from m the items from middle to the end’.

\(^1\)In earlier versions of the course, we only used paintings, but this proved to be quite challenging for some participants
After just listing the sounds, participants are encouraged to also play with them. For example, the line from Figure 1 could also be written like in Listing 3, making it rhyme with other lines ending in ‘em’.

### 2.1.2. Rhythm

After the rhyme part, we move on to rhyme. In the first rhythm exercise, participants simply count the number of syllables of a given line of code. We asked them to add dots in between the syllables, and add the number as a comment so we could easily see and discuss the choices they made, as shown in the screenshot below.

![Figure 1 – The six syllables of a line of code added as comment](image)

After simply observing the rhythm, participants also play with the metrum of a collection of lines. What can be changed to make a whole method have a certain rhythm? A simple exercise is to manipulate two lines of code so that they have the same number of syllables.

As an example, consider the two lines below:

#### Listing 4 – Two lines with a different number of syllables

```python
alist[i] = min(alist[i], alist[i+1]) #13
alist[i+1] = max(alist[i], alist[i+1]) #15
```

By introducing an additional variable and initializing this variable to 0, the lines will consist of the same number of syllables:

#### Listing 5 – An addition line followed by two lines with a the same number of syllables

```python
x = 0 #4
alist[i+x] = min(alist[i], alist[i+1]) #15
alist[i+1] = max(alist[i], alist[i+1]) #15
```

Both exercises together lead to some methods or algorithms with a nice metrum and rhyme.

### 2.2. Experiences

#### 2.2.1. Rhyme

This first lesson of the Code as Art, Art as Code course was ran at a conference for professional software developers in Norway, with about 20 attendees. In the first exercise, concerning rhyme, participants made interesting choices here and there. Some people annotated the sounds of lines of code quite literally, like this:

#### Listing 6 – Literal sounds for lines of code

```python
def bubbleSort(alist): #list
    for passnum in range(len(alist)-1,0,-1): #one
        for i in range(passnum): #num
            if alist[i]>alist[i+1]: #one
                some = alist[i] #i
```
Others however pronounced list operations with the list name at the end, as seen in Listing 7. Appending an item \( x \) to a list \( \text{less} \) could be pronounced like ‘add \( x \) to \( \text{less} \)’ putting the list name at the end.

\[
\text{alist}[i] = \text{alist}[i+1] \quad \# \text{one}
\]
\[
\text{alist}[i+1] = \text{some} \quad \# \text{some}
\]

**Listing 7 – Comments rhymes with code**

```python
if x < piv:
    \text{less}.append(x) \quad \# \text{less}
if x == piv:
    \text{equal}.append(x) \quad \# \text{equal}
if x > piv:
    \text{greater}.append(x) \quad \# \text{greater}
```

When manipulating rhyme, some added comments to rhyme with lines of code, like this:

**Listing 8 – Comments adding to rhyme with lines of code**

```python
if(you <= 2) { // you
    return 1; // one
} // done?; // one
```

In this case, the comment even has meaning in the algorithm, in addition to the right sound, as these lines of code check whether the recursion is finished, i.e. ‘done’. Some people let themselves be inspired by keywords, for example naming a variable ‘strength’ to rhyme with length:

**Listing 9 – Variable names selected to rhyme with code**

```python
\text{int} \ \text{len} = \text{data}.\text{length}; \quad \# \text{length}
\text{int} \ \text{loop, i = 0;} \quad \# \text{zero}
\text{key} = \text{data}[\text{strength}]; \quad \# \text{strength}
```

### 2.2.2. Rhythm

In the second exercise, participants focus on the rhythm of code. In the first exercise, they just observed the metrum of a line of code. This seems like a futile exercise, but it revealed some interesting truths about source code. For example, how many syllables are in this line?

**Listing 10 – How many syllables are in this line of code?**

```python
\text{int} \ x = 5
```

\text{int}, \ x \text{ and } \text{5} \text{ clearly all contain just one syllable. But what about } \text{=}? \text{ It that one, since it is } \text{	extbackslash{=}S}\text{? Or two, since it is not really an } \text{	extbackslash{=}S\text{ but more of an } \text{v-	extbackslash{=}S\text{? We found a number of interesting choices. Some preferred the simple } \text{	extbackslash{=}S\text{, others choose to vocalise this as } \text{v-	extbackslash{=}comes}\text{, as the variable takes on the value of the following expression, while a third group thought } \text{	extbackslash{=}S\text{ is more correctly representing what it means. In any case, this lead to some interesting discussions among the participants.}}}}}}}

Another interesting fact was revealed when participants started to play with the numbers of syllables. That made them reflect on where variables were used, since a change of a variable name would impact the number of syllables of only the lines in which they were used.

All in all this course was received as very interesting and insightful by participants, since developers usually do not view source code in a vocal and poetic way.
3. Lesson 2: Code as Art: Paintings

The second art form explored in the Code as Art course is visual art. Programming as it is now is, in a sense, like realistic art, it tries to represent the world. This holds especially for programming education, in which often the real world is taught through modeling real world objects. What happens if we let this go? The metaphor of code as a painting is a rich one. Paintings can be beautiful or intentionally ugly. Modern art forms especially played with the notion of art itself, for example with Warhol creating Brillo boxes, an everyday utensils as an artwork. Could we do the same for code? Can we create code that is not representing the world in a realistic way? Can we envision cubistic or expressionistic source code? Can source code be liberated from the tyranny of usefulness or even executability?

3.1. Setup

In this second lesson of Code as Art, the goal is to view code as an artistic expression. Participants received a random painting or artwork from a preselection we made: cubism, surrealism, graffiti, De Stijl, art nouveau, expressionism, dadaism and rococo. Participants were then asked to work in pairs, and to select an sorting algorithm and recreate that in the style of their painting. The selection as well as the implementation of the algorithm could be inspired by the art style. For example insertion sort might fit a more realistic style painting as it is how people usually sort cards. Inspired by a certain brand of realism, this algorithm could be further adapted.

3.2. Experiences

This lesson was ran twice, each time with around 10 computer scientists and programmers. It was interesting to see what participants came up with. Some participants really were inspired by the philosophy of the art style. For example, one group received Dada, and created a sorting algorithm from random lines of code, since in Dada, there were no rules. Another group, receiving Rococo, placed it in the history of art after Barok, attempting to be simpler than what was before, but also still more involved than art that came later. They therefore selected shell sort, since it is more efficient than insertion sort, but not by a lot. Another Rococo group observed that Rococo is not efficient at all and thus devised a very inefficient algorithm based on bin tree sorting.

However, not all participants had a deep knowledge of art history, and this resulted in some groups to focus on the artwork we used as illustration of the concept, rather than on the idea of the art style itself. For example, the de Stijl painting we used was Victory Boogie Woogie by Piet Mondriaan (Figure 2) in which the canvas is divided in blocks. This lead participants to use Radex sort, in which the input list is divided into groups as well. While this of course is a fine interpretation, we would have prefer a more high level interpretation of the art works like the Dada and the Rococo groups desired above did, or maybe even more abstract.

For example the idea that surrealism aimed to free art from the dogma of the bourgeoisie and to allow more people to be artists. It pushed the boundaries of what art is and is not, and we are interested in hearing what sorting (or programming) is or is not.

A lesson we took from this for subsequent lessons is that programmers or computer scientists may have too little knowledge of art forms, and that courses in the future should be preceded with an introduction to art. A more viable approach might be to select one art style, explain that in more detail including its history, and then having all participants create an algorithm in that style.

---

2Yes, we would love to do this at PPIG 2017!
Some participants however did use the idea of the art style as an inspiration, for example, one group that received rococo made the observation that rococo is the opposite of efficient, it uses a lot of fluff to deliver a message. They therefore created a sorting algorithm as inefficient as possible.

4. Lesson 3: Art as Code: Poetry
4.1. Setup
In the poetry course, we start with the generation of simple sentences. For this, only basic programming concepts such as lists and random numbers are needed. Figure 12 for example generates a noun and a random verb. These together can form a sentence by using two nouns with a verb in between, for example ‘Polly desires cracker’. Note that there are words that will not lead to correct sentences, like ‘must’, given we create noun-verb-noun sentences.

```
function getRandomNoun(){
    var nouns = [ 'apple' , 'tree' , 'laptop' , 'Carl' , 'Frances' , 'house' ];
    var index = Math.floor(Math.random() * nouns.length)
    return nouns[index];
}

function getRandomVerb(){
    var verbs = [ 'eats' , 'wants' , 'loves' , 'desires' , 'must' , 'kills' ];
    var index = Math.floor(Math.random() * verbs.length)
    return verbs[index];
}
```

Participants are then encouraged to change or add words and to play with the order to create better (or worse!) sentences. In a second exercise, conjugations are added to make more complex sentences by adding other word types like conjugations.

```
function getRandomConjugation(){
    var conjugations = [ 'because' , 'unless' , 'as' , 'and' , 'but' , 'since' ];
    var index = Math.floor(Math.random() * conjugations.length)
    return conjugations[index];
}
```

This resulted in funny, crazy or weird sentences, like “Carl must house since house eats tree”.

---

Figure 2 – Victory Boogie Woogie - Piet Mondriaan
4.2. Experiences
We ran this course two times. One with about 40 children aged 8 to 10. These children had been following our programming lessons for 4 weeks prior, so they had some experience with programming in Scratch. For that group we used Scratch. Secondly we ran it with about 100 teenagers in groups of 20, aged twelve to seventeen at a science festival for schools. Most of these teenagers had no prior programming experience, although some of them had used Scratch or Lego Mindstorms, or named Minecraft as programming experience.

Overall, we observed that children of both groups were very engaged with this exercise. Over 80% of the children in our Scratch named this lesson as their favorite of all Scratch lessons. The other lessons being more traditional Scratch lessons in which they created games and animations. The teenagers too loved the lesson, it seemed especially those that previously thought that they would not like programming.

There were interesting differences in approaches among both groups. Some children of both ages took the first sentence they got and started creating a poem of story with that. Other wanted to ‘tweak’ the programming, and added more words until it gave them a sentence they liked, without playing with the structure. A third group wanted to add more word types and sentences, learning quite some things about grammar too in the mean time.

We believe that the fact that this course supports many different paths to a success experience makes it an interesting first introduction to programming.

5. Lesson 4: Art as Code: Paintings
In the fourth lesson of the course, code is used to create Mondriaan artworks. This is another way of demonstrating to novice programmers the wide range of applications for programming. Mondriaan art is a nice topic, because it is relatively easy to create with source code but is still recognizable for children as ‘real art’.

5.1. Setup
For this course the setting is a big lecture hall, where an instructor creates one program with input of the participants. We gave all children a green and red card which they could use to vote on questions like ‘is this program correct?’ or ‘do you think this is the best way to program this’. With these questions as input, the presenter created one program while explaining the steps taken, in a form of observational learning, where a teacher demonstrates a task before learners attempt it (?, ?).

The course starts by asking children to draw a Mondriaan painting. Most children in the Netherlands are familiar with the work of Mondriaan and can create this with relative ease. In many schools around the country this is a common drawing exercise, especially in 2017, the Mondriaan year^{3}.

When they have drawn their Mondriaan, the next step is to compare their drawing to those of a neighbor and ask for one commonality and one difference. These factors will be input into our program. For example, if the two paintings differ in color, that should be a setting on the resulting program.

When the participants have created and compared their paintings, the programming starts. We first simply create a program that draws a line, and then a colored area too, as shown in Figure 3.

---
^{3}http://www.destijlutrechtamersfoort.nl/en
This code is already a bit ‘smelly’, and the course leader can ask the audience if they agree, which they do in our experience. The code can then be put into functions for lines and areas, making it easier to generalize. A nice technique for this is to first make the code a bit worse, by creating a second line, as follows:

The two pieces of code can now be placed side by side, as an illustration of their similarities and differences:
It is fun to have the participants perform the same exercise on code as they did on their paintings, further stressing the fact that indeed code is like art and art is like code. After putting the line and the area into blocks, we brainstorm with the room on what variability is needed to create as many Mondriaans as possible. For example, what colors are needed, how many lines do the paintings have, and what should their direction be. In one of the courses a parent drew Victory Boogie Boogie (Figure 2), seriously complicating things.

While this course could also be done by participants individually, or in pairs, we opted for a plenary form, since it introduces high level concepts like functions which children aged 10 to 12 are usually not familiar with.

5.2. Experiences

This lesson two was ran two times, both times with about 70 children, once aged 11 to 12 with parents present and once with children aged 8 to 11 without parents. The lecture took a little but over an hour in both cases. We used the plenary method of teaching, starting with drawing their one Mondriaan and comparing them to the shared solution.

One observation is that some children were disengaged and even annoyed initially by the fact that they had to start by creating a drawing. “Why do we have to do this?” “This is supposed to be about programming!” We promised them this was useful practice for programming skills, and they sort of went with it although some kept compacting a little bit. This seems proof that programming and making things on the computer are already very much tied to each other in young children’s brains. Interestingly enough, this was more common in the younger group without the parents.

When we pointed out later in the exercise that the drawing and the comparison with the drawing of neighboring children had been useful practice for creating the blocks, it made sense to many children.

At the end of the lecture, we succeeded in creating a Mondriaan painting, and, at the suggestion of one of the children, we even added some randomness to it to create new paintings.

After the lecture, we asked children after the lecture what stuck with them most, and the answers generally fell into two categories. One group of children stayed that the biggest take away was the idea that you can create something like any with programming. Other children were impressed by the notion of abstraction and the ability to create custom blocks.

6. Related work

Papert, one of the founders of programming education for children already argued programming is a creative endeavor (Papert, 1980). Papert states that the unique educational power of a computer is to support children in creating, exploring and experimenting, aspects of learning that are ignored in traditional education. While this work is almost forty years old, it seems to still be representative of modern day teaching, where most focus is on learning facts and techniques and not on learning though creating. Kafai and Peppler (Kafai & Peppler, 2012) described that games are used in education, but mainly to support traditional learning while they argue that game design can also contribute to creative and critical thinking in children.
More and more art shows and exhibitions are showing digital art, from computer generated paintings, to digital installations using projections. Peppler and Kafai note that professional artists are using technology increasingly, while children in art class rarely explore programming as a means of creating art (Kafai & Peppler, 2009). Romeike performed a literature review of introductory programming and found that creativity is rarely used in programming education, despite some authors describing promising results (Romeike, 2007).

Barnard argues that teaching programming and art together matters, since the process of creating an artwork is so similar to creating a program. Like the programming, an artist has a dialog with their work, which can only start when the real work has started (Barnard, 2015). In her thesis, van Groenestijn (Groenestijn, 2016) describes a method to integrate teaching programming and creativity, for example by having children create a digital tour of their school.

7. Concluding remarks
The world of programming is filled with programmers bringing their own ideas of what programming is. The ruling opinion seems to be that programs should have a purpose, such as apps, games and websites. We found that children as young as 10 already clearly have this bias, but many professional developers have too since they develop useful software all the time. To that end we have developed a course series Code as Art - Art as Code consisting of four lessons in two parts. In Code as Art, participants view source code as poems and paintings, observing the rhythm, rhyme and structure of lines of code. This part of the series was ran with professional developers and computer scientists, and we observed that they enjoyed a fresh way of looking at source code, and making decisions on how sounds of operators was even insightful. In Art as Code, participants use source code to create poems and paintings, so they learn to see programming as a broad tool, but we found that they also learned about language and abstraction in the lessons. In this paper we described sessions of Code as Art with professionals and Art as Code with children, however we think that the reverse should also be possible. Code as Art with professionals is no problem at all, but for ‘Code as Art’ some knowledge of programming is needed. One future research direction we envision is to explore how much programming experience is needed for this. Maybe starting with reading source code aloud is a nice way to start programming education, even if the source code makes no sense yet? After all, children also start to draw some letters before they can read.

In any case, we plan to continue this way of teaching to children as young as possible, so we can educate a new generation of creative programmers.

8. Acknowledgements
Thanks to the hundreds of participants that were willing to try out crazy things with me, including the Rainbow Group in Cambridge. A special shout out to the organizers of Boosterconf in Norway, who were the first ones to let me run Code as Art - Art as Code at a developers event.

9. References


Methods in user oriented design of programming languages

Clayton Lewis  
Department of Computer Science  
University of Colorado, Boulder  
Fellow, Hanse-Wissenschaftskolleg  
clayton.lewis@colorado.edu

Abstract

Card and Newell, in an influential 1985 paper, described programming languages as "obviously symmetrical" between programmer and computer, and called for balanced investment on the programmer and computer sides of the design space. But the design space is in fact more complex than that, with important impacts of purpose, as well as of programmer and computer, on effectiveness. Further, each part of the space is fragmented into many distinct areas, reflecting consequential differences in people, purposes and computational setting. Empirical methods face challenges in spaces of this complexity. As suggested by research on the role of mechanisms in scientific thought, cognitive dimensions analysis is better suited to operate in this complex space than are empirical methods, and should be promoted and extended.

1. Introduction

Millions for compilers but hardly a penny for understanding human programming language use. Now, programming languages are obviously symmetrical, the computer on one side, the programmer on the other. In an appropriate science of computer languages, one would expect that half the effort would be on the computer side, understanding how to translate the languages into executable form, and half on the human side, understanding how to design languages that are easy or productive to use. ... The human and computer parts of programming languages have developed in radical asymmetry.

--Newell and Card, 1985

I've given talks for years built around the above quotation from Newell and Card, calling for greatly increased attention to user oriented design of programming languages (UODPL). But recently I've come to think that Newell and Card's framing of the issue needs substantial attention. The opportunity as they suggest it is too simple: all we have to do is pay more attention to the human side of the picture. The actual situation is much more complex, and, accordingly, the response needed from us isn't simple, either. Thinking through these matters, I will argue, provides strong support for the continued development of lines of work identified strongly with PPIG as an institution, and indeed for the defense of this intellectual approach against seemingly attractive alternatives.

The original impetus for this rethink came a few years ago, when I led an undergraduate seminar on UODPL. This would be easy, I thought, with the symmetry of Card and Newell's diagnosis as a starting point. As we reviewed and discussed the literature, however, I felt that work didn't fall into line in the way I expected, and that the picture became murkier, not clearer, as we progressed. Why?

The urge to do more with this question rose above threshold during the panel session, 'PPIG in the wild - what should we be studying?’, at PPIG 2016. There two panelists, Meredydd Luff and Steven Clarke, presented sharply differing accounts of trying to do real UODPL work, on an academic project and in industrial development work respectively. Luff found no guidance in the PPIG literature, or elsewhere in the literature, because the seemingly relevant quantitative results were inconsistent, and thus incoherent. In particular, different measures of programmer effort were uncorrelated. Clarke, on the other hand, presented a gratifying story of real impact on the thought and work of developers, based on a combination of videos of actual programmer experience, and cognitive dimensions analysis. I'm suggesting here that it's the
complex nature of the UODPL landscape that explains why the literature is so unsatisfying, and why Luff did not find what he was looking for, on the negative side, and (on the positive side) why Clarke has been so successful.

Based on this analysis I'll argue that some approaches to our subject will be more successful than others. In particular, those of cognitive dimensions style will be impactful, but generalizations based on quantitative data will be much less so, despite frequent calls for this particular kind of scientific work.

2. The complex landscape of programming language design

As suggested by the sketch of the language design landscape in Figure 1, Newell and Card identify two abutting regions, the programmer side and computer side. The computer side has been studied a lot, and the programmer side, not much. We need to get to work, and it's clear where.

![Figure 1: The design landscape in Newell and Card (1985)](image)

Figure 2, below, shows the landscape as I now believe it to be. There are lots of differences.

One big change is the addition of a whole new region, that of "purposes". How well a programming language works is determined not just by how it is implemented, and who is using it, but crucially what it is being used for. This isn't news: we've thought since early days that FORTRAN is a better tool than COBOL for "scientific" work. But when we think about what "programming" is, we often lose sight of the differences. Perhaps talk of "general purpose" languages helps us forget. But surely we can accept that there aren't actually any "general purpose" languages, and never could be. Is Java "general purpose"? Would any sane person make a cat character dance and emit thought balloons in Java rather than in Scratch?

Another change is that all the regions are subdivided. The new "purposes" region is carved up into different kinds of target domains, of which I've suggested just a few in the figure. Many domains require mathematical reasoning of various kinds, including those calculations rooted in physics for which FORTRAN was designed. Others are mathematical, but not in the same way; FORTRAN isn't as well suited to logic or graph theory. A vast range of domains are hardly mathematical at all. The record structures that originated in commercial data processing in the assembler era, and gradually came into higher level languages, provide basic representational tools in a very wide range of important applications, but these are barely mathematical. Some target domains involve dynamics in essential ways, while many do not; concepts of timing, or of synchronization may or may not be needed, and be available. Some domains involve interaction, requiring another, different suite of concepts, and so on.

Someone wanting to understand what will go well or not so well for a Scratch programmer (to take just one example) can't overlook these variations in the landscape. Scratch provides quite direct support for scattered aspects of these domains, and little or none for others. The same is true for any other language one might name.
Cutting across these target domains is an additional set of contrasts, that might be called "contextual requirements". Someone creating an antilock braking system, or a defibrillator, must deliver an extremely strong correctness guarantee, while someone making a cat character dance need not. Thus in the former case it will be rational to require quite laborious conceptual work, if in return the needed strong correctness guarantee can be obtained; in the latter case it's silly and counterproductive.

Contextual requirements are themselves diverse. Some programs have to run very fast, others need not; some must interface with or exploit stipulated hardware (GPU, sensors, ...), or not; must interface with stipulated software; must be readily reconfigurable to work with new devices or programs; must be maintained so as to operate for many years, or not; the situation being addressed requires millions of lines of code, or just a few; and so on and on and on. Just as for correctness guarantees, any of these requirements will change the rational assessment of programming language features. Complications that can and should (must?) be avoided in some situations can't be avoided in others.

Not shown in the figure is another set of purpose distinctions. Are we observing someone thinking about a program, and trying to make a plan, or someone who has a plan and is writing code, or someone who has code and is trying to debug it, or someone who has someone else's code and is trying to debug that, or someone who has code and is trying to assess its correctness empirically, or someone who has code and is trying to prove its correctness, or someone who has code and is trying to extend it, or ..., or .... All of these distinctions are consequential, in that an aspect of a programming system can be good or bad for one kind of purpose, and bad, indifferent, or good for another.

The “programmer” region is crisscrossed with distinctions, too. Is the programmer a child? A systems programmer? A pensioner? A scientist? Do they like to learn by tinkering, or step by step? Do they like to understand what they are doing, conceptually, or would they be happier not to? Other differences, not
shown, cut across these. Is the programmer working alone, or in a group? Are they knowledgeable about the mathematical aspects of the situation with which they are engaged, or not? Do they have years of experience with Python? It's obvious that any observations of programming by people drawn from this pool will be extraordinarily variable, and that deriving lessons from what is observed in one part of the pool about what might happen in another part will not be easy.

The computer side has its own subdivisions, only suggested in the figure. The challenges on phones are different from those on clusters. Further, we know, but often forget, that development environments and tools condition the effectiveness of language features. So does the language paradigm within which language features function, not shown. Imperative programming is not all of programming. Do we think that learner difficulties we see in imperative programming are a guide for what we would see in functional programming? Reactive functional programming? Logic programming? In object orientation, what is the impact of multiple inheritance? How about spreadsheet programming?

These complexities can be illustrated by a controversy of the past. In the 1970s two styles of electronic calculators were popular. Texas Instruments calculators used "algebraic notation", in which complex calculations were described using a notation similar to that used in written mathematical formulae. Hewlett Packard calculators used reverse Polish notation (RPN), in which arguments to an operation are entered first, followed by the operator. Which notation is better? Surely one must be better, and people argued about which that was.

The empirical results of user speed tests tended to favor RPN, or to show no difference (Card, 1979; Kasprzyk et al. 1979; Agate & Drury, 1980; Hoffman et al., 1994). The size of the difference depended on the details of the tasks, but never showed a clear advantage for algebraic notation. Yet RPN calculators have all but disappeared in the decades since these evaluations were done. What did they miss? The evaluations all used skilled users, often engineering students. For people generally, the familiarity of evaluating 2+2 that way, rather than as 2 2+, seems to carry the day. So, which notation is better? It depends on whether performance by skilled users is important, or familiarity in casual use.

The impact of the complexities in the present has been described by insightfully by Luff himself. In his paper, “Empirically Investigating Parallel Programming Paradigms: A Null Result” (Luff, 2009) he lists several obstacles to the empirical comparison of language features. Besides weak metrics for programmer effort, he cites the effects of learning, where test participants learn things from an earlier task that affects their performance on a later one; the effects of an old design being more familiar than a new one with which it is being compared; toy problem syndrome, the problem of test tasks being smaller and simpler than real ones; the variability among different possible test tasks; the difference between ease of writing and ease of maintaining code (that is, contrasting purposes); student test participants not being representative of real users; and others.

3. Methodological implications

3.1 Measurement considered hopeless.

If we consider a seemingly clear and simple notion, "programmer effort", against the tessellated landscape of purposes, people, and computational structures we can see that it is hopeless as a target for quantitative measurement. Thus Luff's frustration with available metrics is just what we expect. The question, "How much programmer effort would using feature F require?", or even "Would using feature F require more programmer effort than feature F?" are both in a rich class of questions in HCI, those that have no answer as posed, however clear, concrete, and practical they seem.

The type specimen of questions without answers in HCI is, "Tell me what colors I should use for these controls?" Are your users members of the public, about 5% of whom will have anomalous color vision, or military pilots who are required to have typical color vision? Do they include Chinese people, for whom red is the color of happiness, and white the color of death? Are there similar controls already deployed elsewhere in this system? How many and what controls need to be discriminated? and so on. As we'll see,
and as many in the PPIG community have seen, there is a way to handle the design needs that lie behind questions like these, but it isn't a metric.

One's first instinct might be to deal with the tessellation problem by what we might call special purpose sampling. If programmer effort depends on many different factors, of audience, purpose, and computational setting, let's roll up our sleeves, determine what levels of all those factors are the ones we care about, and collect some measurements in just a small corner of the space. This is almost never workable, for multiple reasons.

First, measurement is a lot of work. Even within small categories (of people, for example) variability is often high, so a lot of data are needed for useful estimation. Only rarely will people judge that the impact of a small design feature warrants the measurement cost.

Second, even if cost is no object, the measurements will often be virtually worthless. The situations of practical interest will almost always take in different levels of our various factors. Perhaps we are interested only in 14 year olds, but will all the 14 year olds know nothing at all about programming? Will all their programs be interactive, or not? Will they be working individually, or will some work in pairs? Will some work at home?

Here we have to worry not only about these several levels, but about their interactions. That is, we can't measure the impact of student age and knowledge, and then separately the impact of the setting in which they work. The impact of work setting may well be different for older students than for younger ones, and if measurement is our approach, we can't tell that without taking measurements for all the combinations. Given the complexity of the space of situations we are seeking to understand this is combinatorially hopeless.

All this is bad enough for us, but it is worse for a designer looking for guidance. Even if somehow we expend the resources to do a really thorough survey of the part of the landscape we care about, when a designer comes along and reads our report, will it shed any light at all on the problems they are working on? Very likely not. If they are interested in professional programmers, or university students, not 14 year olds, they're out of luck.

The situation is actually even darker than that. The designer wants to know the effort associated with a new language feature that they have just invented, one we could not possibly have collected data on. We have to accept that measurement can't tell us what we (and the designer) need to know.

3.2 The logic of cognitive dimensions

Why is Clarke happier than Luff? Because Clarke did not rely on measurement at all, but on a combination of cognitive dimensions analysis and observation. Let's focus first on cognitive dimensions.

Cognitive dimensions (see, among many papers, with many contributors, Green, 1989; Green and Blackwell, 1998) work by describing aspects of a situation that are likely associated with difficulties (and hence with errors, effort, and other costs.) For example, a hidden dependency is an aspect of a situation that will cause trouble when someone has to understand the consequences of some modification they might undertake, such as changing the declaration of a variable. A programmer might not see exactly what uses of that variable will be affected by the change; a programming environment might (or might not) provide information that makes the dependency more (or less) apparent.

Note that there is nothing in the definition of a cognitive dimension that is particular to audience, purpose, or computational setting. Hidden dependencies will affect children and scientists, high performing and low performing programs, and imperative or functional languages in the same way. It makes sense for a designer to examine their completely novel design, to see whether their users are likely to encounter issues of this kind. Even better, the logic of hidden dependencies includes within it guidance on remediating it. Thus a designer may learn not just that their design has a likely problem, but also why the problem occurs, and thus how it might be avoided.
Clarke's experience demonstrates that the sophisticated developers with whom he works find the logic of cognitive dimensions clear and compelling. They not only recognize the force of design critiques based on them, but also make such critiques themselves, and see the value of responding to them, resulting in improvements that are meaningful to them.

On the other hand, Clarke's presentation showed that his collaborators did not respond well when he gave them data showing low success rates by their users. But aren't data the gold standard? No.

In my view Clarke's coworkers were actually right to be resistant. As we've seen, the sample could hardly be representative, and worse, the data include no information useful for responding to the problems they suggest.

3.3 The role of observation

Clarke got started with videos that showed examples of people having the kind of difficulties that cognitive dimensions identify. Why was this important? I suggest there are two reasons. First, someone new to cognitive dimensions might well not see that the relatively abstract descriptions that Cognitive Dimensions provides can actually be applied in their situation. The videos show that they can, and can give an assurance that they are useful. Second (as this essay argues throughout) programming is an extremely complex activity. Even an experienced cognitive dimensions analyst can get things wrong, by not seeing issues that users actually encounter, or not seeing how users can in fact avoid an apparent pitfall. Observing actual user behavior provides a check on what the cognitive dimensions analysis suggests.

Does the need to make observations of users plunge us back into the combinatorial tar pit of different users, purposes, and so on? Don't we have to observe many different kinds of people, doing many different things? A little, but only a little. Let's take two cases. First, suppose we observe a few people, and there are no surprises. We can't be sure that there aren't problems lurking somewhere out there, for a user with some unforeseen misconception, or trying something we hadn't thought of. But we can have practical confidence that in the normal case things will be more or less OK. Why? Because we have formed a theory of what will happen in the normal case, and we don't see any reason to think that theory is wrong.

We have no ultimate assurance, of course; we never could. Measurement wouldn't provide it, either.

Second, suppose we make some observations, and there is a surprise. People have trouble that we didn't expect, or perhaps a trouble that we anticipated doesn't materialize. Either way, here we have learned that our theory is wrong. If the issue is consequential, we need to fix the theory, or modify the system to steer around the issue. Note that in neither case will it help us to blanket the space with sampled users and tasks, in the teeth of the combinatorics. We might observe a few more people, hoping to see more examples that can help us to understand what's happening, but surveying that huge space won't help with that.

3.4 The situation in science generally

One view of science is that is it, or ideally should be, a parade of randomized controlled trials. For some, only such experiments produce reliable knowledge; advocates of "evidence based" approaches in medicine and education promote this idea (see for example National Center for Education Evaluation and Regional Assistance, 2003). But studies of actual scientific practice give a picture more compatible with cognitive dimensions logic.

Darden and colleagues (Machamer, Darden, & Craver, 2000) have documented the pervasiveness of mechanisms in scientific work and thought, especially in biology. Mechanisms explain phenomena of interest by describing the entities, relationships, and activities that fill out a narrative of how a phenomenon is produced. Darden (2002) also identifies mechanism schemata: "A mechanism schema is a truncated abstract description of a mechanism that can be filled with more specific descriptions of component entities and activities" (p. S356). Darden describes how these schemata provide a structure for scientists seeking to understand a new phenomenon. "Once a schema is chosen or sketched in a discovery episode, then the task
is to find the entities and activities, or modular groups of them, that play the roles outlined in the abstract schema. A schema has placeholders, variables, black boxes, that may be filled piecemeal as empirical evidence is found for the various components. The lack of an entity or activity or module to fill a role in a schema points to the need for further work” (p. S360).

Note that the aim of the scientists’ work is not to show that something is true, but to explain why it is true. The explanation is provided as a description of the mechanism that produces the phenomenon, or at least a candidate mechanism that could produce it.

Russ (2006), building on the work of Darden and colleagues, argues that mechanistic reasoning is more effective than formal empirical investigation: "[I]t is by knowing what mechanisms act in the systems that scientists pare down the number of variables for further rigorous study. If scientists or students relied only on covariation data, they would be working forever testing infinite numbers of variables only a few of which would actually be important in the situation” (p. 43).

One might respond to these suggestions by pointing out that supporting language design choices is a different activity from developing scientific understanding. Surely one can decide whether A is better than B without having a scientific understanding of either, for which mechanisms would indeed be important. Here Russ provides two further relevant arguments.

First, Russ points out that mechanisms are important in distinguishing relevant correlations from irrelevant, drawing on Koslowski (1996). For example, lots of factors are correlated with crime in cities, including number of churches. Everyone recognizes that these correlations are not causal, just because there isn't a mechanism connecting the factors.

This might not seem relevant to language design, which doesn't much concern itself with correlations and their interpretation. But consider an A-B test of two language designs, that shows an advantage for A over B. Consideration of mechanisms might suggest that the observed advantage of A is due to some particular feature of the test task that would not be encountered often in practice. Similarly, consideration of mechanisms might suggest that had the test users possessed, or not possessed, some particular knowledge, B would have been better. (For an example of the impact of user knowledge on performance, also including interactions with task types, see Halasz and Moran, 1983.) The apparent clarity of the empirical test results would not stand up to mechanism-based critiques like these.

Finally, Russ argues that "Mechanistic reasoning is more helpful than formal empirical investigations for understanding novel situations"(p. 47). Her argument is parallel with our discussion of the difficulties posed by the tessellated design space, above:

"When faced with new physical situations, it is unlikely that students will already have a store of covariation information collected under the exact same conditions with which to draw conclusions. Covariation information is narrow; it only gives insight into the precise case that generated it. Establishing that a particular variable is associated with a particular result in a very particular set of circumstances (a controlled experiment) does not help predict what will happen when that same variable is found in a different set of circumstances. Only in knowing the properties of a variable and the process by which a cause brings about an effect – the mechanism - can we know what that variable may or may not do in another situation" (p. 47).

Here again, Russ's focus on covariation shouldn't prevent us from recognizing the relevance of this conclusion for design work. Knowing that A outperforms B under particular test conditions can't give us confidence that it will do so under novel conditions, if we don't understand the mechanisms involved. Further, knowing that A outperforms B doesn't allow us to predict whether or not a novel design A' will outperform B, if we don't understand the mechanisms. As Landauer and Galotti (1984) say, commenting on confused empirical findings on command language design, "For completely practical purposes, only … theoretical understanding will make it possible to make good design decisions about new systems in advance” (p. 428).
Cognitive dimensions analysis can be seen as supplying a collection of mechanism schemata for explaining phenomena that occur when people use representations, in activities like programming. Applying cognitive dimensions analysis to a problem in designing a programming language has the same status as a biologist applying a mechanism schema to a problem in molecular biology. It has the same virtue of providing insight into why a design feature works or fails to work. It helps deal with the problems Russ identifies in formal empirical work.

Three other areas of thought bear on these matters, and can be mentioned briefly. First, some work in machine learning, following insights by Schank et al. (1986), has favored explanation-based generalization over similarity-based generalization (see e.g. Mitchell et al., 1986; Lewis, 1988), especially in situations in which generalizations from very limited data are required (see also Jones et al., 2006, for this distinction in psychological studies of generalization.) As argued earlier, language designers will very often be seeking to generalize in data-poor circumstances, for reasons that include novelty and contextual complexity. Second, workers in evidence-based medicine have recognized the difference between a prediction for a population and a prediction for an individual patient: “It may be hazardous to presume that the point estimate of risk derived from a population model represents the most accurate estimate for a given patient. … [D]irect measurement of subclinical disease (screening) affords far greater certainty regarding the personalized treatment of patients, whereas risk estimates often remain uncertain for patients. In conclusion, shifting our focus from prediction of events to detection of disease could improve personalized decision-making and outcomes.” (McEvoy et al., 2014). Similarly, someone designing software for a very large audience, and able to collect performance data on a very wide scale, might feel that population-level A-B trials would give adequate design guidance. But the resulting design might prove to be quite bad for many particular users, in particular situations. Finally, in general HCI, claims analysis (Carroll and Kellogg, 1989; McCrickard, 2012) shares features with cognitive dimensions logic; see also Haynes et al. (2009).

4. Implications for PPIG
The PPIG community should be proud that cognitive dimensions analysis emerged from the work of people in its ranks, Thomas Green, Marian Petre, Alan Blackwell, and others. We should be skeptical of calls to replace its use with A-B trials or other quantitative methods that cannot cope with the complexity of the language design landscape. When results of A-B trials and similar studies are presented, we should diplomatically ask for the mechanisms that are involved to be described. Colleagues who present the results of such trials should be prepared to respond to this request, so that the generalizability of their results can be assessed. Thinking aloud studies, such as were used effectively by Halasz and Moran in their calculator work, or other ways of observing the activities of individual users in detail, as in Steven Clarke’s videos, can be a good supplementary method for investigators who are doing empirical comparisons. Indeed, adding a cognitive dimensions analysis to an empirical comparison would also increase generalizability.

We should help our colleagues understand the value of mechanism-based analysis, given that for many it doesn't seem "scientific" or "technical" in ways they can readily recognize (see e.g. Moody, 2009). Studies that combine cognitive dimensions analysis with user observations, as was done so effectively by Clarke, could be a good way to do this.

Another, quite different, way could be to explore the relationship between cognitive dimensions and the state of the art in cognitive modeling, as represented by the work of the ACT-R and SOAR communities (http://act-r.psy.cmu.edu/, http://soar.eecs.umich.edu/). This could be seen as a way to clarify the role of cognitive dimensions as mechanism schemata, with roles to be filled by specific cognitive mechanisms described at a lower level. This could enhance their explanatory power, in line with Darden's description of scientific progress. See the related recommendations in Church and Mărășoiu, given at PPIG 2016.

Finally, we should seek to expand the repertoire of dimensions, even to include dimensions not properly cognitive. The tripartite picture of the design landscape shows not only people and machines but also purposes. We may be able to identify mechanism schemata that help people design for particular purposes, for example maintainability. One would suppose that failures of maintainability result from the action of
mechanisms that could be identified. The discussion by Basman et al. (2016) at PPIG 2016 perhaps suggests what might be done. While such an effort would draw the Psychology of Programming Interest Group a little beyond the properly psychological, it could place PPIG’s psychological work in a setting in which its value is more apparent to a wider audience.

5. Acknowledgements
I thank the Hanse-Wissenschaftskolleg, Delmenhorst, Germany, for support while preparing this essay. Emily Moore, Luke Church, Antranig Basman, and the students in CSCI 4830 in spring, 2014, provided useful ideas for which I am grateful.

6. References


A Systematic Literature Review of Cognitive Dimensions

Mohammad Hadhrawi  
Computer Laboratory  
University of Cambridge  
mkah2@cam.ac.uk

Alan F. Blackwell  
Computer Laboratory  
University of Cambridge  
alan.blackwell@cl.cam.ac.uk

Luke Church  
Computer Laboratory  
University of Cambridge  
luke@church.name

Abstract

We report a Systematic Literature Review (SLR), exploring the ways that the Cognitive Dimensions of Notations (CDs) framework has been applied since being proposed in 1989. We analyse over 1,600 publications that have cited key references in the CDs literature. Our research questions include 1) whether CDs are used as formative discussion tools or for summative evaluation; 2) which elements of the framework are most widely applied; and 3) the balance between applications and theoretical research contributions.

1. Introduction

Systematic Literature Review (SLR) is a research method introduced in the Software Engineering community, advocated as a rigorous process for meta-analysis of research publications in this field and others (Kitchenham, 2004; Kitchenham et al., 2009, 2010). SLR emphasises the application of quantitative hypothesis-testing procedures to research literature, including: formulation of prior hypotheses; use of replicable search procedures for identifying relevant publications; objective criteria for inclusion (or exclusion) of items in a study corpus; and assessment of hypotheses in a numerical manner (e.g. reporting frequency with which a particular kind of statement is found in the corpus).

In this paper we use SLR to investigate ways in which the Cognitive Dimensions of Notations (CDs) framework has been applied. CDs was first introduced by Green (Green, 1989), then rigorously defined and greatly popularised by Green and Petre (Green & Petre, 1996). Their intention was to create an analytical framework that would allow the usability of notational systems to be compared in a ‘broad brush’ manner, informed by the body of evidence from earlier empirical studies of programmers, and avoiding the ‘death by detail’ in which relatively trivial individual programming language features could only be compared by repeating controlled experiments, rather than generalisation from prior understanding of relevant cognitive principles.

The CDs framework was welcomed, both as one of the first comprehensive frameworks synthesising the psychology of programming literature, and also as a source of evidence-based guidance that was presented in a manner accessible to programming language designers. As a result, it has been adopted by programming language design groups in major corporations, and has also been widely influential in the academic community - for example, as the most cited original publication in the Journal of Visual Languages and Computing (Blackwell, 2006). Further overview of CDNs can be obtained from these publications, and from textbook coverage in (Carroll, 2003).

Although apparently widely used in industry, and extended in many ways, our own understanding of how CDs is used has been based largely on anecdotes, and discussions with collaborators. The result of this is that, although there is considerable consensus within the PPIG community regarding the value of CDs, adoption by other communities is not well understood. To the best of our knowledge there has not yet been a systematic review of the use of CDs. In this paper, we address this deficiency by conducting a systematic literature review of the papers that cite CDs.

2. Methodology

Following the guidelines proposed by Kitchenham (Kitchenham, 2004; Kitchenham et al., 2009, 2010), we conducted a systematic literature review to explore the ways that CDs framework are applied. We are
interested in analyzing 1) if CDs framework is used for evaluation or discussion, 2) which elements are widely used, and 3) How the cognitive dimensions of notation framework shaped the current literature.

2.1. Research Questions
During this research, we investigated the following research questions:

RQ1. Are CDs used as formative discussion tools or for summative evaluation?
RQ2. Which elements of the framework are most widely applied?
RQ3. How have CDs publications shaped the research literature?

We used quantitative and qualitative methods in our analysis to investigate these questions. To answer RQ1, we introduced the following subquestions:

RQ1.1 Did the citation occur only in a context that discussed evaluation of a completed design?
RQ1.2 Did the citation occur in a context where design properties were being discussed in the light of cognitive dimensions?

Similarly, we wanted to investigate the following subquestions to answer RQ2:

RQ2.1 Which of the dimensions proposed in (Green & Petre, 1996) are most widely applied?
RQ2.2 Have the newer dimensions proposed in (Blackwell et al., 2001) been applied?
RQ2.3 Which of the distinctive activities formulated in (Green & Petre, 1996) are applied?
RQ2.4 Which of the other CDs framework elements outlined in (Carroll, 2003) are most widely applied?

For RQ3 we wanted to investigate the following subquestions:

RQ3.1 Are CDs used in other ways, beyond discussion and evaluation of notational systems?
RQ3.2 What general scientific principles have been adopted from the CDs literature?

2.2. Search Process
We identified nine articles (listed in Table 1) to be the root publications for the cognitive dimension domain. Then, we collected all papers that cite any of these publications.

To create our corpus, we used Google Scholar to find metadata for papers that referenced a paper in the root collection. We generated a list that includes publication titles, number of citations, authors, date of publication, and whether the result has a direct PDF file link, or not. Then, we removed all non-English titles and duplicates from this list. Next, we downloaded all the PDF files from the filtered list. For publications that did not have a direct PDF link to download, we manually entered the full publication title as a Google search, reviewing the first two pages of search results for any site that would provide a downloadable PDF file of that publication. We used University of Cambridge library credentials to obtain a PDF from any publisher providing a valid institutional subscription. Where PDFs were paywalled (for example, some Springer conference proceedings), we did not proceed further.

For each downloaded PDF, we searched for specific strings to assess whether to include or exclude that publication. The remaining PDF files were searched for strings “green” or “cognitive”, and if the matched text is not a reference to cognitive dimensions (e.g., the colour green, or reference to cognitive psychology), we continued searching for repeat alternatives until the end of the file. If the first occurrence of “cognitive dimensions”, “green” or “cognitive” occurred in the bibliography, we noted the citation reference, usually an index number or BibTeX code, and searched the paper for this citation.
Table 1 – Source/Root Publications. Note that root04 and root05 are the same publication but cited differently.

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>root01</td>
<td>Usability analysis of visual programming environments: a 'cognitive dimensions' framework (Green &amp; Petre, 1996)</td>
</tr>
<tr>
<td>root02</td>
<td>Cognitive dimensions of notations (Green, 1989)</td>
</tr>
<tr>
<td>root03</td>
<td>Notational systems - the cognitive dimensions of notations framework (Blackwell &amp; Green, 2003)</td>
</tr>
<tr>
<td>root04</td>
<td>Cognitive dimensions of information artefacts: a tutorial (Green &amp; Blackwell, 1998a)</td>
</tr>
<tr>
<td>root05</td>
<td>Design for usability using Cognitive Dimensions (Green &amp; Blackwell, 1998b)</td>
</tr>
<tr>
<td>root06</td>
<td>A Cognitive Dimensions questionnaire optimised for users (Blackwell et al., 2001)</td>
</tr>
<tr>
<td>root07</td>
<td>Cognitive dimensions of notations: Design tools for cognitive technology (Blackwell et al., 2001)</td>
</tr>
<tr>
<td>root08</td>
<td>The cognitive dimension of viscosity: a sticky problem for HCI (Green, 1990)</td>
</tr>
<tr>
<td>root09</td>
<td>Instructions and descriptions: some cognitive aspects of programming and similar activities (Green, 2000)</td>
</tr>
<tr>
<td>root10</td>
<td>Ten years of Cognitive Dimensions in visual languages and computing: Guest Editor’s introduction to special issue (Blackwell, 2006)</td>
</tr>
</tbody>
</table>

Search for index number 99 was speeded up by initially searching for "[99]", "[99", "99]", or “99,”, in order to avoid occurrences of the same digits in a year, page number, data values etc. If one of the root papers had appeared in the bibliography, but no citation was found by the above search procedure, this was coded as “no citation”. Note that this does not guarantee that there was no citation, but simply that, if a citation did occur, it was phrased in a manner that did not match any of the above search strings. It is possible that for some PDF files, this resulted from OCR processing being more accurate for the bibliography than for the body text (for example, body text was sometimes incorrectly encoded as image regions).

2.3. Inclusion and Exclusion Criteria

Initially, we included all publications which cited any of the nine root articles. However, we had to exclude a subset of this collection due to: a) publications not written in English, b) duplicate publications. To remove non-English publications, we reviewed all titles manually, excluding any article with a non-English title. Also, we did a sanity check of titles by searching for Unicode characters outside the standard English character set. Using this new filtered collection, we removed remaining duplicates manually by sorting titles alphabetically, and by running a string comparison script.

In addition to duplicates and non-English publications, we excluded all c) publications without a downloadable PDF file. From our manual search, we either could not find a downloadable PDF file in the first two pages of the Google search results, or the publication was not accessible due to licensing and purchase requirements.

Furthermore, we excluded all d) PDF files that do not support text search. Usually this is because PDF files are image scans of old papers, or occasionally because OCR was incomplete or faulty.

Finally, e) publications without specific string occurrences are excluded. These strings are: “cognitive dimensions”, “green” (since Thomas Green was a co-author on all root papers) or “cognitive” (in this order).

2.4. Data Analysis

To investigate and answer our research questions, we conducted a quantitative, contrast and qualitative analysis.
2.4.1. Quantitative Analysis
To conduct a quantitative analysis on our corpus, we converted all PDF files to text. Then, we searched and counted the frequency of occurrence of certain keywords in each PDF. We separately considered four different sets of keywords: first, the original ‘core’ set of cognitive dimensions (RQ2.1); second, the set of ‘new’ dimensions (RQ2.2); third, notational activities (RQ2.3); and fourth, other CDs framework components (RQ2.4). These keywords along with their regex search terms are listed in Table 2.

2.4.2. Contrast Analysis
To support our quantitative analysis, we repeated the quantitative analysis process from the previous section on the proceedings of CHI 2016 (Kaye & Druin, 2016) and CHI 2017 (Mark & Fussell, 2017). The CHI dataset can be expected to slightly over-estimate the usage of cognitive dimensions terminology, when compared to other writing in English.

The occurrence frequencies in the CHI dataset are used as contrast material for the occurrence frequencies in the papers that cite cognitive dimensions. If a word occurs frequently in the CHI dataset it is excluded from further analysis or reporting in the quantitative sections of this paper as it would not be possible to determine whether it was used frequently because of its relevance to cognitive dimensions or a more general term of art.

2.4.3. Qualitative Analysis
In this section we describe the coding scheme for the qualitative analysis. The primary qualitative coding procedure was based on the text context in which the citation appeared. In some papers, there were multiple citations, in which case each occurrence was inspected before making the coding decision. The coding decision procedure meant that there were no cases in which multiple citations resulted in ambiguity of the coding decision.

- If citations occurred only in a context that discussed evaluation of a completed design (including the design of a notation such as UML, or the design of an interactive system), then this paper was coded as “evaluation”.

- If citations occurred in a context where design properties were discussed in the light of cognitive dimensions, this was coded as “discussion”. Note that, in some cases, discussion occurred within a context that might be described as formative evaluation of alternative design options. These cases were coded as discussion, if CDs were being used as a discussion vocabulary within a design process rather than for assessment of a completed design. In many cases, only a small number of dimensions were mentioned - we discuss this further in the results section.

- If the context of the citation did not mention any cognitive dimensions, did not discuss a design in terms of CDs, and did not mention evaluation, it was coded as “only cite”. in these cases, the authors were citing one of the root papers for a purpose unrelated to the main objective of CDs, for example as a reference to support mention of visual languages.

For those cases where a root paper was cited, but with no discussion of CDs, the apparent topic was recorded. An open coding approach was taken, in order to allow summary frequencies of the purposes for which citations were made. In cases where the authors used the citation to support a statement that directly contradicted the core contributions of CDs (for example, a claim that visual languages are always superior to text, or that text is always superior to visual languages), the complete text of the statement was copied for later analysis.

3. Results
Our search process resulted in an initial count of 2,555 articles. After cleaning the data, removing duplicates, non-English titles and non-downloadable PDFs, we ended up with 1,638 articles that we run our analysis on.
3.1. Quantitative Results

We address the subquestions of RQ2 through analysis of keyword frequencies. Each plot shows a histogram of the number of publications that mention each keyword a particular number of times. For each keyword, the mode is 1 (i.e., in most cases, the keyword is only mentioned once in the publication). However, there are a few publications that repeat a keyword many times - sometimes more than 30 times.

3.1.1. Core Dimensions

Figure 1 – CDs Framework Core Dimensions Histogram Comparison. Each histogram represents a dimension with the dimension name as the title, and the regex search term as the subtitle. In the parentheses \( a \) in \( b \), \( a \) is the total number of occurrences, and \( b \) is the total number of publications where the words occurred. The x-axis is the word occurrences, and the y-axis is the number of publications.

Figure 1 shows the frequency distribution charts of the core dimensions. Words that had high frequency in the contrast materials as described in section 2.4.2 were excluded. These were "abstraction", "consistency" and "visibility". "viscosity" was marginal, having slightly higher frequency in the contrast material than the other terms here, so it should its high occurrence here may be partially accounted for it being a semi-common word in the broader field beyond CDs.

From these charts it appears that "Viscosity" dimension has the highest number of counts and appeared in more articles than other core dimensions (3,116 counts in 558 articles). Whereas "Provisionality" has the lowest number of counts and appeared less compared to other core dimensions (416 counts in 140 articles).

From these results, we found that some articles have a specific dimension keyword that occurred over 40 or 50 times. These instances were used to introduce the dimensions keyword as part of new dimensions/framework, or the article is primarily focused on that specific dimension. For example, one article used the word "provisionality" 45 times. This article used this dimension and introduced it as part of communicative dimensions. Another article used "premature commitment" 50 times, which focused...
primarily on Premature Commitment dimension.

When dimensions keywords occurred more than 20 times, and in some cases over 10 times, in a single article, the article would use the dimension for evaluation. For example, when the dimension "closeness of mapping" occurred over 10 times in a single article, the article describes this dimension and use it for evaluation or comparison.

### 3.1.2. New Dimensions

![Figure 2 – CDs Framework New Dimensions Histogram Comparison](image)

The new dimensions (RQ2.2) appeared far less frequently than the core dimensions, as seen in figure 2. As with the core dimensions results, we removed two keywords that have high frequency in the contrast material: "indexing" and "specificity". However, an interesting finding from our analysis appears in the keyword "permissiveness". The results shows an occurrence of 47 times over eight publications of which one of these publications used the word 42 times. When we investigated that particular publication, we found that the publication is introducing that new dimension to the framework.

### 3.1.3. Activities

![Figure 3 – CDs Framework Activities Histogram Comparison](image)

As in previous sections, we excluded the two CDs activities because of their frequency in English: "modification" (2,561 counts in 420) and "transcription" (478 counts in 138 articles).

From figure 3, the CDs framework "incrementation" activity occurred 180 times in 66 articles. One of these articles used the "incrementation" term 45 times. From our investigation, it appeared that the author is describing the different CDs framework activities in every dimension.

An interesting finding is the word count of the CDs framework activity "exploratory design", which occurred 362 times in 111 articles.
### 3.1.4. Other Framework Components

![Figure 4 – Other CDs Framework Components Histogram Comparison](image)

Figure 4 compares keywords related to other aspects of the CDs framework. Once again, we exclude terms that frequent in the contrast material: "evaluation" and "notation". The component "Trade-off" has the highest number of occurrences, which occurred 1,686 times in 471 articles. However, based on random sampling of these articles, it seems that the word is not being used primarily in association with the CDs framework.

"Notational layer", which occurred nine times across seven articles is the least used component. "Abstraction Manager-Mechanism-Gradient-Barrier" component has a relatively high occurrences (849 times in 296 articles). This could be due to our query for that aggregate the results of the four keywords together "Abstraction Manager", "Abstraction Mechanism", "Abstraction Gradient" and "Abstraction Barrier".

Investigating the "sub-devices" component, which appeared 1,237 times in 27 article, we found that when the word occurred over 10 or 20 times within one article, that article used the component in an evaluation context.

### 3.2. Qualitative Results

#### 3.2.1. Further exclusions

71 papers were removed from the corpus during the qualitative coding process. 13 of these were not in English, but had not been detected by our initial filter. The remainder were documents that are indexed by Google Scholar, but were not academic publications, and made no reference to CDs. 12 papers were image scans with no searchable text. 50 further papers had no identifiable citation to any of the root papers, following the procedure specified above. Furthermore, 56 publications were determined to be duplicates, with titles that were identical or close to identical.

#### 3.2.2. Publications about CDs

90 publications in our dataset were classified as contributions to, or extensions of, the CDs framework itself. This count includes the original root papers (for completeness of the dataset in relation to citation metrics), presentations of the framework to other disciplinary audiences, methodological contributions proposing ways of using cognitive dimensions in applications or research, theoretical extensions and alternatives to the CDs approach such as the Tangible Correlates (Edge & Blackwell, 2006) or Ontological Sketch Modelling (Blandford & Green, 1997), and other frameworks inspired by the CDs approach such as Communicative Dimensions (Hundhausen, 2005), Collaborative Dimensions (Bresciani, Blackwell, & Eppler, 2008), "Physics of Notations" (Moody, 2009), and the Patterns of User Experience (Blackwell & Fincher, 2010). We do not further analyse these - we presume that much of this work is familiar to the specialist audience, and our main concern here is to analyse how CDs has been applied, rather than the history of how it has been, and continues to be, developed.
3.2.3. Use of CDs for summative evaluation

A total of 211 papers met the criteria defined above for a publication that reports using CDNs to evaluate a design, rather than as a discussion tool informing the research or design process. One of these claimed that the evaluation had been "formative", but was reported as a summative contribution (within a project validation phase). 22 papers expressed an intention to evaluate a system using CDs, but reported this in a "future work" section, meaning that it may or may not have occurred. One reported that the authors had intended to do an evaluation, but did not. 5 papers simply stated that the system had been evaluated using CDNs, but provided no details of what this had involved, or what the outcome had been. 25 of the evaluation papers stated that they had used the CDs questionnaire. These involved a fairly wide variety of questionnaire respondents, ranging from members of the research team to experimental participants. Two papers reported more "objective" evaluation using CDs - in one case as the basis of an experiment design, and in another as a numerical score of the number of dimensions that had been satisfied.

3.2.4. Use of CDs for design discussion

585 publications cited the root papers in order to discuss cognitively-relevant system properties, in the manner that may be considered as closest to the original advocacy in papers by Green, and by Green and Petre. In 16 of these cases, the authors expressed an intention to engage in CDs analysis, but this was stated in a "future work" section, with no further content. One paper simply stated that discussion had taken place, but gave no further information. 11 papers used the CDs questionnaire to structure discussion, among members of a design team or with potential users, while another described CDs as "a discount usability method, [that] can be used without users".

The remaining 555 papers in this category simply go about the business of discussing specific dimensions, considering desirable attributes of notational systems for particular applications, in the way that a PPIG audience might expect. This discussion involved a wide range of different degrees of detail. Some papers were devoted to comprehensive CDs analysis of a particular application area, including consideration of activities, profiles, relevant dimensions and so on. However, this was relatively rare, as indicated by findings in the quantitative analysis above. More often, authors simply named specific dimensions as being relevant to their work, and then proceeded to discussion of design factors that would be influenced by that dimension. The relative frequency of dimensions chosen has been discussed in the quantitative analysis above.

One distinctive characteristic was the large number of these papers, 208, in which a single dimension was the main focus of attention. Different dimensions were relevant on different occasions, as reflected in the numerical distributions reported above.

15 papers use the CDs as a coding frame for qualitative analysis. In two of these cases, the qualitative data was collected using the CDs questionnaire as a prompt to participants, while the remainder collected data in other ways, including interviews, focus groups, Wizard of Oz and think-aloud studies. We consider these cases as a discussion tool for researchers, despite the fact that a specific design may not be under discussion.

3.2.5. Further citations of the CDs framework

Finally, 521 of the publications in our corpus cite one of the root papers, but do not engage either in discussion or evaluation of notational systems. In many cases, these are simply "completeness" citations, in which the authors have presumably found it necessary to report that the CDs framework exists, but see no pressing need to apply it in their own research. This applies to 260 of the cases in this category.

3.2.6. Open coding of other citations

Slightly more interesting are the publications that cite one of the root papers, but for other purposes beyond application or extension of CDs. As described in the methods section, we carried out further thematic open coding of these 261 papers.

We found 46 papers cite the Green and Petre paper (Green & Petre, 1996) as reference for the concept of a visual language. A further 4 papers cite it as a reference to visualisation or representation, 7 as
a reference to domain-specific languages, 4 to dataflow languages, 1 to high-level languages and 4 as a reference to user interfaces in general. The paper is also used as a citation for specific languages that appear as case study examples: 2 citations as a reference to spreadsheets, 4 to LabView, and 1 to Prograph.

27 publications cite CDs papers as general references to cognitive psychology, including the nature of cognition (10), learning (7), cognitive load (2), as well as perception, understanding, working memory, tacit knowledge, implicit information, external representation etc. 18 papers cite the Green and Petre paper as a reference to support the concept of usability in general, and 1 as a reference to the field of HCI. 5 papers cite a CDs paper as a reference for psychology of programming or PPIG.

Specific kinds of programming activity are another focus of citations, with 6 papers referring to CDNs as evidence for exploratory design, 4 for opportunistic planning, 2 for plan composition, 2 for direct manipulation, 1 for information seeking, 1 for searching, 1 for diagram editing, 1 for program comprehension, 1 for modularisation, 2 citing novice/expert differences, and 3 for programming in general.

35 publications cite a CDs paper in support of earlier work by Green and colleagues. 19 of these address the problem of superlativism - the assumption that any one programming language could be best for all purposes - while 16 report the "match/mismatch" finding, that notations are effective in situations where they match the structure of the task.

CDs is reasonably often cited to illustrate a general analytic strategy, including an emphasis on notation (11), on information artefacts and representation (7), or on editing environments (1) as well as the way these concerns are addressed in design processes (6) with the consequent need to acknowledge trade-offs in design (5). Three papers refer to CDs to demonstrate the value of design vocabularies as a research contribution, while another two cite it as an example of a research framework that is open and extensible. Two papers (mistakenly) cite a CDs publication as the source for "Physics of Notations".

4. Discussion and Future Work

From our search process, we found that some articles were indexed by Google Scholar, but were not academic papers or did not make reference to CDs. We attribute these to data quality issues in Google Scholar, or the way in which we were using it. In future work conventional citation services, although having less complete coverage than Google Scholar, could be used to test whether bugs resulted in some publications not being found in our search procedure.

For future filtering, a more sophisticated filter could use an edit-distance metric to check for duplicate titles. However, we note that a number of authors publish very similar papers with slightly different titles. This is, unfortunately, common academic practice. In this analysis we have followed academic convention by counting these as separate publications if they appeared in separate venues. However, it is an issue that should perhaps be addressed more rigorously in future SLR studies.

We mentioned that we excluded publications that are images scans with no searchable text. We would like to go over the image scans articles and code them manually in future work.

From the quantitative results, it appears that the "new" dimensions suggested as extensions beyond the set described in the earliest CDs publications appears far less frequently (RQ2.2). This suggests that many authors regard the original set as a complete basis for evaluation, despite the stated intention that the framework should be considered as extensible. We noted during this analysis that articles refer to exploratory design activity more often than other activities (notwithstanding our exclusion of very common terms). This reflects the fact that exploratory design has not been a common focus of attention in software engineering research, meaning that CDs has drawn attention to a phenomenon that was previously neglected.

From the qualitative results, more often, evaluation using CDs was complementary to a more conventional user study, with the results of the two investigations being aggregated for summative evaluation of a prototype system. This approach was particularly associated with certain research groups.
In our analysis of the use of CDs for design discussion, the use of CDs to draw attention to specific properties of interactive systems is very much consistent with Green’s original observations and intent, as in the highlighting of "viscosity" as a specific dimension that could become a focus of greater design attention in future (Green, 1990). As seen in the quantitative analysis, the CDs literature has been influential in promoting a variety of specific design concerns for notational systems, including secondary notation, and in particular closeness of mapping, which has been received by the research community as a property that is especially valuable and significant for the design of domain-specific notations.

As mentioned in the qualitative results section, some publications were coded as using CDs for discussion, despite the fact that no specific designed artefact is discussed. However, the kinds of activity resulting in qualitative data were all broadly situated within a design research context, meaning that the authors’ findings might in future be applied in design work. The value of a discussion tool here is in focusing analysis by the researchers toward design concerns that are likely to be relevant in notational systems.

The results indicate a significant number of publications that cite the root papers without engaging in discussion or evaluation of notational systems. This is a typical academic convention, that work may be cited due to its reputation, rather than because of a specific influence on the citing work. This is a cautionary observation, not only for members of the PPIG community who hope to have some influence on future technical practice and might use CDs as a benchmark of their highest aspirations, but also for those who might be tempted to regard citation metrics as strong evidence of scientific value.

As the results suggested in the open coding of other citations, there are a number of papers that cited the Green and Petre paper (Green & Petre, 1996) as a reference for the concept of a visual language. Since it was published in the Journal of Visual Languages and Computing, presumably any paper in this journal would have done - or indeed a citation to the journal itself - but this paper was taken to be in some way representative of the whole field. In addition, the Green and Petre paper offers a rich selection of case studies, and is occasionally cited by authors who have no interest in CDs, but simply need a source for one of these. It is thus cited as reference for the rocket simulator program, for logic gates, or paperless interfaces. Some are more puzzling, for example citing a CDs publication as a reference for XML pairing, use of space, of metaphor, or for ‘non-linear art’. Some of these may have been referencing errors, or perhaps more innovative chains of reasoning and connection that, while clear in the author’s mind, have not been fully communicated to the reader.

From the 35 publications that cite the CDs in support of earlier work by Green and colleagues, some authors appear to have misinterpreted these central points of the framework, for example "In accordance with Green et al., we agree that graphical representations are, in most cases, inherently superior to textual representations" or, as another author states in direct contradiction, "But Visual Programming Languages did not work well in practice as diagrams cannot be as expressive as text". Although these quotes are relatively extreme, several other papers cite CDNs in passing as a reference to support either the claim that text is inherently superior to graphics, or that graphics are inherently superior to text.

From our qualitative results, there are a small number of papers that reflect on design theory, including the need for frameworks, the role of evaluation etc. Unfortunately, a small number of these also seem to ignore or misunderstand the key concerns of CDs, contradicting the central message, as in "Generally, the Cognitive Dimensions framework cannot be used to directly guide us in creating visually-enhanced notations for code."

5. Conclusions

This paper has presented the results of a systematic literature review, studying applications of the Cognitive Dimensions of Notations framework, as discussed in over 1,600 publications. The CDs framework is very widely cited, and considered influential, yet much of the literature draws on CDs in ways that are different to the original intentions of the framework developers.

We did find a significant number of papers that use CDs as a discussion tool, in order to draw attention
to cognitively-relevant aspects of system design. But in many cases, the discussion focused on a single dimension, meaning that the role of CDs has been to alert language designers to a particular property not previously understood, rather than introducing new sophistication to the design process more broadly.

There are many richer discussions of CDs properties, but these are often used in a purely summative way, after the design is completed, rather than as intended to inform trade-offs and design manoeuvres during an ongoing process of design guided by research. In some cases, even evaluation appears to be an afterthought, with authors expressing their intention to evaluate their work in the future, presumably placing higher priority on publication of their design proposals than on evaluation of the usability benefits they hope these proposals might deliver.

As one of the most highly cited references in Psychology of Programming, it is also interesting to note that the most popular CDs publication is taken as a single representative source that is assumed (from the perspective of software engineering researchers) to represent everything that is known in our field, or in the fields of visual languages, usability, HCI, or cognitive science.

When a paper becomes so widely cited, it is inevitable that it should also become misunderstood, and it is not unusual for researchers to cite this paper in support of claims that are opposite to the actual scientific findings. These practices suggest a degree of laziness in contemporary peer review, and perhaps a lack of rigour in fields such as Software Engineering and Information Systems. Although our own field often values innovation and insight beyond the incremental observation of controlled experiments, we should be alert to the potential for misunderstanding, amongst audiences who may be unfamiliar with design-oriented research and broad-brush analysis.

6. References


<table>
<thead>
<tr>
<th>Keyword</th>
<th>Regex Search Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>Viscosity</td>
<td>&quot;[Vv]iscosity&quot;</td>
</tr>
<tr>
<td>Visibility</td>
<td>&quot;[Vv]isibility&quot;</td>
</tr>
<tr>
<td>premature commitment</td>
<td>&quot;[Pp]remature[\s\S]?[Cc]ommitment&quot;</td>
</tr>
<tr>
<td>hidden dependencies</td>
<td>&quot;[Hh]idden[\s\S]?[Dd]ependencies&quot;</td>
</tr>
<tr>
<td>role-expressiveness</td>
<td>&quot;[Rr]ole[\s\S]?[Ee]xpressiveness&quot;</td>
</tr>
<tr>
<td>error-proneness</td>
<td>&quot;[Ee]rror[\s\S]?[Pp]roneness&quot;</td>
</tr>
<tr>
<td>abstraction</td>
<td>&quot;[Aa]bstraction&quot;</td>
</tr>
<tr>
<td>secondary notation</td>
<td>&quot;[Ss]econdary[\s\S]?[Nn]otation&quot;</td>
</tr>
<tr>
<td>closeness of mapping</td>
<td>&quot;[Cc]loseness[\s\S]?[Oo]f[\s\S]?[Mm]apping&quot;</td>
</tr>
<tr>
<td>consistency</td>
<td>&quot;[Cc]onsistency&quot;</td>
</tr>
<tr>
<td>diffuseness</td>
<td>&quot;[Dd]iffuseness&quot;</td>
</tr>
<tr>
<td>hard mental operations</td>
<td>&quot;[Hh]ard[\s\S]?[Mm]ental[\s\S]?[Oo]peration[s]?&quot;</td>
</tr>
<tr>
<td>provisionality</td>
<td>&quot;[Pp]rovisionality&quot;</td>
</tr>
<tr>
<td>progressive evaluation</td>
<td>&quot;[Pp]rogressive[\s\S]?[Ee]valuation&quot;</td>
</tr>
<tr>
<td>Creative</td>
<td>&quot;[Bb][Cc]reative[\s\S]?[Aa]mbiguity&quot;</td>
</tr>
<tr>
<td>Specificity</td>
<td>&quot;[Bb][Ss]pecificity&quot;</td>
</tr>
<tr>
<td>Detail in context</td>
<td>&quot;[Bb][Dd]etail[s]?[\s\S]?[Ii]n[\s\S]?[Cc]ontext&quot;</td>
</tr>
<tr>
<td>Indexing</td>
<td>&quot;[Bb][Ii]ndexing&quot;</td>
</tr>
<tr>
<td>Synopsis</td>
<td>&quot;[Bb][Ss]ynopsis&quot;</td>
</tr>
<tr>
<td>Free rides</td>
<td>&quot;[Bb][Ff]ree[\s\S]?[Rr]ides[s]?&quot;</td>
</tr>
<tr>
<td>Useful awkwardness</td>
<td>&quot;[Bb][Uu]seful[\s\S]?[Aa]wkwardness&quot;</td>
</tr>
<tr>
<td>Unevenness</td>
<td>&quot;[Bb][Uu]nevenness&quot;</td>
</tr>
<tr>
<td>Lability</td>
<td>&quot;[Bb][Ll]ability&quot;</td>
</tr>
<tr>
<td>Permissiveness</td>
<td>&quot;[Bb][Pp]ermissiveness&quot;</td>
</tr>
<tr>
<td>transcription</td>
<td>&quot;[Tt]ranscription&quot;</td>
</tr>
<tr>
<td>incrementation</td>
<td>&quot;[Ii]ncrementation&quot;</td>
</tr>
<tr>
<td>modification</td>
<td>&quot;[Mm]odification&quot;</td>
</tr>
<tr>
<td>exploratory design</td>
<td>&quot;[Ee]xploratory[\s\S]?[Dd]esign&quot;</td>
</tr>
<tr>
<td>Evaluate, evaluation...etc.</td>
<td>&quot;[Ee]valuat[\w]+&quot;</td>
</tr>
<tr>
<td>abstraction manager</td>
<td>&quot;[Aa]bstraction[\s\S]?([Mm]anager</td>
</tr>
<tr>
<td>abstraction mechanism</td>
<td>&quot;[Aa]bstraction[\s\S]?([Mm]anager</td>
</tr>
<tr>
<td>abstraction gradient</td>
<td>&quot;[Aa]bstraction[\s\S]?([Mm]anager</td>
</tr>
<tr>
<td>abstraction barrier</td>
<td>&quot;[Aa]bstraction[\s\S]?([Mm]anager</td>
</tr>
<tr>
<td>notation</td>
<td>&quot;[Nn]otation&quot;</td>
</tr>
<tr>
<td>formative</td>
<td>&quot;[Ff]ormative&quot;</td>
</tr>
<tr>
<td>broad brush</td>
<td>&quot;[Bb][road][\s\S]?[Bb]rush&quot;</td>
</tr>
<tr>
<td>notational layer</td>
<td>&quot;[Nn]otational[\s\S]?[Ll]ayer[s]?&quot;</td>
</tr>
<tr>
<td>trade-off</td>
<td>&quot;[Tt]rade[\s\S]?[Oo]ff[s]?&quot;</td>
</tr>
<tr>
<td>sub-devices</td>
<td>&quot;[Ss]ub[\s\S]?[Dd]evice[s]?&quot;</td>
</tr>
</tbody>
</table>

**Table 2 – Quantitative analysis keywords and their regex search terms**
Abstract

Usability issues that exist in security APIs cause programmers to embed those security APIs incorrectly to the applications they develop. This results in introduction of security vulnerabilities to those applications. One of the main reasons for security APIs to be not usable is currently there is no proper method by which the usability issues of security APIs can be identified. We conducted a study to assess the effectiveness of the cognitive dimensions questionnaire based usability evaluation methodology in evaluating the usability of security APIs. We used a cognitive dimensions based generic questionnaire to collect feedback from programmers who participated in the study. Results revealed interesting facts about the prevailing usability issues in four commonly used security APIs and the capability of the methodology to identify those issues.

1. Introduction

Cyber attacks and data breaches have become exceedingly common that we hear about such incidents more often than not (Acar, Fahl, & Mazurek, 2016). Even though new technologies and methodologies to secure applications and data are introduced, so far they have failed to reduce the number of attacks and data breaches (Acar et al., 2016). One of the main reasons for this failure is that some security mechanisms are not easily learnable and understandable to programmers, and they find it difficult to use those mechanisms correctly when developing applications (Wurster & van Oorschot, 2009). Specially, security Application Programming Interfaces (APIs) that provide the interface for programmers to access security mechanisms are often not usable (Acar et al., 2016) (Wurster & van Oorschot, 2009) (Myers & Stylos, 2016). Therefore, programmers may end up using them incorrectly and hence, results in introducing security vulnerabilities to the applications they develop.

If the usability of security APIs can be improved, it will help programmers develop more secure applications and therefore, will help prevent possible cyber attacks and data breaches (Acar et al., 2016) (Myers & Stylos, 2016). However, so far security API developers have failed to develop security APIs in a way that programmers who are using security APIs will find them usable. One of the major obstacles for developing usable security APIs is the lack of a proper methodology to evaluate the usability of security APIs (Myers & Stylos, 2016) (Mindermann, 2016). If there is a proper methodology to evaluate the usability of a security API and identify usability issues that may exist in it before it is delivered as a finished product, developers can fix those issues and deliver a more usable product (Mindermann, 2016).

The main objective of our research is to facilitate API developers to develop usable security APIs, by proposing a systematic approach to evaluate the usability of security APIs. To achieve this objective, first we will assess the applicability of existing API usability evaluation techniques, which have been used to evaluate the usability of general APIs, for evaluating the usability of security APIs. In this particular study, we focus on the API usability evaluation methodology based on the cognitive dimensions questionnaire (Clarke, 2004). In our previous work, through a literature survey, we argued that the cognitive dimensions framework and questionnaire proposed by Clarke (2004) may not be sufficient to evaluate the usability of security APIs (Wijayarathna, Arachchilage, & Slay, 2017). Therefore, we proposed an enhanced version of this framework and questionnaire to be used in security API usability evaluations (Wijayarathna et al., 2017). In this study, we evaluate the applicability of both these questionnaires (i.e. Clarke (2004)’s questionnaire and our questionnaire (Wijayarathna et al., 2017)) for evaluating the usability of security APIs through an empirical study.
We are in the process of conducting an empirical study using participants who are programmers. Even though we have not completed the study, early results revealed useful insights that could be used to outline a methodology for usability evaluations of security APIs. This work in progress paper will discuss the study we conducted and the observations made so far.

2. Related Work
Several methods have been introduced to improve the usability of general APIs such as cognitive dimensions questionnaire based empirical evaluation (Clarke, 2004), heuristic evaluation (Grill, Polacek, & Tscheligi, 2012) and API walk through method (O’Callaghan, 2010). However, these methodologies do not consider security related usability characteristics such as “security knowledge prerequisites”, “hard to misuse” and “amount of security related code programmer has to implement” when evaluating the usability of APIs (Green & Smith, 2016) (Wijayarathna et al., 2017). This suggests the inapplicability of these existing evaluation techniques for evaluating the usability of security APIs.

In this particular study, we are focusing on the cognitive dimensions questionnaire based API usability evaluation methodology (Clarke, 2004), because it seems to be more effective than the other methodologies for evaluating the usability of security APIs (Wijayarathna et al., 2017). In this methodology, experimenters recruit programmers and ask them to complete an individual task which may involve writing, reading or debugging a code that use the API under evaluation (Clarke, 2004). Once the task is completed, each participant has to answer the cognitive dimensions questionnaire based on their experience in completing the task. Evaluators will identify usability issues exist in the API by going through the answers for the questionnaire provided by the participant.

The cognitive dimensions framework and questionnaire used by Clarke (2004) consists of 12 dimensions and questions to cover each of these dimensions. However, these 12 dimensions does not cover security API related usability aspects such as “security knowledge prerequisites”, “hard to misuse” and “amount of security related code programmer has to implement” (Wijayarathna et al., 2017). Therefore, in our previous work, we proposed an enhanced version of the framework and questionnaire that consists of 15 dimensions (Wijayarathna et al., 2017). New questionnaire we proposed consist of all the questions used by Clarke (2004) as well as new questions that we introduced. The work described in this paper intends to evaluate the applicability of both these versions of cognitive dimensions framework and questionnaire for evaluating the usability of security APIs.

3. Methodology
For this study, we used four different programming tasks that use four different security APIs. We used more than one API because, we are trying to derive a generalized result for all security APIs. The APIs we used are Google authentication API, Bouncy castle lightweight crypto API, OWASP Enterprise Security API (ESAPI) and a proprietary API that provides SSL related functionalities. For each API, we designed a programming task which makes use of the most important objects and methods exposed by the API. For example, in the task which uses the Google authentication API, participants had to make use of signIn and signOut functions and other supporting functions. All the tasks were designed to use Java or JavaScript programming languages.

We recruited participants who had at least one year of experience as a programmer. At the recruitment, they had to complete a short demographic questionnaire where we collected information such as their experience as a programmer, their proficiency in Java and JavaScript programming languages and their previous experience with the aforementioned four security APIs. Then we assigned a programming task from the four tasks to each participant. They could complete it in our laboratory or remotely using their own computer. While completing the task, they had to verbalize their thoughts and their computer screens were recorded. On completion, they had to answer our proposed questionnaire (which consisted of all the questions used by Clarke (2004) as well as new questions that we added).

Then the first author analyzed the answers provided by each participant to the questionnaire and identified usability issues each participant came up with. Then he divided those issues as “identified by
Clarke (2004)’s questions” and "identified by questions we added". Then the first author went through the screen recordings, think aloud results and code artifacts produced by participants, and identified usability issues each participant came up with.

4. Results

So far we collected and analyzed data of 7 participants. 3 of them did the task which required to use OWASP ESAPI, 2 participants did the SSL task, 1 participant did the task which required to use Google authentication API and 1 participant did the task which required to use Bouncycastle crypto API. 5 participants completed the study remotely while 2 participants completed the study in laboratory. Every participant used their preferred Integrated Development Environment (IDE) where 5 participants used IntelliJ Idea, 1 participant used Netbeans and 1 participant used Eclipse.

Table 1 summarize the number of potential usability issues identified by each participant with each method and those numbers as a percentage of the total number of issues identified by the particular user.

<table>
<thead>
<tr>
<th>Participant number</th>
<th>Total number of issues identified</th>
<th>Number of issues identified through observation</th>
<th>Number of issues identified from the answers to the complete questionnaire</th>
<th>Number of issues identified from the answers to Clarke’s questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>29</td>
<td>16 (56%)</td>
<td>15 (52%)</td>
<td>9 (31%)</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>3 (38%)</td>
<td>6 (75%)</td>
<td>3 (38%)</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>5 (38%)</td>
<td>10 (85%)</td>
<td>8 (62%)</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>5 (25%)</td>
<td>19 (95%)</td>
<td>15 (75%)</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>1 (8%)</td>
<td>11 (92%)</td>
<td>9 (75%)</td>
</tr>
<tr>
<td>6</td>
<td>17</td>
<td>12 (71%)</td>
<td>9 (53%)</td>
<td>5 (29%)</td>
</tr>
<tr>
<td>7</td>
<td>15</td>
<td>7 (47%)</td>
<td>10 (67%)</td>
<td>7 (47%)</td>
</tr>
</tbody>
</table>

Mean Percentages 40% (sd = 20.5%) 74% (sd = 17.6%) 51% (sd = 19.7%)

Table 1 – Issues Identified by each participant in each method.

5. Conclusion and Discussion

From the results we captured, there were few main conclusions that we could made. Average 74% (SD = 17.6%) from the total issues identified by each user have been revealed from his/her responses to the questionnaire. This suggests that questionnaire method is effective for collecting feedback from participants in API usability studies. However, the intersection between issues identified by observing users and issues identified through the questionnaire was small (mean = 25.8%, sd = 17.6%).

By observing 7 participants and analyzing the code they provided, we identified 44 potential usability issues that exist in the 4 APIs used. Out of these 44 issues, only 20 (45%) were revealed by the questionnaire answers. Even though the other 24 issues were not identified by the responses to the questionnaire, questionnaire answers gave a high level idea about some of these issues. For example, by observing the participant who did the programming task that used Bouncycastle API, we identified that “parameters of Scrypt.generate() method are not obvious”. This issue was not identified by the answers that participant provided for the questionnaire. However, his answers to the questionnaire mentioned that “API does not reveal information about function parameters and what they return” which gives a high level opinion about the previously discussed issue. Giving a high level view of the actual usability issue has advantages as well as disadvantages. The questionnaire answer does not directly reveal the exact place the issue exist. Therefore, evaluators will not be able to locate where the exact issue is. If the task only requires to use few objects and functions, evaluators will be able to identify where the issue is easily. However, receiving a high-level idea could be useful in some cases. For example, for large APIs like Bouncycastle, it is not practical to design a programming task that will cover all the objects and functions.
exposed by the API. However, as the questionnaire provides a general feedback on the usability issues, evaluators can use them as guidelines to identify usability issues that exist in the components of the API, including those that were not used in the programming task. Sub devices concept that was introduced by Blackwell and Green (2000) can possibly be the solution for the above mentioned issue. In API context, these sub devices can be mapped into classes/functions of the API. However, Clarke (2004)’s cognitive dimensions questionnaire has not used this and also this will lengthen the questionnaire.

Furthermore, questionnaire method identified some issues that could not be identified by observation and code analysis. Each participant’s questionnaire answers revealed average 8.8 (SD=3.8) issues that could not be identified by observing the participant and by analyzing the code s/he provided. Specially, questionnaire method seems to reveal issues related to cognitive dimensions such as progressive evaluation, premature commitment, API elaboration, consistency, end user protection and testability that could not identify by observing the participant and analyzing the code artifacts provided by the participant.

Our improved version of questionnaire revealed 11.6 (SD = 4.6) usability issues per participant compared to Clarke (2004)’s questionnaire which revealed 8 (SD = 3.7) issues per participant. This suggests that the improved version of the cognitive dimensions questionnaire is more effective in identifying usability of security APIs compared to the questionnaire proposed by Clarke (2004).

6. Future Work
One of the main limitations of our current results is that the usability issues were identified solely by the first author. The identified issues may change if this analysis was done by a different person. This is not only a limitation of the study we conducted, but also a limitation when using cognitive dimensions based usability evaluation for identifying usability of APIs. One of the possible solutions for this is to do the analysis by several persons (Analyst Triangulation), so that we might get a more general result.

This is a still work in progress where we have only used 7 participants so far. We will continue doing this study with more participants to get a more statistically significant result on the applicability of cognitive dimensions questionnaire for security API usability evaluations.

7. References


Abstract
This paper describes research into supporting the creation of engaging learning experiences with programming. It describes a fieldwork study conducted to explore the framing of learning programming in tasks that motivate and are of value to the learner. The findings – that ownership, personalisation and purpose can support learning whilst providing considerable motivation - should provide educators with insights to support key design decisions for the creation of engaging programming learning experiences. They support the assertion that factors outwith programming content can significantly affect success in programming. The complex interplay between different skills associated with computer programming will remain a challenge to learners. When placed in a rich context that fits the learner well and supports the learning aims, many of these difficulties can be overcome. The work described here suggests it should be possible to compile positive features of a learning experience to enable learners have the best possible opportunity to engage with and succeed with computer programming.

1. Introduction
This work describes an investigation into the value of learner ownership, personalisation and purpose for learners of programming. It rests upon the findings of previous studies (Martin and Hughes, 2011; Martin et al., 2017a; Martin et al., 2017b) in which learners were constrained to solve a puzzle devised by the educator. In contrast, in this Digital Makers study, design decisions were intended to make the product less constrained for the learners. This made it possible for learners to apply their newly acquired programming skills to solve a problem of their own.

In the first work (Martin and Hughes, 2011), a Robot Dance investigation demonstrated that a tight cycle of content delivery and learner consolidation was effective for a wide range of learners. The second study (Martin et al., 2017a), known as Whack a Mole, attempted to mimic this but increased the learners’ control over the pace by using video tutorial, with mixed results. The subject of this paper, Digital Makers, is a study built upon the Robot Dance approach with a gradual loosening of the cycle as the session progressed. In a ‘Robot Dance in the Community’ study (Martin et al., 2017b), it was shown that if given choice, learners would form different learning groups. This freedom was given in the Digital Makers study for the second part of the workshop. In contrast to the two previous studies, Digital Makers was a day-long event, giving much more space for learners to acquire skills and then apply them. It aimed to address the research question:

How do personalisation, ownership and purpose in an activity affect introductory programming learning?

2. Background
The Digital Makers study was part of One Day Digital, a series of digital making events organised by Nesta Scotland (2014). The aim of the Nesta event was to give 400 young people a taste of digital making. A range of five workshops was assembled, with topics including programming, stop motion animation, web making and games programming. The events ran on four consecutive weekends in Dundee, Aberdeen, Glasgow and Edinburgh respectively. Each event had five one-day workshops on offer to young people, starting at 10:00 a.m. and concluding at 5:00 pm. The events were advertised widely and participants had to register for the workshops in advance. Digital Makers was one of the five workshops delivered at each venue.

2.1. Physical Computing
Physical Computing is the construction of a digital device that uses a range of physical input and output components. The Arduino has emerged as the dominant microcontroller in the field of physical
computing (Arduino, n.d.). Originally, the Arduino was developed to assist interaction design learners to build prototypes with digital functionality. From this beginning, Arduino has grown to be a very powerful and accessible physical computing board to work with. One of the strengths of Arduino is the fact that the project has both open source software and hardware. This has resulted in a vast and varied community of learners, makers and professionals building a wide array of things (Banzi, 2012).

Building physical computing projects requires a range of interrelated core skills including electronics, craft and computer programming. Each of the core skills share some common features. They all have elements of design, creativity and problem solving and they often have a specific notation to support needed specification of designs and code. These shared features offer some interesting opportunities for cross-domain learning. For example, if the project is an alarm clock, there is a need to specify the electrical components to be used and their relation to each other: buttons, display and audio output. There are formal notations for this task and at some point there must be a transition from a circuit diagram to a physical layout such as a printed circuit board. In terms of the software, there is a need to specify the various functions of the alarm clock the end user can perform: set an alarm time, turn an alarm on and off, and so on. Various notations support the transition of a software design task from conceptual design through to the final notation of source code. The same is true for the physical product: the materials used, the form and the sequence of interaction all add to its design language, which can be captured in various notations, such as sketching, storyboards and mood boards. Each of these elements must come together for the alarm clock to function and fit the needs of the intended user.

This range of desirable competencies affords a degree of flexibility to the individual learner’s experience. Some projects may have very well established design features that demonstrate good understanding of the problem and the user who will engage in the technology. Other projects may be light on design and user consideration, but may be technically sophisticated with extensive electrical or computational ability demonstrated. When a group forms containing individuals with a range of such skills, there is a good opportunity for the learners to see the value of collaboration and varied contributions to a project.

2.2. Physical Apps
A physical app is essentially a tangible equivalent of a mobile device app. It should be a simple compact physical computing object. It should solve a single well-defined problem. This naturally lends itself to educational workshop activities with a truncated time for completion. Rather than attempting a complex multifaceted project, the aim is a single purpose project that is buildable in as short a time as possible. In contrast to a vertical prototype, which is a technical demonstrator for part of a system, the physical app stands on its own. This is important so that the learner has the opportunity to experience the full development cycle from idea, design, build and test, to demonstrate.

3 Study Description
There were two parts to the study. In the morning the learners were walked through the process of wiring and programming some components with their Arduino; for this stage learners worked as individuals. In the second stage, learners were given the chance to self-select groupings and build a physical app utilising the morning’s teaching. The study ended by giving all groupings an opportunity to share their idea and resultant physical app with the whole group. These stages are described next.

3.1 Morning: Laying the Foundation
The learners who attended these events came from a relatively large geographical area. There was therefore a good chance that individuals would not know each other. For this reason, the first activity was planned to 'break the ice' and set the scene for the day of making, sharing and appraising the work of their peers. A volunteer was sought from the group; the volunteer was placed at one end of an open space with his/her back to a small box. The rest of the learners gathered round. The volunteer was then instructed to throw small soft balls blindly over their head in the rough direction of the box. The rest of the group was encouraged to offer advice and direction on what the volunteer must do to get closer to getting a ball in the box. The group was coached as necessary. When a ball finally made it into the box (which on all occasions it did), a discussion was facilitated with the group. Example discussions related to how the volunteer felt when a little stressed and on the spot, and what was found to be helpful, such as general support ("you are doing well") or specific feedback ("angle is good but a little more power").
This was used to highlight that demonstrating something you have made to a group can be stressful and make you feel a little exposed. This stress can be alleviated, however, in a supportive environment. The other important aspect is that when someone suggests a change to a piece of work or action, it is not necessarily a negative thing or a criticism of what has been done. Often it indicates they have thought about the problem, reflected on your solution and have identified a possible improvement. This should be taken as positive, and a sign of respect and consideration of your work.

Following the ice-breaker, participants were given the knowledge and understanding pre-test to complete individually. This led into the taught component of the day where wiring and programming of a range of Arduino components was taught. In small sections, the wiring-up of a component was demonstrated and described, and then carried out by the learners, with individual support as required. This was very informal and small groups allowed a good degree of dialogue between tutor and learner. Following this, the programming of the component was demonstrated and then carried out by the learners. In this iteration of short demonstration followed by enactment by learners, the learners completed the following tasks: making an LED blink, using a potentiometer to control the blink rate and using a button to make the LED blink when pressed. The first set of examples took around 40 minutes to complete.

To change the activity and introduce a creative disruption to the flow of tuition, the participants were then guided through an idea-generation session. Equipped with Post-Its and marker pens, learners were asked to identify three things that make them excited and note them down concisely on the post-it wall. Learners were then encouraged to bring their Post-Its to the front and stick them on a predetermined part of the wall that was visible to the group throughout the day: the ‘wall of situations’. The ideas gathered together on the wall served as an information radiator (Sharp et al., 2009) for use later in the day. This process was repeated for things that make them cross and for things that make them stressed. The purpose of this was partly to move the learners out of their seats and force them to change where their attention was placed. The wall of situations also served to condense physical app ideas to form around later that day. Bringing all the ideas together allowed learners to react to each other’s experiences and stimulated memories and new ideas.

The learners were then guided through some additional Arduino output devices: servo, speaker and red green blue (RGB) LED. The servo and speaker both offered the opportunity to show learners the examples that are built into the Arduino IDE. In particular, the servo requires an external library; this offered the opportunity to describe how software libraries are used as structuring tools. Having been shown the Arduino examples (which are very accessible and well documented), the learners had a way to explore further capabilities of the equipment they were using after the teaching had concluded.

The final example the learners constructed was a red, green and blue colour mixer. With a single RGB LED and three potentiometers, a physical colour mixer was constructed. This task requires a relationship between the potentiometers and the intensity of the red green and blue component of the LED to be established. The potentiometer provides a value in the range 0 to 1023 and the intensity of the LED output is given a value in the range of 0 to 255. This requires various built-in functions and the use of variables and assignment. In a natural progression, the learners were shown how to group this now quite complex program into a single user defined function and how to alter this so that the colour of LED was specified by three parameters passed to the function. Extending this further and utilising the random function and bringing in some sound with loudspeakers, playing beeps of a program specified tone, the learners created a light and sound show.

### 3.2 Afternoon: Sketch and Build

The afternoon started with the group revisiting the post-it wall of situations that excite, irritate and stress them. Learners were asked to pick several Post-Its they could relate to and expand upon them. The idea of the physical app was then described: a single-purpose object, like a kitchen appliance, which will perform one task or solve one problem well. Finally, the learners were given three hours to build a physical app based on one of the ideas from the selected Post-Its, with support available as required.

The participants were introduced to the technique of storyboarding developed by Disney in the 1930s: creating a series of linked rough sketches that communicate an interaction or chain of events. The storyboard technique was used to support their thinking and help them to articulate their physical app
idea. The system they were constructing was inherently interactive and thus hard to capture in a single sketch or diagram. The advantage of storyboarding soon became clear: several designs can be explored in a short space of time without the expense of building and programming them. The act of formalising a sequence of events and interactions can also be helpful in making a learner's ideas more concrete. After some time was spent developing several storyboards, all storyboards were gathered in. At random, a storyboard was selected from the stack. The learner responsible was encouraged to share what problem s/he was solving and how the design would solve the problem. The storyboard serves as a link to the learner’s description. Feedback can then be received from the rest of the learners. This further shares ideas between the group and encourages individual learners to reflect on the work of others.

At this point the learners were encouraged to form groups and attempt to build one of their physical app ideas. The group formation was left entirely in the hands of the learners. If it appeared that an individual was having difficulty getting into a group, however, they would be offered support if they desired it. Throughout the four sessions, a total of 24 physical apps were developed by ten individuals, seven pairs and seven groups. Throughout the building time, learners were left to work independently. The tutor and two additional helpers were available to offer support as groups required it. Their role was clearly defined as one of facilitation. Learners were to pursue their own ideas and facilitators were on hand to support and tune (where necessary) these ideas to fit within the confines of the workshop, the equipment available and the available time. For example, one group was keen to make a model police car as a toy for a younger brother. Initially the team hoped to make a car that could drive about. The facilitator talked through the technical challenges that this presented and re-focused the team on more achievable static model with lights and sound. The key to facilitation is including the learners in the decision-making process.

To allow participants to construct convincing prototypes, they were given access to various craft materials including balsa wood, modelling clay, foam board, hot glue, various marker pens and a selection of card. Many of the physical apps made use of these materials to embed the technology in a physical form. When the build was concluded, the learners were asked to complete the knowledge and understanding post-test.

The final activity involved getting the individuals, pairs and groups to share their physical apps. Given the time and materials, the physical apps were best described as working prototypes held together with tape, hot glue and blue tac with an Arduino at the core. Across the four sessions, the learners explained very well what they had made and why. This was well received by the other learners. Before the workshop ended, participants were asked to complete the emotional response questionnaire, making a first pass to indicate what emotions had cropped up throughout the day and then going back to offer some contextual detail.

3.3 Study Design
A combination of quantitative and qualitative methods was employed to evaluate this workshop. Questionnaires were designed to measure changes in knowledge and understanding, and emotional response. In addition to the questionnaires, digital photographs and videos of each of the completed physical apps were captured for subsequent review and analysis. The following section describes in detail the study design decisions made for each of the following parts of the study: measuring change in knowledge and understanding, measuring emotional response and understanding the sophistication of the physical apps.

As this study involved human participants, ethical approval was sought and granted by the University of Dundee’s School of Computing Ethics committee. As part of the digital making initiative organised by Nesta, the study did not present any significant ethical dilemmas: it was not replacing or affecting statutory education; it involved neither working with vulnerable participants nor misleading participants. Participants were recruited by Nesta, which also enforced the only exclusion criteria, which was that participants were over 16. The study design used a variety of pre- and post-measures and participant observations that were considered appropriate and in keeping with the inquiry described to the Ethics Committee. Informed consent was obtained from all participants, as was approval for use of digital photographs and videos for research purposes.
3.3.1 Knowledge and Understanding
The basic measurement of this part of the study is to investigate if a change in knowledge and understanding (KU) has taken place in the individual learners as a direct result of taking part in the activities involved in making a physical app. A pre/post-test method was adopted to obtain a baseline of learner KU and then a post-test of KU was used at the conclusion of the study to determine any alteration from the baseline. The only inclusion criteria and prior information about the learners was their age and the fact they had self-identified an interest in the workshop, based on the event description.

The questionnaire designed contained eight multiple-choice questions: four were knowledge-related and four pertained to understanding. The learner was presented with a statement demonstrating an element of programming knowledge or understanding, and given the opportunity to indicate whether it is true or false. In addition, the opportunity to indicate they do not know was also given as a strategy to reduce guessing by learners when knowledge is not present. When scoring answers, a learner received a point for a correct answer, lost a point for an incorrect answer and received no point for selecting not sure. This was to disambiguate between misplaced knowledge and absent knowledge (Perkins and Martin, 1986). This also offers the ability to see where a learner has transitioned from no knowledge to correct knowledge and when a learner has transitioned from incorrect knowledge to correct knowledge.

3.3.2 Emotional Response
Learners’ emotional response was measured using the Reflective Emotional Index described in (Martin et al., 2017a). Based on the HUMAINE project work (Petta et al., 2011), it encourages learners at the conclusion of a learning experience to reflect and identify emotions they may have experienced and the intensity of each of these experienced emotions. In addition, space for free text response is given to offer insight as to why and when a particular emotion was felt by the learner. This is an important part of the study as it offers an insight into the learner's response to the activity in the circumstances of a reasonable degree of freedom in what they are building and working towards. The learner's emotional response to the activity is a good indicator of the value of this freedom.

3.3.3 Review of the Physical App
At the conclusion of each session, each group presented its completed physical app, describing what it was, whom it was for and what problem it aimed to solve. These presentations were videoed to provide a persistent record of each of the physical apps created. Videos then were reviewed and a qualitative analysis performed to derive an understanding of the sophistication of the different builds using a card sort system. The previous measures (knowledge and understanding and emotional response) related to the learner's internal experience of the activities associated with creating a physical app. A further measure, ‘sophistication’, scrutinises the product of the learner's efforts with awareness of the context for which the learners are creating the app. In many ways, ‘sophistication’ captures the extent to which learners have embraced learning to program in a rich context as well as being able to use the taught skills successfully. As a measure, it considers three elements: (i) does the idea consider the context and for whom the app was being built, (ii) does the build have a strong aesthetic and (iii) is the physical app technically complex.

4. Results
The physical apps study engaged 48 young learners, with a range of experience and different backgrounds from across Scotland. There was a substantial gender bias, with the majority (83%) of the participants being male. The following sections present the findings from the three areas of inquiry described in the study design: measuring change in knowledge and understanding, measuring emotional response and judging the sophistication of the physical app. Finally, the composition of groups will be considered with respect to the sophistication of the physical app.

4.1 Knowledge and Understanding
Knowledge and understanding are perhaps the most measured outputs from a learning experience, and for good reasons. In the simplest sense, the purpose of creating and engaging in a learning experience is to increase knowledge and understanding of a given topic. Thus, the success of a given learning experience is reflected well by an improvement in the learner's knowledge and understanding. In the physical apps study, change in knowledge and understanding was measured by administering the same eight question multiple choice paper test at both the beginning and the end of the study.
Mean knowledge scores improved by 26% and mean understanding test scores improved by 9% (Table 1). The mean post-test score (knowledge and understanding combined) showed an improvement of 17% as a result of the activities undertaken. This is evidence that programming knowledge and understanding has increased significantly as a result of the physical apps study.

<table>
<thead>
<tr>
<th></th>
<th>Pre-test</th>
<th>Post-test</th>
<th>n</th>
<th>t</th>
<th>Df</th>
<th>Sig. (2-tailed)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Std. Deviation</td>
<td>Mean</td>
<td>Std. Deviation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Whole sample</td>
<td>39.52</td>
<td>25.09</td>
<td>56.73</td>
<td>21.24</td>
<td>39</td>
<td>-5.05</td>
</tr>
<tr>
<td>Knowledge</td>
<td>46.79</td>
<td>34.97</td>
<td>72.79</td>
<td>25.12</td>
<td>39</td>
<td>-4.89</td>
</tr>
<tr>
<td>Understanding</td>
<td>32.05</td>
<td>22.90</td>
<td>41.47</td>
<td>26.49</td>
<td>39</td>
<td>-2.65</td>
</tr>
</tbody>
</table>

Table 1: Digital Makers Results

4.2 Emotional Response
Emotional response was captured with the Reflective Emotional Inventory, with responses being grouped via ten high level headings. The first five are negative emotions and the second five are positive emotions. The responses were collated and normalised to an intensity range from 0 (emotion not felt) to 2 (emotion felt intensely). The most striking result was that positive emotions were reported as being far more intensely experienced than negative emotions (Figure 1).

![Figure 1: Strength of Emotional Responses](image)

4.3 Review of the Physical App
Across the four sessions, 24 different physical apps were developed, responding to a wide variety of ideas. Each of the apps built was reviewed for the elements of idea, build and complexity. The elements were judged using a structured framework that involved rank ordering the apps for each element and identifying different levels in each set according to the characteristics of that category. Aspects of these elements are considered next.

**Idea**: this considered the quality of the physical app idea and its purpose. The top performers (Figure 2, noting the group size and gender balance) demonstrated a robust acknowledgement of the brief: to build a physical app based on one of the earlier ideas suggested by each group. They provided evidence of being derived from a problem rather than a technical capability and by being a novel idea with strong links to specific users and/or context with a resolved sequence of interactions. App 16 was a homework progress monitor with a clear consideration of user and good resonance with the brief. This also solved
a problem the learners personally identify with. App 23 was a physical alarm clock that starts with a small noise that gradually increases in intensity, then progresses to a physical arm that bashes the sleepy person until s/he wakes. This was a response to the challenge of getting out of bed for school after the holidays, put on a post-it in the morning.

Figure 2: Idea – Top Performers

App 4 provided a single purpose audio-visual notification of a Facebook notification. This was in keeping with what they had been asked to do and demonstrated the potential need for more eye-catching notification. App 1 provided a performance-rewarding app that quite literally gave you a pat on the back and also kept note of it with an LED bar graph. The final app in this category, App 9, was entitled the FIFA notification station. It had a moving physical arm that drew attention to new information pertinent to a computer game that was popular among the group. The learners spent a great deal of time choosing specific items of information and mocking up the visual display with marker pens on the foam core board. The common feature across all top performing apps was that they are driven by an idea rather than technology. The physical app told a story of who it was for and why it was for them. In most cases the user was the learner, which meant they were building something that related to them whilst they learn about physical computing and what it entails. The weaker apps were driven by technical expertise and were simply an assemblage of the learner's newly acquired technical skills, with little thought of purpose or for users.

Build: this was based on the non-technical physical components of the physical app and the effort given to construction. This took into account use of materials, structures created and the aesthetics. The strongest apps had a refined and complex build and made good use of a range of materials and consideration of aesthetics. A top performing app (Figure 3) was defined as a refined and complex build that made good use of materials and had considered aesthetics. App 3 was a police car with light and sound. The learners had spent a great deal of time embedding the electronics in a model that offered a good representation of a police car. They made use of a range of appropriate building materials. App 2, the robot toy, was constructed with an array of components cleverly embedded in a convincing robot model. Where appropriate, such as with the servos, the hardware component was concealed. Where it could add to the appearance of the build, such as the speaker mouth, it had been made visible. In addition, a novel construction approach was used to create an articulated abutment join with the foam-board. By inserting straightened paper clips, the learners were able to attach appendages with some flexibility. The bottom level apps of this element failed to function or had not augmented the electronics with any modelling at all.

Figure 3: Build – Top Performers

Technical Complexity: this refers to the complexity of the build with respect to the code and hardware components used (e.g. servos, LEDs and buttons). The best in this element was expected to have a well-integrated combination of the taught examples and to contain an element of novelty or an extension of the taught concepts. The only app in this category was App 16 (Figure 4). This group identified that there were issues with using servos and pulse width modulation at the same time. To combat this, they made use of multiple Arduinos and used serial communication, a technique not taught, to allow the
boards to communicate with each other. The app also included multiple components and good integration of a range of taught material all reliably functioning.

**Group composition:** For the morning taught component of the workshop, learners worked as individuals, with a 1:1 ratio of equipment to learner. In the afternoon, when learners had the opportunity to build a physical app of their own design, they were left to form groups, pairs or work as individuals as they desired. As a result, there was a rich mix of groupings. There were eight female learners and forty male learners across the four sessions. Of those 48, ten learners opted to work as individuals (all male), seven formed pairs (one female and six male), four formed groups of three (one mixed and three male) and three formed groups of four (two mixed and one male).

To be in the top 20% of apps, the learners had to demonstrate excellence across the range of skills and notably all had identified excellent ideas. Of the four in this category, three were individual male learners with the top place going to a mixed group of four. All apps in this band were robustly demonstrable. With the exception of the top-placed group, small teams and individuals performed best overall. Further research with a greater number of learners is required to explore this finding further.

### 5. Discussion

#### 5.1 Knowledge and Understanding

In addition to the high degree of engagement and the emotional response described, there was an observable learning effect. The Digital Makers study resulted in an observable increase in knowledge and understanding amongst the learners that participated. This goes some way to confirm that the additional task given to learner to flesh out the programming did not obstruct the intended aim of supporting engaging programming learning.

#### 5.2 Emotional Response

The emotional response from learners was similar to that of Martin et al., (2017a). Learners reported negative emotions such as frustration related to bugs in code and wiring, and, the strength of the positive emotions was notably greater than that of negative emotions. It is proposed that the error-prone nature of programming will always result in frustrations. The skill in designing a learning experience is to support this rather than to attempt to remove it. Facilitating a learner through the resolution of a software or hardware bug sets them up with some skills of value beyond programming. It is also important to ensure learners do not reach a state in which they have invested significant effort without feedback, as this may result in wasted effort and potential damaging frustration. An example would be wiring up five identical components before confirming that the first one is wired up correctly. This is a tricky balance to achieve.

#### 5.3 Review of the Physical Apps

**Idea:** there was a mix of outcomes regarding how learners engaged with placing their app in context and relating it to a real world problem. The majority of learners made a good attempt to contextualise their creation, many found resonance with the brief to a greater or lesser degree. One of the most interesting findings is that of the 24 apps only Apps 12 and 18 were devoid of any idea, consideration of users or a sense of place in the world. This suggested that for the majority of learners the rich real world framing of the task was valuable. The two apps created with no context were produced by capable programmers who were demonstrating that their interest lay in the technology and they did not desire the distraction of a wider sense of place in the world for their work. Whether this is a good or bad quality in a programmer is a topic for debate elsewhere. There is evidence that the majority of learners engaged well in programming within the bounds of a real world application.
**Build:** Most of the learners took great pleasure in engaging in the opportunity to use various craft materials as part of the physical app challenge. The top two-thirds of the sort had learners demonstrate the ability to create mechanically intriguing and aesthetic solutions to give their physical app function and form. Of the bottom third, many learners attempted to engage with the craft materials but lacked the necessary skills to successfully construct the structures they desired. It is important to note that craft skills were not explicitly taught in the workshop, although assistance and advice were provided in response to requests. Only three participants did not attempt to engage with the building materials. All were technically competent programmers who most likely viewed this as a distraction from their primary interests. To return to the central premise of this study, the build aspect is to stimulate the learner to take up the challenge of programming. In the case of the three apps where learners chose not to engage with the build aspect of the task, there is little doubt that they avoided the craft as a direct result of their hunger to pursue the challenge of programming with which they had proficiency.

**Complexity:** this measured the technical competence demonstrated by the learners as they responded to the challenge of developing a physical app. In many respects, this had most in common with the traditional measures of success such as change in knowledge and understanding and the ability to demonstrate what was learned. The top two-thirds of learners were able to demonstrate the ability to apply the morning’s tuition in novel circumstance to meet the needs of a self-directed challenge, which is quite an achievement. They mixed and adapted examples to solve their own problems. As for the bottom third, seven out of the nine were able to reproduce examples from the morning and attempt to frame them. Only two learners were unable to produce a working app by the end of the workshop.

**Group composition:** each of idea, build and complexity will be considered in turn,

For **idea** generation, individuals performed well with around 70% of apps created by individual learners appearing in the top half of the idea card sort. The larger groups with four participants were also in the top half of the idea performance. Groups of two and three performed less well in the idea generation stage. Individuals performing well in an idea-generation process perhaps may be unexpected. This may be explained by the large group idea-generation activity that took place before the build. This allowed the whole group to consider and externalise possible app ideas. Individuals may have found it simpler to select one of these ideas, whereas the medium-sized groups spent time negotiating.

For the **build** sort this was flipped, with individuals performing less well. For build, around 70% individual learners were in the bottom half of performance. The pairs, trios and quartets all performed well, which is perhaps unexpected. It is likely that a well-functioning group with more members offered a good opportunity to delegate tasks. Building the physical model was also an activity that could take place in parallel to code generation, thus favouring larger groups.

For **complexity**, pairs and individuals perform best with 60% of individuals in the top half and over 50% of pairs in the top half. None of the trios and only one quartet made it into the top half. This is unsurprising, as many of the more complex apps were produced by technically capable individuals who were focused on exploring the hardware and software. These individuals were less engaged in group work or in placing their hardware in a specific context.

6. **Limitations**

There is always a need to apply pragmatism when designing a study involving education. In this study, the single observer issue could be described as a limitation. A further limitation is the possibly untruthful or poorly remembered self-reporting of emotions experienced. However, there is no evidence of any learners reporting their emotions other than as best they could. A stronger candidate for consideration as a limitation is the balance between sample size and sample bias. This study has a good sample size, with a rich set of data from 48 learners across Scotland. However, it does suffer a sample bias as a result of being piggy-backed on to a national public engagement event. It is therefore important to reflect on this when considering the results. The sample was taken from across Scotland for a bookable event. Participants who attended were likely to have a good awareness of what they were attending and therefore likely to engage fully and perform well. This has been evidenced from the various measures taken throughout the study where the majority of learners have performed well. These
methods will need to be developed across mainstream education to test if the results described generalise to the broader population of young learners.

7. Conclusion
The main findings of this study can be summarised by referring to the research question: “How is introductory programming learning affected by designing activities that enable personalisation, ownership and purpose?” The Digital Makers study used ownership, personalisation and purpose to create a highly engaging learning experience that resulted in an evident learning effect and strong positive emotional responses from learners.

The qualitative evaluation of the physical apps gives the best insight into the effect of the rich context that was built around the intended learning. Reflecting on a learner’s app, ideas, builds and complexity offers a good indication of the extent to which that learner embraced this approach to learning. Almost all learners engaged very well with this approach to learning, showing commitment to solving the problems they had defined. The ideas for physical apps reflected the culture the learners belong to, such as with notifications for FIFA and Facebook. In some cases, they personalised their product by creating toys for younger siblings (such as a police car with lights and sound). Almost all learners attempted to construct some kind of physical app using the craft materials provided. The complexity of the builds varied from relatively simple extensions of the demonstrations to complex compositions with multiple sensors and actuators. The only learners not to engage fully with the rich context were learners that had come to the session with an existing knowledge of Arduino and had premeditated plans for what they wanted to explore. For these learners, the rich context was a distraction to their intentions.

A combination of quantitative and qualitative methods has provided insights into the value of ownership, personalisation and purpose to design engage learning experiences in programming. Qualitative methods were used to judge the work of learners creating physical apps and to capture their emotional experiences immediately after a programming experience. The former provided evidence of the very positive emotions experienced by learners developing physical apps. The latter allowed identification of links between the emotions experienced and their origin. The empirical evidence suggests that the majority of learners engaged deeply in the rich context in which the learning was situated. This was coupled with a strong learning effect. Jointly, these have helped to confirm the value of creating a highly engaging learning experience that includes a layer of cultural relevance.

8. Acknowledgement
Thank you to NESTA for supporting this study.

9. References


Programming Education to Preschoolers: Reflections and Observations from a Field Study

Alaeddin Swidan and Felienne Hermans
Delft University of Technology
{Alaaeddin.swidan, F.f.j.hermans}@tudelft.nl

Abstract

In recent years, there has been a rise in methods and tools dedicated to programming education for children of primary school age. In this paper, we present our experience of providing five programming sessions to a group of eleven children between four and six years. Our sessions followed problem-solving and game-playing themes and featured two newly-developed tools: the unplugged Robot Turtles, and the robotic Ozobot. The activities embed programming concepts such as the order of operations, symbolic representations, and functional abstraction. The observations show that children understood and applied concepts such as sorting, sequential operations, and functional abstraction. However, children struggle with giving directions to the object which highlights a spatial awareness limitation. Finally, we link the observations to Piaget’s theory and his limitations to thoughts for children in this age. We find that some of Piaget’s limitations such as egocentrism can explain a few observed behaviors. However, a few limitations contradict our observations such as the irreversibility and transductive reasoning.

1. Introduction

Computational thinking education for children has been developing for decades (Papert, 1980; L. Morgado, Cruz, & Kahn, 2006). The aim is to make computer programming, a core aspect of computational thinking (Ambrosio, Almeida, Macedo, & Franco, 2014), widespread and more accessible. Many programming systems and languages were developed primarily for children as young as four, in schools and kindergartens (L. C. Morgado, 2005). The majority of these were computer aided (Kelleher & Pausch, 2005). For older children, Scratch has recently become a favored platform (Meerbaum-Salant, Armoni, & Ben-Ari, 2010; Hermans & Aivaloglou, 2017). Unplugged techniques, i.e., away from computers were also designed ¹. With these tools in place, the focus is still to a great extent on developing more systems and tools for children, and to incorporate them into school curriculum (Fessakis, Gouli, & Mavroudi, 2013). However, few studies investigate the children cognitive and social reactions to these educational tools (L. C. Morgado, 2005).

To that end, we provide programming education sessions to eleven children between four and six at a primary school in the Netherlands. We use both unplugged and robotics materials to teach concepts such as the order of operations, symbolic representation, and functional abstraction. By guiding the children throughout these sessions, we aim at observing their reactions while using these educational tools and what difficulties they face in completing the tasks. We then compare our observations to Piaget’s theory (Piaget, 1964), one of the dominant theories in the psychology of educational development (Mitchell & Ziegler, 2013). Results show the children’s ability to perform sorting and classification tasks, to order the operations sequentially and to use a basic level of abstraction. On the other hand, children have difficulties with spatial awareness needed to give directions to the objects within the games. For the teacher, it is recommended to establish an active learning environment with a high level of engagement with the children. Finally, we find that some Piaget’s limitations to thought, such as egocentrism, can help in explaining some of the observations. In other cases there are opposing observations to a few limitations such as irreversibility and transductive reasoning.

¹http://csunplugged.org/
2. Background: Preschoolers Education

In this section, we provide an overview of constructivism, one of the most established theories in educational and development psychology. We follow by highlighting a few aspects of Piaget’s theory as a major influencer in constructivism.

2.1. Constructivism

In the psychology of learning, constructivism is one of the dominant theories (L. C. Morgado, 2005; Mitchell & Ziegler, 2013). Derived from the word construct, constructivists assert that students and learners acquire new knowledge by actively processing sensory data against their previous knowledge. Ben-Arie (Ben-Ari, 1998) provides a comprehensive revision to constructivism and its application in education, with a focus on computer science. In summary, constructivism promotes an active learning environment combined with exploration and discovery activities. The role of the teacher is important to guide this process by designing the activities, and assisting the students especially when conflicts occur between the new data and previous knowledge. In his revision, Ben-Arie provides examples showing that constructivism is applied in mathematics and physics education, but fewer efforts in CS education.

2.2. Piaget’s theory

There are a variety of theories under the umbrella of constructivism (L. C. Morgado, 2005). We highlight Piaget’s theory (Piaget, 1964) as it is one of the major influences in constructivism. His theory focuses on the cognitive development of children. Piaget theory proposes four stages of cognitive development: sensori-motor, pre-operational, concrete operational and formal operational (Piaget, 1964). In his theory, new information directed towards the child is either assimilated according to the already existing cognitive models or accommodated by trying to build a new cognitive construct (McLeod, 2015). A state of equilibrium occurs in between these two processes where no new knowledge is required. Even though the child, through personal experience and social relations, is responsible for this mental process, the environment is still recognized as a playing factor in his theory (L. C. Morgado, 2005). The teacher as part of the environment is the motivator to the learning process by creating situations which cause the child to be in a disequilibrium state, and allow new cognitive constructs to be developed. The teacher’s role expands to provide helping information, asking questions and making comparisons for example. In this paper, we investigate children in the preoperational stage of development, which Piaget defines between two and seven years. Children in this stage are expected to develop language, complete operations, solve one-step logic problems (McLeod, 2015). More complex logical thinking starts to develop only after the age of seven. During the preoperational stage, children suffer from many limitations to acquiring knowledge, which Piaget calls “limitations to thoughts” (L. C. Morgado, 2005). Table 1 summarizes these limitations in the preoperational stage.

3. Study Goal

In this paper, we describe our efforts to teach programming concepts to children between four and six years old. Our study aims first at exploring how children in this early age react to problem-solving activities in a programming context. Secondly, we target to observe difficulties that limit the children’s ability to complete an activity or grasp a particular concept. These difficulties might be related to the internal mental structures already developed in children, or to the complexity of the concepts being taught. Finally, we want to compare the difficulty we observe to Piaget’s limitations to thought. In summary, we want to answer questions such as:

- How do children react during these activities? For example: to what extent are they motivated and engaged? Do they enjoy the materials used?

- What limitations to thought from Piaget’s theory on preoperational children still apply in activities related to programming education?

- What variations in performance are observed in concerning age difference?
Table 1 – Piaget’s limitations to thought in the preoperational stage of development. The first three columns are taken from (L. C. Morgado, 2005)

<table>
<thead>
<tr>
<th>Limitation</th>
<th>Description</th>
<th>Example</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centration</td>
<td>The child focuses on a single aspect of the situation, disregarding any other.</td>
<td>John cries when his father gives him a biscuit broken in half. Since each half is smaller than the full biscuit, John thinks he’s getting less.</td>
<td>Agree</td>
</tr>
<tr>
<td>Irreversibility</td>
<td>The child is unable to realize that an operation or action can be reversed.</td>
<td>John doesn’t realize that both halves of the biscuit can be joined to make a full biscuit.</td>
<td>Disagree</td>
</tr>
<tr>
<td>Static thinking</td>
<td>The child is unable to realize the meaning of state transformations</td>
<td>In a conservation task, John fails to realize that the shape transformation of a liquid (from a glass to another) doesn’t change the quantity.</td>
<td>Not observed</td>
</tr>
<tr>
<td>Transductive reasoning</td>
<td>The child doesn’t employ either deduction or induction; going instead from one particular aspect to another, seeing a cause where there is none.</td>
<td>“I had ill thoughts about my brother. My brother got ill. So, I made him get ill.” Or: “I misbehaved, so mum and dad divorced.”</td>
<td>Disagree</td>
</tr>
<tr>
<td>Egocentrism</td>
<td>The child assumes that everyone thinks like he/she does.</td>
<td>Mary picks up a game and tells her mum, “This is your favorite.” She’s assuming that her mother likes the game as much as she does.</td>
<td>Agree</td>
</tr>
<tr>
<td>Animistic thinking</td>
<td>The child sees life in inanimate objects.</td>
<td>Mary thinks the clouds are alive because they’re moving.</td>
<td>Agree</td>
</tr>
<tr>
<td>Inability to distinguish appearances from reality</td>
<td>The child mistakes appearances for reality.</td>
<td>John thinks that a sponge made to look like a rock is indeed a rock.</td>
<td>Not observed</td>
</tr>
</tbody>
</table>

4. Setup
To achieve the goals mentioned in the previous section, we provide five sessions to eleven children aged between four and six (five girls and six boys). The sessions include problem-solving activities in a game-playing theme. Our approach is influenced by the main principles of the constructivism: an active learning environment, with a guided exploration and immediate feedback. We follow an observational methodology where two to three supervisors guide and monitor the children’s activities in each session. After the sessions, we discuss the observations and write the ones that are agreed upon by two supervisors. The sessions are held in a Dutch primary school. The children are part of an after-school club, and activities are performed in their daily classroom, a familiar and friendly environment. The children work in groups of two or three. When arranging the groups, we make different combinations in each session; sometimes based on gender or age differences and sometimes purely random.

5. Materials
We used both unplugged and robotic tools, in a game playing setup. Previous research showed the positive effect of game playing activities in the education of computational thinking and programming(U.S., 2011). In addition to the constant problem-solving theme, the exercises embed a variety of programming concepts, such as the order of operations, the symbolic representation, and functional abstraction. See Table 2 for the full list of the exercises, materials and their associated programming concepts. Following is a detailed description of the tools we use.

5.1. Sorting and Classification
This session was the first we did with the children. It was a session with light activities because we aimed at introducing ourselves to the children and getting to know each other. By doing this, we create a friendly environment where children feel free to act upon and express their thoughts. The session includes papers with printed animal pictures, and we ask the children to perform various sorting and classification activities. For example, to sort some animals based on size or color. Another example is to classify animals based on the habitation such as farm, wild and sea animals.
Table 2 – Summary of exercises performed during the sessions showing the programming concepts which the game and the exercises collectively serve

<table>
<thead>
<tr>
<th>Session Order</th>
<th>Programming Environment</th>
<th>Exercise</th>
<th>Overview</th>
<th>Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Printed animal pictures</td>
<td>Arranging cards based on some criteria</td>
<td>Sort the animals based on size, Classify the animals based on their habitation.</td>
<td>Data sorting and classification</td>
</tr>
<tr>
<td>2 and 3</td>
<td>Robot Turtle</td>
<td>1. An Empty maze with diamonds in the center</td>
<td>Give directions to the turtle in relation to the diamond</td>
<td>Problem analysis, Symbolic representations, Order/sequence of operations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. A maze with obstacles</td>
<td>Implement a workaround based on the obstacle’s type</td>
<td>Abstraction (functional)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. A maze with obstacles #2: Full path constructed</td>
<td>Without moving the turtle, the child need to complete the whole path.</td>
<td>Fault isolation and debugging, Team working (pair programming)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4. A maze without obstacles, only with diamonds, with the need to use the frog card</td>
<td>Use the frog card to help the frog to reach the diamond faster</td>
<td></td>
</tr>
<tr>
<td>4 and 5</td>
<td>Ozobot</td>
<td>Explore with the Ozobot</td>
<td>Explore the basic concepts of the Ozobot: sensor-based autonomy following a line, reflections of colors, and the behavior at line branches</td>
<td>Problem analysis, Symbol interpretation, What-if analysis, Alternative branch concept (visual), Team working (pair programming)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Use the color codes in two tracks, so that the Ozobot can get from home to school, and from home to shop</td>
<td>Choose the appropriate color codes</td>
<td></td>
</tr>
</tbody>
</table>

5.2. Robot Turtles

Robot Turtles is a board game that “teaches programming to kids”\(^2\). The board game, shown in Figure 1, was invented by Dan Shapiro, a software engineer. According to the game’s website, it aims at allowing preschoolers to learn “the fundamentals of programming while they are playing”.

The game resembles a simple visual programming language, with the cards being ordered by the child player to direct the turtle through obstacles until it reaches the diamond. The adult who demonstrates the computer processor moves the turtle depending on the order of the cards. The game, as a result, features principles of simplicity, visualization, simulation and autonomy, which are recommended when considering children and novice programmers (Du Boulay, O’Shea, & Monk, 1999; Jenkins, 2002). In addition to the basic direction cards, the game has some special cards. The laser card melts down the ice obstacle, the frog card abstracts a function that can help the turtle performs repeatable actions, and the bug card by which a player breaks the execution and notifies the adult that a problem exists within the sequence of cards. Looking back at programming concepts, we notice that the Robot Turtles game focuses on some of them. In particular, the order of operations (through the sequence of cards), function abstraction (through the frog card), debugging (through the bug card) and code refactoring/optimization (through using shorter paths i.e., less amount of cards). We used the Robot Turtles for two sessions, during which we gradually introduced more game features and programming concepts. We additionally increased the difficulty of the maze the turtle has to solve. In total, we had four exercises where the children performed the tasks on two copies of the game in parallel. See Table 2 for an overview of the exercises.

We introduce the game to the whole group: describe the basic cards with their action. We then work

\(^2\)http://www.robotturtles.com/
closely with the children to repeat some information. We perform the same collective description for each new exercise; describing the goal and introducing new cards. Within the same exercise, the difficulty level was unified: children have the same mazes. We do not know in advance what are the limitations per child; in fact, this is an issue we want to observe.

5.3. Ozobot

The Ozobot is “a miniature smart robot that can follow lines or roam around freely, detect colors, and can also be programmed”. The Ozobot can be programmed through a set of color codes, to perform a variety of actions: speed, direction, and funny moves. The Ozobot is designed to randomly choose directions when facing a line branch or a junction. According to the Ozobot website, the Ozobot empowers the STEM (Science, Technology, Engineering, and Mathematics) education, and can be used to teach subjects such as programming. Regarding age, the Ozobot is designed for all levels of the elementary school. For younger children, it is not advised without the accompany of an adult, primarily because of its small size. Similar to other studies which used robotics in programming education (Magnenat, Shin, Riedo, Siegwart, & Ben-Ari, 2014; Kaloti-Hallak, Armoni, & Ben-Ari, 2015), Ozobot can be used to develop certain programming concepts. These concepts include the analytical problem solving, sequential programming skills, and understanding of computer machine autonomy. See Table 2 for the concepts associated with the Ozobot. We used five Ozobots in two sessions. Each Ozobot was controlled by two to three children. We used the Ozobot Basic Training lessons and resources provided by Ozobot. The first session was more exploratory to the features and nature of the Ozobot. Children learned more about its tracking the lines and colors, but also about the randomness and indeterministic decision at a junction. Also, they learned about the color codes, and they explored their possibilities in movement, speed, and directions. In the second session, the children were asked to solve two exercises which require choosing specific color codes in order to get the Ozobot from a source to a destination. Figure 2 shows three solved exercises stemmed by children in our sessions.

---

Figure 1 – The Robot Turtle game: (a) an empty board (b) a board with a maze (c) one solution to a Robot Turtle exercise in the classroom (d) the basic movement cards

Figure 2 – Ozobot exercises performed by children throughout the sessions

---

5http://ozobot.com/
4https://education.microsoft.com/Story/Lesson?token=qgSYB
5http://portal.ozobot.com/lessons
6. Observations
In this section, we list six observations: behaviors of the children during the sessions. For a behavior to qualify into an observation, it needs to be demonstrated by at least two children and observed by two of the supervisors.

Observation 1: Children suffer when giving directions especially with different viewpoints
One of the most recurrent mistakes in both games is the choice of a wrong direction to the turtle or the Ozobot. Typically, the direction decision involves two objects: where the child is looking initially, and where to turn/look finally. However, in these exercises, the moving objects were not the child; they were the turtle and the Ozobot. Figure 3 shows two different boards where the turtle has a similar and a different viewpoint. This behavior is associated with a computational cognitive skill known as spatial reasoning, and it defines “the ability to recognize and view figures, and the ability to rotate or follow the movements of figures” (Ambrosio et al., 2014). In their study, they considered it as one of “the crucial dimensions for introductory programming”.

Observation 2: Children get derailed from a task by fantasies they connect to
Children find themselves immediately drawn to joys and fantasies familiar to them found in these games. For example, quite a few children (mostly boys) felt very enthusiastic when we introduced the laser card in the robot turtles exercises. They started to point it to each other making a sound imitating the laser hitting other objects. One boy later in the exercise asked if he can put the laser card before reaching the ice obstacle because the laser can transfer and melt from a distance. In the Ozobot exercises, it was more tempting for children to get away from the original task because of the many temptations involved: robots, funny and fast moves, and coloring. Almost all of the children were amazed when we introduced the Ozobot. Some kept the amusement for the overall two sessions. They enjoyed the idea of having a moving robot, and tried to control it by voice commands like “move” or “go left” and ordering it to be colored. Some children got their hands on the coloring markers quickly and immediately started coloring on their own. When it comes to color codes, many children mostly boys wanted the Ozobot to go fast and make funny moves all the time regardless of the problem in question.

Observation 3: Immediate execution helps children better grasp game’s rules and identify faults
The immediate feedback of the computer (the adult in Robot Turtles and the Ozobot) was essential for children to understand the symbolic meaning behind the games’ material. In the case of Robot Turtles, we simulated sounds, when the turtle moves, to send confirming messages to the children about the meaning of some actions and obstacles. For example, the turtle can push a box, but not a wall. Even though we told the children about the meaning verbally, some children made mistakes and tried to move...
forward through a wall. By making a special sound and visual appearance of difficulty to move a wall, children made fewer faults in the next tasks. The exercise of the full path creation at once showed that the performance, when the turtle starts to move, motivates the child to anticipate the fault in the card ahead. As soon as the turtle moves, we can see that some of the children detect the error just before the adult executes the subsequent action. They then try to intervene and fix it with the correct card. Adding the right card, however, did not cause the children to adjust the sequence of cards which follow after the fault. They again waited for the simulation of the turtle and then react, and so on.

**Observation 4: Children comprehend functional abstraction in Robot Turtles faster than expected**

We introduced the concept of the frog card as a helper to the turtle, to move faster through jumping some ordered actions. We tried to be very clear in the explanation, giving them examples. We found the children to be reasonably fast in understanding the application of a frog (function) card in the main sequence. Having said so, we highlight some concerning observations in this area. All of the children chose to apply a single-type operation for the frog function, which is the moving forward card. They wanted the frog to move the turtle two or three steps ahead. In response, we provided thought-provoking questions after they completed the task like “what other cards you could have used”. Another concern is that some children placed the cards for the frog i.e., defined the function, but they did not use it within the main sequence of cards. When notified by the supervisor “Oh I don’t see a frog!” the children eventually placed the card, only in one location in the main sequence. We provided them with additional suggestions, and only then few children got the idea that another place in the main sequence is possible for a second frog card call. Finally, we observed some variances in the speed of getting the idea and the quality of its application among the children. These differences relate primarily to the age group of four years old and will be discussed later in Section 7.

**Observation 5: Adult-supported and active learning environment is a necessity**

Limiting the role of the adult in such experiments to the activity designer, and imitator of knowledge does not suit the young children. These children have different interpretations to a single information outspoken by adults. They might not understand it or might accommodate it with the incomplete knowledge they already have. Thus it is important for adults to discuss individually or in small groups of the children what they think about a particular aspect of a task or a concept. We often did this by asking questions in the form of “What do you think…”, “Why did the object behave in a specific way…” and “What if you did this action? what will happen?” The environment of the classroom was an additional support to children: it is their daily place to learn and play, they feel very confident and relaxed moving around and using available resources. We saw little influence to learning from peers. Time was not sufficient for one group to be more knowledgeable than the other group. In one occasion a child who finished his robot turtle exercise tried to give his peer group the answer to their next card. Unfortunately this was the wrong direction card, and fortunately, the recipient boy thought for a second and chose to stick to his original card. The influence, in this case, would have been adverse to the solution of the problem. We suspect this is because the intervening child lacked the proper spatial awareness, and so he rushed with the wrong card.

**Observation 6: Children often do not express their sense of difficulty or lack of understanding**

Children will go directly into action after explaining the task. In one case only we saw a six years old girl approaching us saying “what do you mean by this” in referring to the final Ozobot exercise. This question came after explaining the task to all children. In another area, children seem to prematurely assess the difficulty of a game. In one case, a boy of six years old, when introduced with the Robot Turtle exercise in the second session shouted “oh no, not this game again. It is so easy”. However, this boy took the longest time to complete the exercise which followed. In fact, other children said the word easy so often when playing with the Robot Turtle, combined with their fingers naturally pointing the route the turtle should follow to reach the diamond. However, their route building procedure using cards usually did not come as smooth as the path pointing. We cannot tell the exact reason. Research show that young children tend to prematurely evaluate the explanation of an issue based on incomplete aspects (U.S., 2011). In our case, this means that their mind considers the pointing as the challenge, and
then judge the easiness of the game based on it. It can be other things such as a mechanism to avoid embarrassment or showing off among peers. It can be that these children did not fully understand how to build the routes by cards, despite being able mentally to point the route.

7. Observation Analysis
In this section, we provide an analysis of the observations by looking back into Piaget’s theory in general, and Piaget’s limitations of thought for preoperational children in particular. We classify the limitations depending on whether it is spotted in one of the observations or not. Additionally, this classification includes other core aspects of Piaget’s theory which we believe worth mentioning.

7.1. Piaget’s theory in agreement with the observations

7.1.1. Egocentrism
Piaget’s definition of egocentrism is not about improper social behavior. It primarily focuses on the child’s ability to considering the perspectives of others (L. C. Morgado, 2005). He showed, using the three mountains experiment (Mitchell & Ziegler, 2013), that children fail to recognize the viewpoint of an object different than their own. The egocentrism of the children in this age is still an aspect the educators need to consider. In our sessions, wrong direction choices were the most frequent mistakes. In the Robot Turtle, the child has an extra support by colors and flower painting on the cards. Each move is associated with a color, as shown in Figure 1d: blue card is forward, the yellow card is left, and the purple card is right. Also, each card has three different flowers matching the color scheme mentioned above. Despite this kind of visual support, children still suffered in spatial reasoning. The problem is observed more when the child perspective is different than the perspective of the turtle or the Ozobot (see Figure 3). The child will choose the card that agrees with his position in relation to the goal (the diamond or the branch respectively).

7.1.2. Animistic Thinking
According to Piaget, the child in this age gives life to non-living objects (L. C. Morgado, 2005). Observation 2 includes behaviors in common with this limitation, as some fantasies the children brought during the sessions fit under this definition. However, this is not necessarily a limitation towards the cognitive development. In some cases, it is a part of the game theme of the activity, such as making the laser sound and playing with each other using the laser card. While in the case of giving voice orders to the Ozobot, it can be considered as a challenge to the child’s ability to understand the nature of forces controlling the Ozobot movement, and then his ability to apply color codes in the process. It is worth noting that the Ozobot has some similarities with living objects, especially the moving. However, children are also affected by previous knowledge about robots. To handle distractions caused by this limitation, we suggest to allow free time playing with the materials, making it clear that afterward attention shall be given to the exercises.

7.1.3. Centration
Centration is defined as the preoperational child tendency to focus on a single aspect of the problem disregarding the other aspects. Observation 3 includes behaviors that can explained by this definition. For example, the children were more engaged and reacted faster when the turtle was moving, as they introduced the next move’s card quickly. On the other hand, building the full path required an extra step of imagination from the child. Most of the children committed more mistakes in the full path part than when the turtle moved step by step.

7.2. Piaget’s limitations in disagreement

7.2.1. Irreversibility
As stated in Table1, the irreversibility limitation in preoperational children describes the inability to understand that a particular operation or action is reversible. This kind of behavior was not observed during our sessions. We found the children motivated to amend any fault immediately, sometime in runtime. As mentioned in Observation 3, children start to put correct cards instead of the wrong ones when the turtle starts moving in an unintended direction. In Ozobot, less reversible actions were possible because of the permanent coloring. However, some children managed to create extra branches for the
Ozobot in order to overcome one wrong color code, see Figure 2. Nevertheless, this behavior was limited and occurred following a hint from the supervisor.

7.2.2. Transductive reasoning
Piaget defined it as the child’s inability to deduce or induce causes and effects correctly (see Table 1). We found opposing observations to this limitation in our sessions. We note, however, the careful and precise explanations we provided to the children prior to making the reasoning. One example from the Ozobot is related to the branch decision. The ozobot would choose a direction randomly when no color code is provided ahead of the branch. First, we asked the children: “Where do you think the ozobot goes on a branch?” We received rushing contradicting answers “always red”, “always left” or “always straight”. We then let them explore with two branches where we ask them where do you think the Ozobot will go before it moves. Children observed how the Ozobot is not predictable: sometimes they get it right by chance, and many other times they do not. After a few runs with these exercises, we asked the first question again. We got a strange moment of silence, followed by a four-year-old saying exactly what we wanted to hear “We do not know!”. As long as we do not tell the Ozobot, by color codes, where to go, we do not know its decision for sure. This is aligned with findings of (Du Boulay et al., 1999; Fay & Klahr, 1996) who described this as the indeterminate problem. Their results concluded that children in this age should be able to distinguish between deterministic and indeterministic situations following some exercises.

7.2.3. Abstraction, sorting, and classification
Piaget theory believes that preoperational children are unable to develop sorting, classification and abstraction thinking until the following stages: the concrete operational and formal operational stages. Children in our case showed a high level of skill in sorting animals based on attributes as size, color, and habitation. While in the Robot Turtle’s frog card, children grasped the concept of the helping card rather quickly. Before the exercise, we presented the frog card usage as a helper to the turtle and gave examples on how to use it. The application of the frog card into the main sequence, however, was limited. Only one type of movement cards was used in building the functions, the forward card, and it was called in one place in the main sequence.

7.3. Age factor
Despite having a group who are very close in age, we noticed a variation between four and six-year-old children. From what we observed, this difference is primarily related to their focus span being shorter than older children. As a result, four years old children had more tendency to lose track of the task, and get easily distracted by other joyful activities like drawing and coloring. Besides affecting their level of understanding the concepts presented, this sometimes negatively affected their teammates in the group.

7.4. Motivation
Overall the children were motivated and engaged in the activities. This motivation was shown during the free time given to them just before leaving. In the robot turtles session, they engaged in building a maze for each team trying to challenge each other. For the Ozobot they created by drawing and coloring their routes and problems for the Ozobot to overcome. Also, they expressed to their parents the wish to continue playing these games at home.

8. Conclusions
In this paper, we report our experience to provide five programming lessons to children between four and six. We present six observations related to the children reaction and understanding of the programming concepts embedded in the activities. Among these observations, we highlight the children ability to perform sorting and classification tasks, to order the operations sequentially and to use a basic level of functional abstraction. On the other hand, children had difficulties with spatial awareness needed to give directions to the objects within the games. We compared our observations to Piaget’s limitations to thought and showed the egocentrism helps in explaining the spatial awareness observations, while transductive reasoning and irreversibility were opposed by our observations.
9. Acknowledgements
We would like to thank Martine Noesen on her efforts in managing the children and co-supervising during the sessions.

10. References


Scores & Scripts — a Bestiary of Intents

Luke Church
Computer Laboratory
University of Cambridge
luke@church.name

Joana Chicau
Independent artist
joanabcq@gmail.com

Abstract
A choreographer and a language designer collect and comment on a catalogue of notations of intent, each for their own purpose. We offer plural interpretations of their similarity, their purpose, their interpretation. The work is presented as a series of large posters each giving a different perspective of what it means to communicate intent.
The Anthropic Principle (Carter, 1973) argues that for a universe to be observed, it must support a form of life capable of observation. This perspective helps explain the apparent finiteness of the observed universe, as the observable universe is assumed to be the result of a process that is itself the result of natural selection. This mechanism is often referred to as “intelligent design.”

In the context of evolution, the Anthropic Principle suggests that the conditions that allow for the evolution of intelligent life are not unique to our own universe, but rather are a consequence of the laws of physics and the processes of natural selection that occurred during the history of the universe. This perspective challenges traditional views of the universe as a “random” or “uncaused” entity, and instead attributes the existence of intelligent life to a combination of necessary and sufficient conditions that have evolved over time.

The Anthropic Principle also highlights the importance of considering the observer effect in scientific inquiry. This includes considering the role of humans in observing the universe, and the potential for our observations to influence the outcomes of experiments and observations. This perspective has implications for the way we think about the nature of reality and the role of consciousness in the universe.
Tica: An environment for exploring tangible vs. screen-based programming

John Wilkie  
Department of Informatics  
University of Sussex  
jw478@sussex.ac.uk

Judith Good  
Department of Informatics  
University of Sussex  
J.Good@sussex.ac.uk

Abstract

This paper describes Tica, an environment designed to explore the differences between tangible and screen-based interfaces when teaching programming to children aged 5–7 years. Tica comprises several components: a tangible programming language and a screen-based equivalent, an Android app to interpret the programming languages and record data, and a physical robot that can be programmed using either the tangible or screen-based language. Tica was designed using a learner-centred approach, with a specific focus on the needs and abilities of the target user group.

Once implemented, an initial pilot study was carried out with 14 adults. Although participants using the screen-based interface completed problems more quickly than those using the tangible interface, participants using the tangible interface reported a higher level of enjoyment. Next steps include a more extensive study with the target population as well as some refinements to the Tica environment.

1. Introduction

Globally there has been growing concern over the need to educate young people not only to be safe consumers of digital technology, but also to understand how it works and to become creators themselves (Howland, Good, Robertson & Manches, 2015). In September 2013, the English Department for Education made significant changes to the National Curriculum: what was previously known as “ICT” became “Computing Programmes of Study”. As a result, from Key Stage 1 (ages 5–7), children are required to be able to create and debug simple programs and to understand what algorithms are and how to implement them. This has resulted in a large increase in the number of young children being taught to program, leading in turn to an increasing focus on how they are taught. One approach to teaching young children to program has been to use tangible programming languages (TPLs). TPLs use physical objects that represent instructions that can be manipulated to write programs in a physical manner.

Although there is an intuitive sense that TPLs may be more appropriate for teaching programming to young children than more traditional screen-based systems (Wyeth & Purchase, 2003), little research has been conducted to assess the benefits (if any) of tangible programming over screen-based equivalents. The system described in this paper, Tica, was designed to allow researchers to investigate these differences in a controlled environment. Tica comprises two functionally equivalent programming languages: both use programming blocks, however, one interface is screen-based, while the other uses physical programming blocks. Both languages can be used to program a physical robot. Given that care has been taken to remove potential confounding variables between the two languages, it provides an ideal test-bed in which to investigate the potential benefits of TPLs for children.

The rest of the paper is structured as follows: Section 2 provides a brief summary of relevant background work, while Section 3 provides a comprehensive overview of the Tica system. Section 4 describes two preliminary evaluation studies, Section 5 describes future work, with general conclusions drawn in Section 6.

2. Related Work

Tangible programming languages are not a new idea. Perlman (1976) noticed the difficulties that children experienced when learning Logo, and believed that one of the barriers was the user interface. He therefore developed two alternatives to inputting instructions via the keyboard: the ‘Button Box’ (Figure 1), and the ‘Slot Machine’ (Figure 2). The ‘Button Box’ was a set of 4 individual boxes, each box having a different set of buttons to control the turtle. Boxes could be connected together as the child mastered more complex controls.
The ‘Slot Machine’ had long rectangular boxes with slots into which the child could place instruction cards. On the boxes was a button that, when pressed, ran the instructions on the cards in linear order.

Since Perlman’s initial work, a number of TPLs have been created which can be broadly categorised by 2 characteristics:

1) *The method the system uses to identify the blocks:* one family uses computer vision to identify blocks and their order, the other family uses in-built electronics allowing the blocks to uniquely identify themselves.

2) *What the TPLs control:* this can be split into one set controlling robots, and the other set being games running on tablets.

One of the most popular commercially available TPLs is Osmo coding (https://www.playosmo.com/en/coding/), which was originally developed as Strawbies (Hu, Zekelman, Horn, & Judd, 2015) for the Osmo (see Figure 3). Osmo is a system developed to enable tangible play with an iPad. It makes use of a mirror to reflect the area in front of the iPad into the front facing camera and then uses computer vision to recognise pieces placed in this area. With the Osmo coding app, the user can create programs using the tangible blocks. These programs control the movements of a monster character in the screen, who moves around a grid-like level to specified squares, collecting strawberries along the way.

The original Strawbies game made use of the TopCodes computer vision library (http://users.eecs.northwestern.edu/~mhorn/topcodes/). This library recognises ninety-nine unique small circular codes and can return an ID number and the location of each code. Strawbies is not the only tangible programming language to make use of the TopCodes library: Tern (Horn & Jacob, 2007)
and Quetzal (Horn & Jacob, 2006) also use it. Both Strawbies and Tern also use jigsaw-like puzzle pieces, ensuring the user connects the tangible blocks in the manner intended (see Figure 4).

![Figure 4 - A TopCode code (left), which is used by Tern (middle) and Strawbies (right)](image)

One of the most recent developments in the tangible programming world is Project Bloks (https://projectbloks.withgoogle.com), a Google research project which is building a platform to allow others to create tangible programming experiences for children. This will potentially allow those with less technical experience to be able to easily create tangible interfaces.

Various studies have been carried out to compare the differences between tangible and screen-based interfaces for programming. Horn et al (2009) conducted a museum-based study in which researchers set up either a tangible or graphical interface to program a robot with simple instructions, then observed museum visitors. Overall, 260 visitors were observed, and 13 family groups interviewed. Their results suggested that visitors found both interfaces equally easy to use, however, visitors were more drawn to the tangible interface, and it was observed that the tangible interface led to more collaboration between group members. There was no significant difference in the time spent on either interface.

Sapounidis and Demetriadis (2013) carried out a study investigating children’s preferences for a tangible programming environment (T_ProRob) or a screen-based equivalent for programming an NXT Lego robot. Three groups of children took part, aged 5-6, 7-8 and 11-12. Children first indicated their preference for either the tangible or graphical interface, then carried out a series of programming tasks using both interfaces. They were then asked which interface they found most enjoyable (at the midpoint of the study and again at the end), and which was easiest to use. Results showed that the tangible language was more attractive to the majority at first sight, and considered more enjoyable to use. In terms of ease of use, the two younger age groups reported finding the tangible interface easier to use, but the opposite was found for the eldest group. Observations showed that the tangible interface allowed multiple children to manipulate the system in parallel, hence programming was more group oriented than with the screen interface.

Sapounidis et al. (2015) performed similar testing while video recording sessions to document the time taken to complete tasks and the number of mistakes made. Five age groups participated: 6-7, 7-8, 9-10, 10-11 and 11-12 years. Results showed that tasks were completed more quickly with the tangible interface in all but the oldest group. Notably, the divergence in task completion time between the two interfaces decreased as age increased. The tangible interface also outperformed the graphical interface regarding number of errors and success of debugging.

Whilst the number of TPLs is increasing rapidly, there is a paucity of research into their potential advantages. The study by Sapounidis et al. suggests that tangible interfaces provide an advantage over screen based interfaces for younger children, however, in both the Horn et al. and Sapounidis et al. studies, users were only exposed to the interfaces for a single session. The next logical step would be to study the effect of interface type over multiple sessions, using a system designed to minimise extraneous differences between the two interfaces, and isolate the variables of interest. The next section describes the design and implementation of Tica, a system developed for this purpose.

### 3. The Tica System

The Tica environment comprises a number of components: a simple tangible programming language, a screen based programming language, a physical robot which can be programmed using either of the languages (Figure 5, left) and finally, a set of tiles of different types (e.g. ‘start’, ‘finish’) which can be formed into a grid to create tasks for the robot to complete (Figure 5, right). Either language can be
used, independently, to program the physical robot. Apart from the obvious tangibility of the tangible programming language, the two languages have been designed to be as similar as possible.

![Tangible interface and Screen interface](image)

*Figure 5 - The two programming interfaces (left) and a simple task for the robot to complete (right)*

In designing Tica, a full list of functional and non-functional requirements was drawn up for the tangible components, the tablet computer application, and the robot. These requirements included a comprehensive consideration of the needs and abilities of the target users in terms of their cognitive and physical development (e.g., determining the appropriate size of the tangible objects based on typical fine motor skill development, font choices for ease of reading, appropriate use of icons, etc.).

### 3.1. The Tica App

The app provides functionality for a facilitator or teacher to design programming tasks for children and run sessions. It is used to communicate with the robot, create new tasks, and record and view data on task attempts. It also provides the platform used for programming using the screen interface.

![Tica app screens](image)

*Figure 6 - Start Screen (left) and Main Menu (right)*

**Tica**’s start screen is shown in *Figure 6* (left). While this screen is displayed, the system sets up the Bluetooth connection, and any problems making the connection are shown on this screen. The main menu (*Figure 6*, right) is designed for simplicity, and uses icons to support ease of understanding.
The task creation screen (Figure 7, left) allows the facilitator to create a new task by choosing specific tiles for the grid, as well as the instructions needed to complete the task. The system will check if the task is well-formed (i.e. has a single start and finish tile, and at least one instruction), and will alert the user if this is not the case.

Tapping on a task in the task performance screen (Figure 7, right) allows the user to view details on that task attempt (e.g. instructions used), shown in a popup window. The user can also delete all current performance data by tapping the “delete data” button.

The task description screen (Figure 8, left) shows each individual task created by the facilitator, while the task selection screen (Figure 8, right) shows thumbnails of the task grids that have been created to support ease of task selection. If there are more tasks than fit on the page, the area becomes scrollable.
The screen-based programming interface (Figure 9, left) has a construction pane at the top of the screen along with the inventory of instructions at the bottom. Instructions are selected by dragging them into the construction pane. In comparison, when using the tangible interface (Figure 9, right), users select instructions by physically placing them in front of the system. The play button on each respective interface will run the instructions, and a dialog will appear to inform the user the robot is running. Once the robot stops running, the system will either move to the task complete screen if the instructions were correct, or alert the user if they were incorrect.

3.2. The Tica Tangible Language

The tangible blocks were designed using key concepts from existing TPLs, and were created using a 3D printer. Seven instructions were created (see Figure 10): three instructions for controlling the direction of the robot, one instruction to generate noise from the robot, an iteration instruction and two control flow instructions. The condition for the control flow instructions is decided based on whether or not the path of the robot is blocked.

Instructions are identified using a TopCode code. During the development of the instructions a prototype was built using OCR (optical character recognition) in order to identify instructions, but this resulted in a poorer identification rate.

A means was required to alert the Android application to read the instructions. A stand was built that incorporated a “play button” (Figure 9, right). Pressing this button signalled the application to process the current instructions via Bluetooth. The application then takes a photo that can then be processed to
identify the instructions. In order to reflect the image into the tablet’s front-facing camera, a mirror is used. This use of a mirror was inspired by the Osmo (https://www.playosmo.com/en/).

3.3. The Tica Robot
The Tica robot is built around an Arduino Uno prototyping board and uses an Adafruit v2 motor shield to control two stepper motors. Stepper motors were chosen to provide semi-precise, repeatable movements. Communication with the robot is via Bluetooth. The case is a combination of wood and 3D printed plastic. The decision to design the case with a bug-like appearance was two-fold: firstly, so that the front and the back of the robot could be easily identified, and secondly, to seem somewhat less mechanical and more friendly, potentially allowing children to create a narrative around the robot.

Figure 11 – Tica robot

3.4. The Tica Grid
In order to lay out tasks for the robot to complete, a set of tiles was required that could be used to create a grid, representing a task. Five tile types were created (see Figure 12).

Figure 12 – Tile types

The tile with the “S” is the tile the robot starts on. The tile with the finish flag marks where the robot must finish. The tile with the speaker icon is a tile the robot must stop on and make a noise (using the noise instruction). The tile with the cross represents a location that the robot must not enter. The empty tile is a normal tile with no special attributes.

3.5. Overall System Setup
The system can be set up to use either the screen based (Figure 13, left) or tangible interface (Figure 13, right). Switching between interfaces was designed to be quick and simple.

Figure 13 – Screen interface set up (left) and tangible interface set up (right)
4. Evaluation

Due to project time constraints, it was not possible to conduct an evaluation study with the target users (although this is planned as future work). Instead, a pilot study was carried out with adult novice programmers. Additionally, a further pilot study investigated the use of the system from the perspective of the facilitator/teacher. These studies are described below.

4.1 Novice programmer pilot study

The primary aim of this study was to examine the use of the system ‘in the wild’ in order to identify necessary improvements and ensure that the system’s data recording mechanisms were accurate. In addition, the study aimed to gather preliminary data on the relationship between the interface type (screen-based or tangible) and task performance, as well as perceived enjoyment.

4.1.1 Participants

Opportunistic sampling was used to select fourteen adult participants with no prior programming experience. Participants ranged in age from 18 to 64 (mean age = 37.4, SD = 17.1). All but two users (aged 61 and 63) had used a tablet prior to the evaluation.

4.1.2 Materials

The materials used for the session were a Samsung Galaxy Tab E tablet on which the Tica app ran, the robot and floor tiles, and the tangible interface components (instruction blocks, mirror holder and tablet stand). Pre- and post-test questionnaires were also developed. The pre-test contained two questions, both of which showed a picture of a task grid and a set of instructions. Participants were asked to decide whether or not the robot could complete the task with the instructions given. The post-test included these same two questions along with a further two questions of the same type (but of increasing difficulty). An additional question included in the post-test required participants to select the correct set of instructions for a given task from a set of three. Finally, participants were asked to rate their enjoyment of the session (on a scale of 1 to 5) and to describe the best and worst aspects of the session.

4.1.3. Procedure

Participants were randomly assigned to either the screen-based or tangible condition (seven per condition). They were first given a description of the session and asked to complete the pre-test. They were then asked to complete six tasks using either the tangible or screen-based interface (see Figure 14). Participants were observed as they completed the tasks, and the time taken to complete each task was recorded. Participants then completed the post-test and were debriefed.

4.1.4. Results

For the two questions which appeared on both the pre- and post-tests, two participants incorrectly answered one of the questions on the pre-test (one from each interface condition), however all answered correctly in the post-test. Although there is insufficient data from which to draw conclusions, it suggests that learning occurred during the session (but shows no difference between the interfaces).

For the remaining three post-test questions, there were five incorrect answers in total, suggesting that the majority of participants had a good understanding of the instructions by the end of the session. Of the incorrect answers, two were from participants in the tangible condition, while three were from...
participants in the screen condition, suggesting that there was little difference between interfaces in terms of program understanding.

In terms of task performance, there were very few incorrect attempts, which was not surprising since the system was designed for children. All tasks were completed within two attempts regardless of interface. Users found it easy to recognise desired and erroneous results, based on their comments about the robot’s behaviour. Participants in the tangible condition would often start to correct problems before the robot had finished the current (incorrect) attempt, something that was not possible with the screen interface.

The task with the most incorrect attempts was Task 2. Several participants who had an incorrect attempt commented that they expected the robot to move laterally to the right rather than turn 90 degrees to the right, when the right instruction was initiated - this area would benefit from some further consideration. Task 4 had the second highest number of incorrect attempts which may well be due to the fact that it was the longest program, requiring a minimum of seven instructions.

Of the nine total incorrect attempts, four occurred using the tangible interface, while five occurred with the screen interface (Figure 15).

![Figure 15 – Number of incorrect attempts by task and interface type](image)

In terms of task completion time, the average time was longer in the tangible condition (mean = 26.9s, SD = 11.8s) as compared to the screen-based condition (mean = 23.6s, SD = 9.9s).

Unsurprisingly, the quickest task to complete was Task 1, which only required two instructions. The task which took the longest time to complete was Task 6 which, although it did not require the most instructions, was the most complex, involving the use of a repeat as well as three other instruction types.

![Figure 16 - Mean task duration for each interface](image)

User-reported enjoyment was overall higher than expected, considering study participants were not the target user group. Enjoyment ratings (on a 5 point Likert scale) were higher for the tangible interface (mean = 4.25, SD = 0.76) than the screen interface (mean = 3.85, SD = 0.69). This may be in part due
to the novelty of the tangible interface. Research over a longer time period would shed light on whether enjoyment of the tangible interface is maintained over time.

4.2 Facilitator pilot study
The facilitator study was designed to explore the usability of the system and identify any improvements needed from the perspective of the facilitator user.

4.2.1 Participants
Five participants were chosen for this role, based on recommendations from Nielsen (2000). Participants were selected from personal contacts, and all were final year computer science students, as they would likely have a similar level of computer literacy to a facilitator.

4.2.2 Materials
The same materials were used as in the participant session, however the tasks revolved around setting up and controlling the system. Additionally, rather than pre-/post-tests, a single questionnaire asked participants to identify the task they found most confusing and explain why, and to identify two positive and two negative aspects of the system.

4.2.3 Procedure
Participants were first given an introduction to the study, and then asked to complete the 6 pre-defined tasks while being observed. Participants then completed the feedback questionnaire.

4.2.4 Results
Time taken to complete facilitator tasks varied (see Figure 17), however all were completed in under 60 seconds on average. Given that participants had not previously used the system, the results suggest that it was intuitive and easy to use.

![Figure 17 – Average task duration](image)

Task 1 (connecting the robot and tangible system to the app via Bluetooth) took the longest to complete (59.4s on average), and was found to be the most confusing by all users. This was not surprising, as Task 1 required turning on two devices. Confusion over whether the robot and tangible system were turned on stemmed from the fact there is no clear indicator of this - a good point for future improvement.

5. Future work
The primary areas for future work involve, firstly, improving the robustness of the system and, secondly carrying out a full-scale evaluation of the Tica system.

The robustness of the Tica system could be improved in a number of ways, many of which revolve around the current version of the robot. Battery life is short, which restricts the length of sessions. This, coupled with the fact that the robot does not have a warning system for a low battery, can result in somewhat erratic behaviour when the battery does run low. A number of improvements could also be made to the app to improve usability and to show the status of the Bluetooth connection with the robot. The iteration instruction is currently fixed in the number of instructions it can hold, and so iterates only twice. Changing this to allow multiple iterations and a variable number of instructions would allow users to create more complex programs with more realistic control flow.

Having run a small-scale evaluation with adults, the next stage would be to run a classroom-based evaluation with children aged 5 - 7 years over multiple sessions. A comparative study of this nature
would allow us to determine the relative benefits and drawbacks of each type of interface, and to assess their impact on learning programming for children of this age group.

6. Conclusions
This paper presented Tica, a comprehensive environment for exploring the differences between tangible and screen-based interfaces for teaching programming to young children. Preliminary research suggests that the environment holds promise: next steps will involve researching the use of the system in situ.

7. References


Perlman, R. (1976). Using computer technology to provide a creative learning environment for preschool children. Logo memo no 24, MIT Artificial Intelligence Laboratory Publications 260, Cambridge, Massachusetts


Finding Patterns in Visualizations of Programs

Cătălin F. Perticaș
Romanian Institute of Science and Technology
perticas@rist.ro

Bipin Indurkhya
Romanian Institute of Science and Technology and Jagiellonian University
indurkhya@rist.ro

Răzvan V. Florian
Romanian Institute of Science and Technology
florian@rist.ro

Lehel Csató
Romanian Institute of Science and Technology and Babes-Bolyai University
csato@rist.ro

Abstract

We present an approach to classify computer programs according to their semantics, by using adaptive and neural network-based algorithms. We first develop a visualization method that reflects program dynamics. In order to validate the existence of patterns in visualizations, we build five features that characterize sorting methods, and apply self-organizing maps to cluster them. We then use neural networks to classify five sorting algorithms: InsertionSort, BubbleSort, HeapSort, QuickSort, and RandomSort. Our experiments show above 90% accuracy on the validation set, thereby showing that the specific sorting algorithm can be inferred from a simple visualization.

1. Introduction and motivation

In recent years, there is a renewed interest in modeling the process of software creation. Some aspects of this research are:

1. Being able to model cognitive processes underlying algorithm construction and program generation, we can develop more effective methods of teaching problem solving, and create educational and debugging tools for learning to program. One related research area is to elicit the user’s model of program execution. For example, Piaget’s model of cognition has been applied to study code completion by professional software developers (Mărăsoiu et al., 2015), as well as to study preconceptions of novice programmers about programming (Corney et al., 2012; da Rosa, 2015).

2. Being able to generate patterns to represent the dynamics of program execution, whether through visualization or other means, we can experiment with various clustering techniques to see if we can group these patterns in a way that captures the semantics of the programs. There have been a number of studies on visualizing the structure of a computer program. For example, Alhammad et al. (2016) evaluated three different tools for visualizing the following three aspects of programs: loops, object-oriented programming and parameter passing by value or reference. The evaluation, however is carried out with respect to learning to program.

3. Being able to describe the behavior of a program at a conceptual level from observing the pattern of its execution.

4. A long-term, and somewhat ambitious, goal of this research is to be able to apply machine learning techniques to create new programs that exhibit a given behavior. When trying
to automate the process of software creation, one idea is to create models that learn from code by applying machine learning techniques. Previous attempts to achieve this with a domain-specific language (DSL) include Balog et al. (2017) and Devlin et al. (2017), where the number of possible operations on the input is limited and well-defined, and the data for learning consists of input/output pairs. This encouraged us to approach simple toy-like problems, where some DSL could be identified, such as the sorting problem where the main operations are swaps.

This paper presents our initial results in following the second aspect of the research direction presented above. In particular, we are exploring ways to generate visualizations of program execution dynamics and applying clustering techniques on them to see if we can capture their semantics. Our choice of studying inline sorting algorithms provided us with two advantages: only one operation to track: swapping elements; only one array to analyze.

2. Related research

2.1. The Role of Sensorimotor Integration and Abstraction in Learning

From a cognitive perspective, it has been suggested (Corney et al., 2012) that learning to program occurs in several stages, though this view is not unique to the field of programming. The first of these stages is the sensorimotor stage. We combined tracking simple operations in programs with the idea of sensorimotor understanding and reached a novel idea, the image of algorithm, which is a representation that integrates the states on which the program operates and the actions taken by the program. This visualization is well-suited for learning models with an architecture designed for visual recognition, such as Convolutional Neural Networks (CNNs), introduced by Le Cun et al. (1990). To our knowledge, CNNs have not been used so far for program classification.

The next stages in the piagetian development for programming are increasingly more abstract, better tracing skills, understanding abstraction and program diagrams in familiar situations, and applying creativity and analogies to create programs for solving new problems. Similarly, deep neural networks (Schmidhuber, 2015) have layers which learn concepts of increasing level of abstraction. We tested this model, and in the future we plan to study whether deep neural networks could be constrained to reflect similar learning stages and abstractions.

2.2. Maps and Visualizations inspired from Dual-Coding

The sensorimotor approach to learning has support in embodied cognition approaches, where it is argued that the brain could not exist without its body. In a way, the body is a map of the mind and the mind is itself a map of the body. Hundhausen et al. (2002) mentions dual-coding, the interaction between two different symbolic systems generating cognition, as one of the key ingredients when it comes to effectiveness of visualizations. Along this idea, we investigated visualizations of spatial and temporal features of program executions. We used (C. Shaffer et al., 2007; C. A. Shaffer et al., 2010) as reference for the types of visualizations in the field of algorithms. Our choice of clustering technique, the self-organizing map, introduced by Kohonen (1990) can be used as an alternative visualization tool, although the reason we chose it was due to its superior performance.

3. Image of Algorithm (IoA)

3.1. Introduction

There are many aspects to representing run-time dynamics of programs. For example, one can look at the pattern of function/subroutine calls, the pattern of values being passed between different modules or the pattern of data objects that are being affected by the program. For our research, we focused on tracking parts of the computer memory that are being affected as the program executes. Our goal was to explore if visualizing these patterns as images, and then applying clustering algorithms to these images will result in clusters that are semantically meaningful: they reflect a conceptual structure.
The evolution of the memory state of an algorithm is difficult to infer from its source code: it requires interpretation and knowledge of algorithms, logic and programming languages. A visual representation of the memory dynamics, although larger in size, is an alternative requiring less background information and is better anchored in the reality of program execution. Moreover, recent advancements in image processing motivated us to explore this way of representing program dynamics so that existing techniques of image clustering can be applied for classifying programs based on their execution dynamics.

3.2. Initial Observations

To test the practicality of our program representations, we started with sorting algorithms since they are well studied and have available implementations. We determined that if we assign grayscale colors to the data stored in memory, and plot the changes through time, we get an image corresponding to how the program is affecting the data objects. We aimed to see if this image has a pattern that recurs over multiple executions of the algorithm given different inputs, so that it can be considered as the signature of the algorithm.

We selected five sorting algorithms — namely, InsertionSort, BubbleSort, HeapSort, QuickSort, and RandomSort — to study patterns in the visualizations of their effect on the data objects. Figure 1 shows images of algorithms. This visualization associates images to sampled executions of the five sorting algorithms mentioned above. Each row shows the state of the array being sorted at a given point in time. The array values were chosen between 0.0 and 1.0. We represented small-valued elements (closer to 0.0) as dark pixels and large-valued elements (closer to 1.0) as bright pixels. Whenever a swap occurred, we took a snapshot of the array and appended it to the image.

Let $\mathbf{A}_0 = [A_{0,0}, A_{0,1}, \ldots, A_{0,N-1}]$ be the initial array, let $\mathbf{A}_i = [A_{i,0}, A_{i,1}, \ldots, A_{i,N-1}]$ be the array at the $i$-th step. The image of the algorithm representation (IoA) of our program execution is simply the collection $\text{IoA} = \{\mathbf{A}_0, \mathbf{A}_1, \ldots, \mathbf{A}_T\}$, where $T$ is the largest number of iterations among the program executions which we take into account. For simplicity, we used collections of equal size by appending the last row $T-k$ times, where $k$ is the number of swaps in the current execution.

Thus, we get two-dimensional images with the horizontal axis representing the space (memory) of the program and the vertical axis representing the time (number of steps). These types of images are not unique to the program and not necessarily deterministic, but do show reoccurring patterns over several samples. We considered one swap operation to be one step along the time axis, although other operations can also be included. This visualization gave us some interesting...
insights into the execution patterns of different sorting algorithms.

For instance, it is visually apparent that the HeapSort (Figure 1c) works in two stages because its IoA changes its patterns after about one quarter of its execution length. Its first stage is counter-intuitive: it appears to do the opposite of sorting by placing many large elements at the beginning of the array. In fact, it creates the structure of its max-heap, which only makes sense for us in a tree-like visualization. The pattern in the second stage reveals the algorithm’s approach to start with processing elements at the end of the array.

The visualization of InsertionSort (Figure 1a) reveals the opposite approach: it starts at the beginning of the array and shifts the array to the right as smaller elements get inserted at their right position. For the BubbleSort (Figure 1b), we can observe the ‘bubble’ phenomena as large elements gradually get pushed towards the end of the array. The QuickSort (Figure 1e) displays patterns that are not so obvious because of the pivot choice, which depends on the actual sampled values for the execution on the algorithms. From the perspective of the IoA, it is more similar to RandomSort (Figure 1d).

In order to improve our study of the patterns to be found in IoAs and to explore other patterns in problems using swaps of elements on arrays, we investigated a few non-sorting algorithms, such as plain array reversal and interval swaps (see Figures 2 and 3).

The reversal program takes chunks from the array and reverses them by iteratively swapping elements from the ends until it reaches the middle of the chunk. This creates a V-shape in the associated IoA. This pattern is visible irrespective of the initial order of the array (see sorted vs. random input in Figures 2 and 3). However, this is not the case for our sorting algorithms, which would simply exit if the array were sorted, with the exception of HeapSort.

The interval swap program switches the order of two intervals (chunks) in the array. This is done by iteratively swapping elements from the first interval with elements situated at the same position in the second interval. This program produces an upside-down N-shape.

4. Feature Extraction and Analysis of Patterns in IoAs

Our goal is to be able to extract relevant features from the IoAs automatically (see Section 6). As a preliminary step, we defined five features based on our knowledge of sorting methods: Distance between Current position and Final position (DCF), Minimum Number of Swaps until Sorted (MNS), Number of Inversions (NI), First Time in Place (FTP), and First time Displaced (FTD). The first three of these are spatial features that are extracted along the space axis of the program.
execution representation. The last two are temporal features extracted along the time axis.

While the spatial features are only applicable to sorting algorithms and other problems involving permutations of objects, the temporal ones can be extracted from any program provided that the traced operations and data collections are available. We now describe in more detail the features and some of the patterns they reveal in the studied programs.

4.1. Spatial Features
Spatial features include the functions one might apply on a row or a subset of rows in the IoA. For this research, we focus on the features that are applicable to problems with a solution space represented by permutations of an array with distinct elements.

For any array at iteration $t$, we can assign a permutation $P_t$ that represents the set of order numbers for $A_t$ and if applied to $A_t$ will sort the array. In other words, $P_t, j = k$ provided that $A_t, j = S_k$, with $S$ being the sorted array in increasing order. Algorithms that sort the array in increasing order have $S = A_T$. For all the spatial features extracted, we will only need to work with these permutations.

We use the following three features in this experiment:

**Distance between Current and Final position (DCF).** We sum the distances between the current position and the final/expected position for each element of the array (for a sorted array, this sum is equal to 0):

$$DCF(t) = \sum_{j=0}^{N-1} |P_t, j - j|$$

**Minimum Number of Swaps (MNS).** Another way to measure distance to the expected solution is to compute the minimum number of swaps required to sort it. This problem reduces to finding the cycles in the permutation. As the array gets closer to being sorted, this measure decreases:

$$MNS(t) = \frac{\text{numberOfCycles}}{\sum_{j} \text{cycleSize}_j} - 1$$

**Number of Inversions (NI).** A similar feature to measure proximity to the exit state of the sorting program is the number of inversions. The number of inversions for a sorted array is equal to 0:

$$NI(t) = \sum_{j} \text{cardinal}\{P_t, k | P_t, j > P_t, k, k > j\}$$

Figure 4a shows the behavior of *InsertionSort*. We can see that the NI value decreases linearly over the number of swaps until the program terminates. This visualization provides a nice hint regarding the time complexity of this algorithm because NI is at most $N^2$ and the problem does indeed belong to the $O(N^2)$ class. At the same time, we can also infer from the slow decrease in the MNS value that the algorithm is not very efficient.

Figure 4b shows the behavior of *BubbleSort*. From the point of view of the spatial features, it is very similar to *InsertionSort*. This algorithm is also $O(N^2)$ and is known to be inefficient with its swapping strategy.

Figure 4c shows the behavior of *HeapSort*. Due to the small array size, its time complexity advantage cannot be observed. Although the algorithm is known to be fast for large arrays, its high time-complexity constant makes it inefficient for small arrays. In the figure, we can distinguish two stages of the algorithm. First, it creates a heap structure. During this stage, the memory state gets further from the desired state, and we can see an increase in the spatial features values. In the second stage, there is an abrupt descent, which is not very smooth, and hints at the use of a tree-like data structure. Its time complexity is $O(n \log(n))$. 

80
Figure 4 – Spatial features. (a) InsertionSort, (b) BubbleSort, (c) HeapSort, (d) QuickSort, (e) RandomSort. Blue: DCF, Red: NI, Green: MNS.

Figure 4d shows the behavior of QuickSort, a very efficient algorithm. We notice a very fast decrease in the feature values. Moreover, its MNS values are the only ones which show a few temporary increases. This suggests it often chooses optimal swaps for sorting the array.

Figure 4e shows the behavior of RandomSort. Interestingly enough, choosing random swaps generates a behavior that is similar up to a point to QuickSort. However, this can be misleading because this algorithm generates many more possible swaps than shown in the image, but only executes the valid ones, which appear in the image.

4.2. Temporal Features
Temporal features are the category of features we can extract along the time axis of our program execution representation, the IoA. Any function that we apply on a column or a subset of columns in the IoA is considered to be a temporal feature.

First Time In Place (FTP). For each memory cell tracked during the execution of the program, we can extract the necessary number of iterations for it to arrive to its final state:

\[ FTP_i = \text{argmin}\{A_{j,i}|A_{k,i} = A_{j,i}, \forall k < j\} \]

This feature compresses the IoA in a way which made it possible to cluster the different algorithms. Dimensionality reduction is necessary in this case because the IoA feature set has a large cardinality and therefore confuses the learning algorithm in the absence of a pattern extraction interface. The FTP features reduce the dimensionality to the number of elements in the array, while keeping traces of the dynamics of the algorithm. We found that the FTP is invariant to the sorting order. Therefore, it can capture the sorting dynamics well, being able to put executions of the same sorting algorithm, but with different sorting order, under the same cluster. This was done using the self-organizing map model. However, the order invariance comes as a disadvantage when we try to discriminate by order.

First Time Displaced (FTD). For each memory cell which is tracked during the execution of the program, we can extract the necessary number of iterations for it to change its initial state:

\[ FTD_i = \text{argmax}\{A_{j,i}|A_{k,i} = A_{j,i}, \forall k > j\} \]

An optimal sorting algorithm, which would perform a minimal number of swaps would have
the FTP coincide with FTD. The sorting methods which we investigated do not exhibit this behavior, so the two features follow different patterns. This can be seen in Figure 5.

5. Feature-based Clustering
5.1. Overview
The following experiments indicate that given the right set of features and clustering techniques, simple programs such as sorting an array can be naturally grouped into categories which reflect their semantics.

5.2. Self-Organizing Map (SOM)
Self-Organizing Maps are artificial neural networks where neurons are represented as cells in a grid. Each input that we feed in the network will activate the neuron with the most similar weights to the input. Each activation changes the surrounding neurons by adjusting their weights to be closer to the activated neuron. The closer the surrounding neuron is to the activated one, the stronger the adjustment is.

The grid consists of 15 x 15 or 20 x 20 neuron cells. The learning algorithm performs 30 activation iterations and the number of input samples is 100 for each of the 5 sorting methods. The arrays used for learning had 10 elements.

In Figures 6 and 7 there are 5 strongly-colored cells, each representing the centroid obtained by averaging the locations of the sample executions for each algorithm. For the remaining cells, the lighter they are, the more input samples activate the neuron represented by the cell. The clustering method has no prior knowledge of which execution sample belongs to which algorithm.

Out of the clustering methods for sorting algorithms, Self-Organizing Map with FTP worked the best in creating well-separated clusters, as shown in Figure 6a. A more thorough analysis of their performance will require assigning an accuracy metric to the clustering techniques.

This SOM clustering algorithm also worked well on the MNS feature (Figure 7a), which ranked the second in our analysis.

On the other hand, the DCF (Figure 7b) and NI (Figure 7c) features cannot distinguish between InsertionSort and BubbleSort. This is to be somehow expected if we take a look at the DCF and NI curves in the two sorting algorithms: they are very similar visually. The FTD feature fails to distinguish between QuickSort and RandomSort, as shown in (Figure 6b).
5.3. Control Experiments

In addition to our initial experiments, we tried clustering the five sorting programs as well as two non-sorting programs mentioned earlier: array reversal and interval swaps. We came up with two variants of these non-sorting programs, one in which the input is sorted and the execution patterns are very clear, and another one where the input is random (not sorted).

Our goals behind this experiment were to test the robustness of our technique — how well our approach would generalize to new programs — and to verify that we get some real cluster difference between sorting and non-sorting programs. We obtained interesting results as shown in Figure 8.

First observation: non-sorting clusters are well separated from the sorting clusters for both chosen features. This confirms the robustness of our features and technique for clustering.

Second observation: for the FTP feature, we get the same one-cell clusters for the two types of problems when controlled by sorted input versus non-sorted input. This reveals two facts: the problems of our choice (reverse and interval swap) do not depend on the array values, as opposed to sorting algorithms (inference on FTP clusters - overlap of sorted input vs. non-sorted input clusters); the two programs have a predetermined dynamics perfectly encoded by the FTP feature, so that the clustering algorithm assigns exactly one neural cell to all samples of these algorithms.

Third observation: clustering on MNS behaves similarly to the one on FTP, but only when the input is sorted. This is because different initial orderings result in different values for the spatial features (including MNS). For the non-sorted input, both programs get a well-defined cluster.

6. IoA Classification

6.1. Overview

The IoA classification is very much like the task of recognizing images. Using neural networks for IoA classification would confirm, if accurate enough, that a model with several layers would be able to capture the patterns inside these images.

For the classification experiment, we used ten types of classes, each corresponding to a pair of
one sorting algorithm and a sorting order (increasing or decreasing). For each class, we used 1000 samples, i.e. different input vectors, resulting in a dataset of 10,000 samples in total. Each input vector was generated randomly, with values from the vector being generated with a uniform distribution between 0 and 1.

We tried two approaches: multilayer perceptron and convolutional neural network, both with four layers on top of the input layer. We used 80% of the data for training and the remaining 20% for validation.

6.2. Multilayer Perceptron (MLP)

We normalized the IoAs, which have a 2D visual representation, by appending the last row until all the images had the same size. Then the images were represented in a vectorial form. Using this format, we trained the MLP on the collected data.

The architecture consisted of 4 layers, 3 hidden ones and one readout. For the hidden layers we used 500 learning units. The input had 420 units (10 array cells x 42 time steps) and the readout had 10 units (5 sorting algorithms x 2 sorting orders). The training time was noticeably smaller than in the CNN case, at the cost of less accuracy — 90% accuracy on the validation set.

6.3. Convolutional Neural Network (CNN)

We used the normalization applied for MLPs, then trained a convolutional neural network with four layers: convolutional (32 features), pooling (2x2 reduction), hidden (1024 units) and readout (10 units). The input had 420 units. This model achieved the highest accuracy - 92.5% on the validation set. This proves that deep learning models can generalize well the patterns extracted from IoA representations of programs, even from relatively small datasets.

Afterwards, we inspected the convolutional layer in order to understand the relevant patterns that help distinguish between different sorting algorithms. Our results showed that the feature activations do not differ too much between different sorting methods, which would roughly indicate that the same set of features could be used to discriminate between them. However, the feature activation maps of program execution for different sorting orders differ considerably. An analysis of the neuron activations from the first convolutional layer indicates that our classification algorithm relies on temporal patterns and the time complexity of the algorithm.

7. Conclusions and Future Work

We have developed the IoA method, which displays the patterns inside programs in a human readable way (Section 3.2). We have shown that automated learning models were able to use the visualizations, as well as features/patterns extracted from them (Section 4) in order to cluster (Section 5.2) and classify (Section 6) with high accuracy five different sorting algorithms. Moreover, we have shown that sorting algorithms generate patterns which can be distinguished from other algorithms (Section 5.3).
We estimate that the models explored so far to group and classify sorting programs based on the exhibited patterns can be extended to more complex algorithms. A large number of computer science primitives could be clustered and automatically identified. There is an abundance of operations besides swap, such as push, pop, append, replace, as well as other data structures, such as stacks, queues and trees, which can be used in future work.

8. Acknowledgement
This work was supported by the European Regional Development Fund and the Romanian Government through the Competitiveness Operational Programme 2014-2020, project ID P_37_679, MySMIS code 103319, contract no. 157/16.12.2016.

References


Dynamic Translation of Spreadsheet Formulas to Problem Domain Narratives

Bennett Kankuzi
Department of Computer Science
North-West University, Mafikeng Campus
P/Bag X2046, Mmabatho 2735, South Africa
bennett.kankuzi@nwu.ac.za

Abstract
Most errors in spreadsheets are formula-based. Referenced cells in formulas are normally presented using the traditional A1 cell referencing style. A spreadsheet user has to therefore mentally map referred cells to their corresponding labels in order to comprehend a formula in the context of the problem domain. In this paper, we give a detailed description of an algorithm that can be used to dynamically translate traditional spreadsheet formulas to their problem domain equivalents which are easier to understand. The translation is done as one accesses a formula cell in a spreadsheet. The formula translation is based on inferred labels of referenced cells in the formula. The aim of the translation is to ease the cognitive load on the spreadsheet user and hence improving the error-prone spreadsheet development process. The paper also highlights some factors that need to be taken into consideration when dynamically translating spreadsheet formulas. The number of referenced cells in a formula and the distance of labels from referenced cells will determine the speed of the translation process hence affecting system responsiveness as one navigates through a spreadsheet. Unpredictable spatial arrangement of data in spreadsheets (spreadsheet layout) can also pose a challenge to the translation algorithm which may lead to mis-translation of spreadsheet formulas. These challenges might increase the cognitive load on the spreadsheet user hence negating the purpose of dynamically translating spreadsheet formulas.

1. Introduction
Spreadsheets are one of the most popular end-user programming environments (Abraham, Burnett, & Erwig, 2008). Their simple and easy to use tabular visual interface makes it easy to manipulate data and instantly see the results of the data manipulation (B. A. Nardi & Miller, 1990). However, spreadsheets are plagued with the problem of being vulnerable to errors (Powell, Baker, & Lawson, 2009). In spreadsheet cells, one can enter either labels, data values or formulas (Abraham et al., 2008). A data value or formula without corresponding labels is meaningless as the labels carry the meaning of the cell value or formula in relation to the context of the problem being solved using a particular spreadsheet.

Computations in spreadsheets are entered through formulas. Research, however, has also shown that most errors in spreadsheets are formula based (Rajalingham, Chadwick, Knight, & Edwards, 2000). Formulas normally reference other cells in a spreadsheet. Referenced cells in formulas are normally presented using the traditional A1 cell referencing style such as B3, D7, G12, etc. A spreadsheet user has to therefore mentally map referenced cells to their corresponding labels in order to comprehend the meaning of a formula in the context of the problem domain. This increases the cognitive load of the spreadsheet user as the mental model of the problem domain has to be constantly mapped back and forth to its spreadsheet counterpart (Kankuzi & Sajaniemi, 2013). Cognitive load can be defined as the mental effort one has to make to learn new information in order to accomplish an intellectual task (Sweller, 1994). The higher the cognitive load, the greater the risk for one to commit an error when completing a task (Paas, Renkl, & Sweller, 2004). It is thus, imperative that even in the formula comprehension process, spreadsheet users do not have high cognitive load.

Usability is an important aspect of human-centred software engineering (Seffah & Metzker, 2004) and a recurring usability goal is to reduce cognitive load for users (Hollender, Hofmann, Deneke, & Schmitz, 2010). One technique of reducing cognitive load in user interfaces is by having users focus on recognition rather than on recall of interface elements (Hollender et al., 2010). One way to do this in spreadsheets is by translating a given traditional spreadsheet formula into its problem domain equivalent which
is easier to understand in relation to the problem being solved.

In previous work, Kankuzi and Sajaniemi (2014a, 2014b) developed an interactive spreadsheet visualization tool that dynamically translates a given traditional spreadsheet formula into its problem domain equivalent which is easier to understand in relation to the problem being solved. The formula translation is based on inferred labels of referenced input cells in the formula and is done as one accesses a formula cell. The inferred labels for a particular referenced input cell form symbolic names for the cell. The translated spreadsheet formula is called a “problem domain narrative” or in short a “domain narrative”. For an active formula cell, a domain narrative is displayed in a box just right below it. Domain narratives are positioned a little bit lower than the active formula cell to avoid distractions when navigating through the cells. The tool also highlights all referenced cells in an active formula cell and marks a formula cell with a pink right border. For example, in the spreadsheet depicted in Figure 1, a formula in cell C9 given as “= SUM(C5:C8)”, is automatically translated to “SUM( Jan | James Bourne ... Jan | Jasmine Hunt )”. An empirical evaluation of the tool by Kankuzi and Sajaniemi (2014b, 2016) found that it helped in reducing cognitive load of spreadsheet users as it helped in mapping a spreadsheet user’s spreadsheet specific mental model to the corresponding problem domain mental model. The tool also helped spreadsheet users to statically identify more errors in spreadsheets (Kankuzi & Sajaniemi, 2014b, 2016).

This paper has two aims. First, the paper gives a detailed description of the translation algorithm that is the engine of the interactive spreadsheet visualization tool presented and evaluated in Kankuzi and Sajaniemi (2014a, 2014b, 2016). Second, the paper also highlights factors that need to be taken into consideration when dynamically translating spreadsheet formulas. We therefore present a technical performance analysis of the algorithm in particular in relation to system responsiveness which is an important usability factor in interactive tools (Waloszek & Kreichgauer, 2009). We thus analyze the time and space complexity of the algorithm. We also discuss how the problem of unpredictable spreadsheet layouts (Abraham & Erwig, 2004; Koci, Thiele, Romero Moral, & Lehner, 2016; Roy, Hermans, Aivaloglou, Winter, & van Deursen, 2016) can affect the formula translation process.

The rest of the paper is organized as follows: in Section 2, we present our algorithm for dynamically translating spreadsheet formulas. In Section 3, we present and discuss factors that need to be considered when dynamically translating spreadsheet formulas to problem domain narratives. We present related literature in Section 4. Concluding remarks are given in Section 5.
2. The Algorithm
In this section, we present an algorithm for dynamically translating spreadsheet formulas to their corresponding problem domain narratives.

First, we get all the cells being referenced in a given active spreadsheet formula.

Second, for each referenced cell, we do the following in sequence:

(i) We extract the column label and row label for the referenced cell. A column label is extracted with respect to the relative position of the referenced cell in a spreadsheet. From the position of the referenced cell, a column label is extracted by visiting upwards all cells in the same column as the reference input cell and in the process concatenating contents of all label cells. A column label can thus be formed from more than one label cell. A “label cell” may be defined as a non-empty cell which does not have both precedent cells and dependent cells (Abraham & Erwig, 2004). In other words, a label cell is a non-empty cell which is not referencing any other cell and it is also not referenced by other cells. A label cell is therefore not just restricted to text but can also contain dates, numbers such as years, etc. We terminate the column label forming process in two situations. If we have encountered an empty cell after already visiting a label cell, that will signify an end of a spreadsheet logical block as spreadsheet users normally separate spreadsheet logical blocks in spreadsheets with empty cells. We also terminate the process if we reach the top boundary of the spreadsheet interface.

Similarly, a row label is extracted with respect to the relative position of the referenced cell in a spreadsheet. From the position of the referenced cell, a row label is extracted by visiting leftwards all cells in the same column as the reference input cell and in the process concatenating contents of all label cells. Just as a column label, a row label can also be formed from more than one label cell. We also terminate the row label forming process in two situations. If we have encountered an empty cell after already visiting a label cell, that will signify an end of a spreadsheet logical block. We also terminate the process if we reach the left boundary of the spreadsheet interface.

(ii) A symbolic name for the referenced cell is then formed by concatenating the extracted column label with the extracted row label but separating them by a “|” to have column label | row label as a symbolic name. The naming follows a “column | row” direction because traditional spreadsheet cell naming follows the column-row convention. There are also some situations where a referenced cell can have an incomplete symbolic name: a referenced cell which does not have a corresponding column label has its row label as its symbolic name; a referenced cell which does not have a corresponding row label has its column label as its symbolic name; and a referenced cell that does not have both its corresponding row label and column label has “unnamed” as its symbolic name.

(iii) Each of the textual references of the referenced cell in the untranslated or partly translated spreadsheet formula are replaced with the symbolic name and thus progressively forming a domain narrative. Occurrences of the range operator “:” in a formula are also replaced by “ ... ”.

Third, a fully translated spreadsheet formula or domain narrative is displayed to the spreadsheet user. For example, for the formula in cell C9 in Figure 1 given by “=SUM(C5:C8)”, its corresponding problem domain narrative is “SUM( Jan | James Bourne ... Jan | Jasmine Hunt )”. The pseudo-code for the translation algorithm is given in Algorithm 1.

3. Challenges in Dynamic Translation of Spreadsheet Formulas
3.1. System Responsiveness
We analyze the time and space complexity of the translation algorithm given in Algorithm 1 to see the effect of the translation algorithm on system responsiveness. From the algorithm, we can deduce that the time complexity is \(O(mn)\), where \(m\) is the number of referenced cells in a formula and \(n\) is the number of cells that have to be visited in order to form a symbolic name of each referenced cell. A large number
Algorithm 1 An algorithm for dynamically translating a spreadsheet formula into a problem domain narrative

1: procedure GENERATEDOMAINNARRATIVE(activeFormulaCell)
2:     referencedCellsList ← GETREFERENCEDCELLSLIST(activeFormulaCell)
3:     for each referencedCell in referencedCellsList do
4:         columnLabel ← GETCOLUMNLABEL(referencedCell)
5:         rowLabel ← GETROWLABEL(referencedCell)
6:         if isNotEmpty(columnLabel) and isNotEmpty(rowLabel) then
7:             symbolicName ← columnLabel + “ | ” + rowLabel
8:         else
9:             if isNotEmpty(columnLabel) and isEmpty(rowLabel) then
10:                 symbolicName ← columnLabel
11:             else
12:                 if isNotEmpty(columnLabel) and isNotEmpty(rowLabel) then
13:                     symbolicName ← rowLabel
14:                 else
15:                     symbolicName ← “unnamed”
16:             end if
17:         end if
18:     end if
19:     Replace referencedCell with symbolicName in spreadsheet formula to progressively form a domain narrative
20: end for
21: Store or display generated domain narrative
22: end procedure

23: function GETREFERENCEDCELLSLIST(activeFormulaCell)
24:     return a list of all cells being referenced in activeFormulaCell
25: end function

26: function GETCOLUMNLABEL(referencedCell)
27:     position currentUpwardColumnCell indicator one cell up in the column of the referencedCell to get currentUpwardColumnCell
28:     while (not(topSpreadsheetBoundary)) do
29:         if hasNoPrecedents(currentUpwardColumnCell) and hasNoDependents(currentUpwardColumnCell) then
30:             columnLabel ← currentUpwardColumnCell + columnLabel
31:         end if
32:         if isBlank(currentUpwardColumnCell) and isNotEmpty(columnLabel) then
33:             exit while loop
34:         end if
35:     position currentUpwardColumnCell indicator one cell up in the column of the referencedCell to get currentUpwardColumnCell
36:     end while
37:     return columnLabel
38: end function
function GETROWLABEL(referencedCell)
43:  position currentLeftwardRowCell indicator one cell left in the row of the referencedCell
44:  to get currentLeftwardRowCell
45:  while (not(leftSpreadsheetBoundary)) do
46:      if hasNoPrecedents(currentLeftwardRowCell)
47:      and hasNoDependents(currentLeftwardRowCell) then
48:         rowLabel ← currentLeftwardRowCell + rowLabel
49:      end if
50:      if isBlank(currentLeftwardRowCell) and isNotEmpty(rowLabel) then
51:         exit while loop
52:      end if
53:      position currentLeftwardRowCell indicator one cell left in the row of the referencedCell
54:      to get currentLeftwardRowCell
55:  end while
56:  return rowLabel
57: end function

of referenced cells will imply that the formula translation process will also take longer as there will be a large number of referenced cells that will need to be translated. If labels are located further away from the referenced cells, the translation process will take longer as there will be a large number of cells to traverse before reaching desired label cells. Longer translation times will imply slow system response as one navigates from one formula cell to the other. However, waiting for system response for more than an expected period affects short-term memory as information in short-term memory decays over time (Miller, 1968). Human beings also want to feel in control rather than having the pace of their task being controlled by a computer (Miller, 1968). For dynamic translation of spreadsheet formulas to be effective, the number of referenced cells in a spreadsheet formula therefore needs to be relatively small and the label cells should be located closer to the referenced cells.

The algorithm has also space complexity \(O(n)\), where \(n\) is the number of referenced cells in a formula. A large number of referenced cells will imply that the formula translation process will need more memory to hold the collection of referenced cells. This might also affect system responsiveness as one accesses formula cells in a spreadsheet. The number of referenced cells in a spreadsheet formula therefore needs to be relatively small for dynamic formula translation to be effective.

3.2. Varying Spreadsheet Layouts
Varying spreadsheet layouts also pose a challenge to formula translations as spreadsheet creators can layout their spreadsheets in any way they want (Abraham & Erwig, 2004; Koci et al., 2016; Roy et al., 2016). This challenge applies to formula translations in general and not just to dynamic formula translations. Some spreadsheets can be quite simple while others can be complicated with hierarchical headers, repeated data blocks, vertical data blocks sharing same column headers, blank rows separating blocks, etc (Roy et al., 2016). Our algorithm produces symbolic names which are inferred by inspecting label cells, column wise or row-wise, from a particular referenced cell. This might lead to incomplete symbolic names in some translated formulas in complex spreadsheets which might end up confusing the spreadsheet user. To mitigate this problem, in the spreadsheet visualization tool which uses this algorithm, we also highlight the cells which are being referenced in an active formula cell. For example, in the spreadsheet in Figure 2, the formula in cell G10, given by “=SUM(D10:F10)” and has been dynamically translated to “SUM( 2010 | Rice ... 2012 | Rice )”. The same translation would occur for the formula in cell L10. This would be confusing to the spreadsheet user. A complete and correct translation for formula in cell G10 would have included the hierarchical headers and consequently would have been translated as “SUM( CountryX ~2010 | Crop Products ~Rice ... CountryX ~2012 | Crop Products ~Rice )”. On the other hand, the formula in cell 10 would be translated as “SUM( CountryY ~2010 | Crop Products ~Rice ... CountryY ~2012 | Crop Products ~Rice )”. To compensate for this shortfall in our
algorithm, the spreadsheet visualization tool (Kankuzi & Sajaniemi, 2014a, 2014b, 2016) highlights the cells being referenced in the active formula cell. In Figure 2, referenced cells D10, E10 and F10 have been highlighted accordingly for formula in cell G10.

4. Related Work
Symbolic names and spreadsheet formula translations have also been used in some spreadsheet auditing tools as well as some spreadsheet systems. A notable difference among the tools, however, being in syntactical notation and location of display of a translated spreadsheet formula.

D. Nardi and Serrecchia (1994) developed a spreadsheet visualization tool that translates spreadsheet formulas. In this tool, labels for input cells to a formula are given out as symbolic names as illustrated in Fig. 3 in which cell C4 is translated as “PROFIT(year_2)” and are displayed in an interface separate from the native spreadsheet system. If in Figure 3, the formula in cell B4 was =B2-B3, the tool by D. Nardi and Serrecchia (1994) would generate an alternative corresponding explanation for the formula as “REVENUE(year_1) - COST(year_1)”.

Spreadsheet Professional, a spreadsheet add-in developed by Spreadsheet Innovations Ltd (2017), translates formulas whereby input cell references in a formula are replaced by symbolic names obtained by searching for the first textual cell leftwards (or optionally rightwards) and/or upwards (or downwards)
from the referenced cell. The drawback with finding symbolic names by just searching for the first textual cell (leftwards, etc) is that it leaves out the neighbouring text which might form a “whole” symbolic name as in many spreadsheets, symbolic names may occupy more than one cell. In Figure 4, Spreadsheet Professional translates the formula, “=SUM(C5:C8)” in the active cell given in cell C9 in the spreadsheet captured as “=SUM(James Bourne Jan:Jasmine Hunt Jan)”.

And in the translation bar, Spreadsheet Professional displays “C9: Total North Jan = SUM(James Bourne Jan:Jasmine Hunt Jan)”.

Spreadsheet Detective, a spreadsheet add-in developed by Southern Cross Software (2017), also translates spreadsheet formulas into a form similar to that of Spreadsheet Professional. Numbers, a spreadsheet application developed by Apple Inc. (2017), also automatically translates spreadsheet formulas by creating named ranges over given data after adding data and headers.

Except for the tool by D. Nardi and Serrecchia (1994), in the other tools and spreadsheet system just stated above, the algorithms on how the formula translations are done, are not explicitly stated.

Hermans, Pinzger, and Van Deursen (2011), however, also presented a spreadsheet formula translation algorithm in which A1 cell references in a spreadsheet formula are also replaced by cell labels. However, their algorithm differs from our algorithm in many aspects. First, in their algorithm, classification of cells is done before the translation process starts i.e. all cells are inspected and marked as formula cell, data cell or label cell. In our case, our algorithm determines whether a cell is a label cell, during the formula translation process as one accesses a formula cell in a spreadsheet. We also do not initially classify formula or data cells. And our algorithm, determines a label cell by simply checking whether it does not have both precedent and dependent cells. This way, label cells can contain text, numbers, dates, etc. Second, in the algorithm by Hermans et al. (2011), cell references in a formula are replaced by labels obtained by searching for the first label in a row or a column. The drawback with this approach is that it leaves out neighbouring labels which might form a complete symbolic name as in many spreadsheets, symbolic names may occupy more than one cell. For example, if a name of a person is split into “first name” and “last name” columns, the algorithm by Hermans et al. (2011) will only pick the first name as the label. In our case, our algorithm forms complete symbolic names by concatenating neighbouring label cells. In other words, our algorithm does not just pick the first label cell in a row or column. Third, the environment of the algorithm by Hermans et al. (2011) is a static spreadsheet tool that is used to generate leveled high level data flow diagrams to aid spreadsheet comprehension. In our case, our algorithm is the engine of an interactive spreadsheet visualization tool, within the spreadsheet environ-
mention, whose purpose is to ease cognitive load on the spreadsheet user by helping to map a spreadsheet user’s spreadsheet specific mental model to the corresponding problem domain mental model (Kankuzi & Sajaniemi, 2014a, 2014b) through the dynamic spreadsheet formula translations. Lastly, in our work, we have also gone further to give a detailed description of the algorithm as well as a technical analysis of the performance of the algorithm in relation to system responsiveness.

Symbolic names in formula translations or problem domain narratives may appear to be similar to standard named ranges in spreadsheets. The usefulness of standard named ranges has, however, been subject to debate. For example, McKeever, McDaid, and Bishop (2009) found that the use of named ranges may lead to a reduction in debugging performance, particularly for novice spreadsheet users. Standard named ranges, however, are statically defined by a spreadsheet user hence being more error-prone. On the other hand, domain narratives or symbolic names in formula translations as used in our context are generated automatically as one accesses a formula cell and they are accompanied by a clear visualization of the referenced cells. Panko and Ordway (2005) also argue that in selecting ranges for standard named ranges, pointing errors can assign the wrong range to a range name and thus although the range will be wrong, it will appear to be correct in formulas that merely reference the range names. This can not happen with domain narratives as they are automatically generated and each narrative corresponds to the referenced range area and as such a “correct range name but an incorrect range” type of error (Panko & Ordway, 2005) is impossible with dynamic translations of spreadsheet formulas. An empirical evaluation of dynamic formula translations by Kankuzi and Sajaniemi (2014b, 2016) also found that they helped in reducing cognitive load of professional spreadsheet users as it helped in mapping a spreadsheet user’s spreadsheet specific mental model to the corresponding problem domain mental model. The dynamic translations of spreadsheet formulas also helped spreadsheet users to statically identify more errors in spreadsheets (Kankuzi & Sajaniemi, 2014b, 2016).

5. Conclusion

We have presented a detailed description of an algorithm that can be used to dynamically translate traditional spreadsheet formulas into problem domain narratives which are easier to understand. The formula translation is based on inferred labels of referenced input cells in a given formula. The aim of the formula translations is to ease the cognitive load of the user in the spreadsheet formula comprehension process. We have also presented challenges that need to be considered when dynamically translating spreadsheet formulas to problem domain narratives. This paper has three key contributions:

i. We have given a detailed description of an algorithm that can be used to dynamically translate spreadsheet formulas to problem domain narratives right within the spreadsheet environment as the spreadsheet user accesses formula cells.

ii. We have demonstrated, through an analysis of the time and space complexity of the translation algorithm, how the number of referenced cells in a formula and the distance of labels from referenced cells will determine the speed of the translation process and consequently system responsiveness. System responsiveness is an important usability factor in interactive environments.

iii. We have demonstrated how varying spreadsheet layouts also pose a challenge to dynamic spreadsheet formula translation algorithms and have shown how we can mitigate the impact of this challenge in interactive spreadsheet visualization tools through measures such as highlighting of referenced cells.

As part of our future work, we intend to investigate on how problem domain narratives can be used to automatically detect errors in spreadsheets.

6. References


Communications of the ACM, 47(12), 71–76.
Towards Webcam-based Eye Tracking in the Eclipse IDE

Sebastian Lohmeier
Freie Universität Berlin and eCube GmbH
sl@monochromata.de

Abstract
Eye tracking is used in program comprehension research and is potentially useful for enhancing inte-
grated development environments (IDEs). While eye tracking plug-ins exist for the Eclipse IDE, eye
trackers do not typically work on operating systems other than Windows. To make empirical studies ac-
cessible to programmers working on Linux or MacOS, a recently presented webcam-based eye tracker
is being adapted to the Java platform and provided as a plug-in for the Eclipse IDE. The eye tracker
shall demonstrate eye tracking in the IDE without external devices to stimulate ideas of how to improve
webcam-based eye tracking and how eye tracking could be used to improve program comprehension
research and IDEs.

Introduction
Progress has been made in recent years in integrating eye trackers into the Eclipse IDE (Shaffer et al.,
2015; Lohmeier, 2015) and iTrace of Shaffer et al. (2015) is now open source. Eye tracking requires an
eye tracker, though. Looking at commercially-available remote eye trackers, they are either prohibitively
expensive, come with restrictive licensing terms, or are only available for Windows. The situation might
be due to a chicken and egg problem: Few eye trackers are sold (at high prices) while the eye tracking
killer app is yet to be found and eye tracking is not used due to high prices, making it hard to imagine
an eye tracking killer app. It is therefore desirable to strive for low-cost eye trackers – both to ease eye
tracking studies in the field and to be able to support programmers’ work via eye tracking.

Webcam-based eye tracking
Webcam-based eye tracking has recently drawn attention with eye trackers based on constrained local
models (CLM) that use mouse and keyboard interaction for implicit calibration. This form of calibration
seems well-suited for the use of eye tracking in programming where participants sit in front of the com-
puter for a long time and mouse and keyboard are used frequently. WebGazer and PACE are examples
of such eye trackers.

WebGazer is a JavaScript-based in-browser eye tracker that is implicitly calibrated instantaneously via
relatively few mouse clicks and mouse movements (Papoutsaki et al., 2016). Its authors report an average
error of at least 100px between gaze and corresponding click locations from a study of 82 participants
with normal vision, contact lenses, and glasses who were free to move their heads and took less than 10
minutes per participant on average.

PACE is an eye tracker for desktop applications that uses a webcam and is calibrated using hundreds of
mouse and keyboard events (Huang, Kwok, Ngai, Chan, & Leong, 2016). The authors report an average
accuracy of 2.56° of visual angle from a study with 10 participants that created at least 1500 interaction
events each during 4 hours on average. PACE is interesting because it is based on a careful study of user
interaction that might be suitable for programming tasks as well.

Eclipse plug-in
Like Papoutsaki et al. (2016), the Eclipse plug-in uses https://www.auduno.com/clmtrackr/
for face tracking. Because Clmtrackr is implemented in JavaScript, J2V8 is used: it embeds the V8
JavaScript engine of Google Chrome in Java. An SVM is trained using eye images and mouse coordinates
to predict gaze after training. The plug-in works with glasses and contact lenses and is tested on
Linux, Mac OS, and Windows. It is a proof-of-concept so far and requires further development, before its tracking accuracy can be evaluated. It will also be necessary to consider Huang et al. (2016) to identify forms of mouse and keyboard interaction during programming that are suitable for training the eye tracker.

**Psychology of programming**

The Eclipse plug-in is motivated by work like Lohmeier (2016) that aims at a cognitive model of program comprehension. Records of eye movements are input into that model to construct one (out of many) plausible representations of a programmer’s memory of source code. These representations can be used to predict knowledge-related comprehension difficulties of individual programmers – parts of the code that integrate less known information might be hard to comprehend. Both working memory and long-term memory might be covered. Collecting such records might be feasible with low frequency and low accuracy webcam-based eye trackers.

One of the key challenges for such applications will be to shape problems in such a way that (a) permits the error that the low accuracy of webcam-based eye trackers introduces and that (b) capitalizes on the easy availability of webcam-based eye trackers. One of such problems might be the identification of potential referents of a referring expression: Instead of presenting an exhaustive list of all identifiers in a program that are equal to the referring expression, potential referents from the programmer’s field of view might be presented.

It would also be interesting to interpret eye tracking data to identify potential comprehension difficulties. It is yet unclear whether webcam-based eye trackers will be able to deliver data fast and accurate enough for such tasks and in sufficient amounts, though.

Webcam-based eye tracking lowers the barrier for industry programmers to participate in studies that use eye tracking to study program comprehension. Another idea behind the eye tracker plug-in is to have software developers consider eye tracking and become creative with this new input device in order to see what kinds of scenarios eye tracking could be useful for and to reflect on what happens when they read and write source code.

**Conclusion**

The current work shows that porting the concept of webcam-based eye tracking to a Java IDE is feasible. The described eye tracker plug-in shall make eye tracking in the Eclipse IDE accessible on a variety of operating systems. It is yet to be shown whether the interactions between programmers and IDEs are suitable for implicit calibration of the eye tracker. If that is the case, webcam-based eye tracking potentially provides easy access to low accuracy eye tracking data to improve IDEs and facilitate program comprehension research.

**References**


The role of the cognitive style in improving the learning to program

Margherita Pasini  
Dept. of Human Sciences  
University of Verona (Italy)  
margherita.pasini@univr.it

Ugo Solitro,  
Dept. of Computer Science  
University of Verona (Italy)  
ugo.solitro@univr.it

Margherita Brondino  
Dept. of Human Sciences  
University of Verona (Italy)  
margherita.brondino@univr.it

Daniela Raccanello  
Dept. of Human Sciences  
University of Verona (Italy)  
daniela.raccanello@univr.it

Abstract

Nowadays informatics is a part of most university curricula. In particular, in scientific and technological studies, a course in computer programming is often proposed in the first years. Nevertheless learning a programming language and solving problems in an algorithmic way is a hard task for many students. To reduce students difficulties and improve their performance, we think that is necessary to achieve a better comprehension of the cognitive processes involved. In this study, we analyse the relationship between cognitive styles, inside the theoretical framework of the empathizing–systemizing (E–S) theory, and performance in a sample of 46 students attending a course of programming in a Applied Mathematics bachelor. In line with the literature, women showed a higher level of empathy than men. However, no differences in performance were found between male and female students, even if for female students a higher level of systematizing quotients was related with a higher performance.

1. Introduction

The role and importance of Informatics as an independent subject in primary and secondary school education have been emphasized in several works (Wing, 2006), (Gander et al., 2013), (Katai, 2015).

The necessity of improving computational skills has been recognized by several educational institution in the world. As a consequence the primary and secondary school curricula are being changed with the purpose of introducing elements of computational thinking in many learning activities. Since among basic abilities considered in computational thinking we have algorithms and coding, we can imagine that the practical implementation of these recommendations shall enable students to cope comfortably with early informatics tasks at the university level.

Even so, it seems that at present time many students have difficulties in thinking about problems in an algorithmic way. Exploring the mechanisms underlying the programming activity, taking into account a psychological perspective, could be a good way to understand this process, and consequently plan adequate teaching activities which consider also individual differences. Furthermore, the importance of motivational and affective aspects has been highlighted also in the context of learning how to program (Pasini, Solitro, Brondino, & Raccanello, 2016), suggesting that research on cognitive processes involved in this challenging task must be deepened analyze.

This study is a first step in an action-research which aims to understand some psychological antecedents – cognitive and affective – of the performance of learning how to program. In this phase, this performance is analyzed considering the cognitive style, inside the theoretical framework of the Empathizing-Systemizing (E-S) Theory (Baron-Cohen, 2002).
2. Empathizing-Systemizing Theory and Programming Aptitude

The E-S Theory suggests that people can be classified on the basis of two different cognitive styles: empathizing, which is connected with the comprehension of emotional states of other individuals, and systemizing, which allow individuals to predict behaviour of systems on the basis of the knowledge of the underlying rules.

Two psychometric instruments had been constructed to measure these two psychological and cognitive styles: the Empathy Quotient (EQ) and the Systemizing Quotient (SQ). The E-S Theory assumes that a high level of SQ should be connected with a good performance in the domains in which this skill is important, such as scientific disciplines, whereas a high level of EQ should be more necessary in other domains in which the comprehension of others is important, for instance humanities and social sciences (Wakabayashi et al., 2006).

Also the link between these two cognitive styles and sex has been broadly explored in the literature, showing that women have generally a higher level of EQ than men. On the other side, men generally show higher scores in SQ.

The link between E-S Theory and programming aptitude has been stated by Wray (Wray, 2007), and recently resumed by Coles and Phalp (Coles & Phalp, 2016). These authors found controversial results, making unclear whether it is true that – as “folk” psychology suggests – a high level of systemizing quotient is related to good performance in programming. While Wray found a positive relation between SQ and a test for programming ability, and a negative relation between this same test and EQ, Coles and Phalp found no correlations between brain type (S-type vs E-Type) and programming performance, even if the brain type was related with the choice of the degree subject.

In this work, we consider a course of Computer Programming with Laboratory, corresponding to 12 ECTS (European Credit Transfer and Accumulation System), which is compulsory in the first year for the bachelor curriculum in Applied Mathematics at the University of Verona. In the past years, it has been observed that the exam in consideration constitutes a significant difficulty for the students and so a remarkable part of them decides to defer the exam at a later time, and others obtain a low grade.

Given to these difficulties, we decided that this sample could be a good starting point to analyse the relation between cognitive styles, specifically SQ and EQ, and performance in early programming learning. We consider the first two months of the activities where students learn the basics of imperative programming with Python. The aim of this research is to explore the relationship between brain type and performance in programming, also distinguishing between female and male students.

3. The Method

3.1. Participants

The research involved 46 university students (62% males, mean age: 19.75, SD 1.76) enrolled at the first degree in Applied Mathematics, attending the course is Computer Programming with Laboratory.

According to the information provided by the students, most of them come from a non technical high school “Liceo” (80%); also the majority of them (70%) have attended a curriculum with scientific or technical emphasis. Even so very few of them have experience in Programming. As matter of fact Informatics was a school subject for a third of them only and an independent subject for a quarter; and, with few exceptions, it is a marginal theme. About coding and programming languages, barely a quarter of students had some experience about. But only a few of them (less than 4%) declare an appreciable knowledge of a programming language.

3.1.1. Brain Type

To evaluate the brain type, we used the short version of the Empathy Quotient (EQ-Short) and the Systemizing Quotient (SQ-Short) (Wakabayashi et al., 2006).

This version includes a total number of 60 items, 20 for the EQ and 20 for the SQ and 20 fillers. We used the Italian translation of items from the Autism Research Center(Autism Research Centre, 2016),
Table 1 – Mean value of the variables (standard deviation in brackets), separately for male and female students

<table>
<thead>
<tr>
<th></th>
<th>M</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>S-Quotient</td>
<td>20.3 (6.57)</td>
<td>18.75 (5.7)</td>
</tr>
<tr>
<td>E-Quotient</td>
<td>17.11 (4.57)</td>
<td>21.31 (6.11)</td>
</tr>
<tr>
<td>TH Score</td>
<td>0.73 (0.19)</td>
<td>0.79 (0.17)</td>
</tr>
<tr>
<td>PR Score</td>
<td>0.52 (0.26)</td>
<td>0.54 (0.29)</td>
</tr>
</tbody>
</table>

when available, and an Italian translation of the residual items proposed by the research group, also in collaboration with some experts on the topic.

Given the fact that we need a quick version, we decide to remove the fillers. Respondents have to choose their level of agreement on a 4-point Likert Scale. Scores can range from 0 to 40 for each scale.

3.1.2. Performance

The didactic work was consists of two parts:

- a “theoretical” activity, where the teacher explains the basics of programming and proposes some practical exercises in preparation of the the programming part to be developed in the laboratory;

- and a “programming” practice, where, after a short introduction, some problems are proposed to the students that have to solve them following the suggestion of the teacher and with a support of the assistant.

During the practical activity, we follow an adapted version of eXtreme Apprenticeship methodology (Vihavainen, Paksula, & Luukkainen, 2011) as described in some previous works (Solitro, Zorzi, Pasini, & Brondino, 2016) and inspired by the work of the Bozen University team (Dodero and colleagues) (Del Fatto, Dodero, & Gennari, 2014), (Del Fatto, Dodero, & Lena, 2015), (Dodero & Di Cerbo, 2012).

After the first two months, the participants took a partial exam that is structured as follows.

1. A few short question about the general concepts of the discipline.

2. Three exercises of coding where the students have to show their skills in
   - understanding a coded solution of a problem,
   - code an algorithm respecting give specification,
   - and, finally, given an informally described problem, define its specification and code the solution.

Performance was operationalized in terms of the score obtained in two different parts of the partial exam: the theoretical score (TH), and the programming score (PR). These scores have been standardized to be in the interval 0 – 1.

4. Results

Descriptive statistics in the study variables, separately for male and female students, are shown in Table 1.

Considering the EQ–SQ scores, in comparison with the scores of students of scientific faculties from previous researches (Wakabayashi et al., 2006), the mean values of both quotients were similar: mean EQ was 18.83 ($SD = 5.47$), mean SQ was 19.89 ($SD = 6.97$).
An independent sample t-test was performed to verify whether male and female students differ in some of the scores (S-Quotient, E-Quotient, TH score, PR score), and the only significant difference was found on the EQ score.

In a second step, in order to better analyze the combined effect of the two factors, sex and SQ-SE score, a mixed ANOVA was run, with “sex” as the between factor and “test” as the within factor, with two level (SQ vs EQ).

Results showed no main effects of the two factors. A significant interaction effect between the two factor, sex and test: $F(1,41) = 5.77, p < .05, \eta^2 = .123$, which is described in Figure 1.

Male and female students had a different trend in the two quotients, with males showing a higher score in the SQ than in the EQ one, and female students showing the opposite result.

This result is consistent with the literature (Baron-Cohen, 2002), (Baron-Cohen, Knickmeyer, & Belmonte, 2005): Baron-Cohen and colleagues state that type S-brain, that is the brain of people with SQ higher than EQ, is more frequent in male population, whereas the opposite, that is E-brain, is more frequent in female population. Our sample shows the same trend.

Concerning performance, students reached higher results in the theoretical part than in the programming part, and this result was the same for male and female students. The mixed ANOVA test showed only the main effect of the type of exam: $F(1,38) = 44.47, p < .001, \eta^2 = .54$; nor effect of sex neither of interaction sex type of exam was found.

Our second aim was to verify the connection with the brain type and the performance. First, a simple correlation test was carried on, considering the whole sample, and no correlation was found between SQ, EQ and performance. Second, the same correlations have been computed separately for male and female students. Correlation matrix is shown in Table 2.

No correlation was found for the male sample (excepted the correlation between theoretical and programming scores). In the female sample, a higher level of Systemizing Quotient is associated with a higher score in the programming part of the exam ($r = .66, p < .001$; see Figure 2.

Also the correlation between SQ and theoretical score was positive ($r = .43; see Figure 3, even if this correlation was not significantly different from zero ($p = .11$).
Table 2 – Pearson’s correlation coefficients for the four variables, by sex

<table>
<thead>
<tr>
<th></th>
<th>SQ</th>
<th>EQ</th>
<th>TH Score</th>
<th>PR Score</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>M</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQ</td>
<td>1</td>
<td>0</td>
<td>-.12</td>
<td>-.03</td>
</tr>
<tr>
<td>EQ</td>
<td>1</td>
<td>.37</td>
<td>.28</td>
<td></td>
</tr>
<tr>
<td>TH Score</td>
<td>1</td>
<td>.60 ($p &lt; .01$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PR Score</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>F</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQ</td>
<td>1</td>
<td>.34</td>
<td>.43</td>
<td>.66 ($p &lt; .01$)</td>
</tr>
<tr>
<td>EQ</td>
<td>1</td>
<td>.11</td>
<td>.05</td>
<td></td>
</tr>
<tr>
<td>TH Score</td>
<td>1</td>
<td>.65 ($p &lt; .01$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PR Score</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 2** – Correlation between Systemizing Quotient and Programming Score.

**Figure 3** – Correlation between Systemizing Quotient and Theoretical Score.
5. Conclusions

This study aims to explore the role of cognitive style – considering the S-E Theory framework – in predicting performance in programming courses for non vocational students.

The first result showed, in line with the literature, that male and female students differ in EQ, with women scoring higher than men.

Concerning the hypothesis that a high level of Systemizing Quotient lead to a better performance, this hypothesis has been confirmed only for the female subsample, and mainly for the programming practice part.

On the contrary, no correlations were found for the male subsample, nor for the theoretical part nor for the programming practice part.

A limitation of this study concerns the small sample size, and we are still gathering data on this topic to cope with this limitation.

On the other side, given the relevant role of affective and motivational dimensions in predicting performance in university students (Brondino, Raccanello, & Pasini, 2014), it is worth examining how motivational beliefs and achievement emotions are related to the constructs examined in this work.

6. References


Towards an IDE to Support Programming as Problem-Solving

Nicholas Nelson  
EECS  
Oregon State University  
nelsonni@oregonstate.edu

Anita Sarma  
EECS  
Oregon State University  
anita.sarma@oregonstate.edu

André van der Hoek  
Department of Informatics  
University of California, Irvine  
anre@ics.uci.edu

Abstract

Programming is inherently a problem-solving exercise: A programmer has to create an understanding of the situation, externalize and contextualize thoughts and ideas, develop strategies on how to proceed with the task, enact changes according to the most appropriate strategy, and reflect to learn from each problem. Therefore, programming is clearly more than just code input, testing, and maintenance. Current Integrated Development Environments (IDE), however, largely focus on the "writing code" parts of programming. In this position paper, we revisit which activities and actions constitute programming, and highlight six challenges to supporting these activities. We then briefly describe a new paradigm of interacting with the IDE on which we are working to more directly support each of the six activities.

1. Programming as Problem-Solving

Programming is more than dealing with language syntax and semantics: it is inherently an exercise in problem-solving that extends beyond the act of editing code in an Integrated Development Environment (IDE). We are not the first to observe this. For instance, programming has been characterized as an iterative process of refining mental representations of computational problems and solutions and expressing those representations as code (Loksa et al., 2016). Other studies have found evidence for the facts that programming requires gathering information from multiple sources (Sillito et al., 2008), includes creating mental models of program structures (Von Mayrhauser & Vans, 1995), and involves exploring and evaluating many alternatives (Hartmann et al., 2008).

We surveyed the literature from the perspective of programming as problem-solving, with Table 1 summarizing key activities that developers employ when programming. These activities can be partitioned into six categories (Activities), with specific actions that represent in more detail how the high-level activities manifest themselves in practice (Actions). Clearly, not every task involves all of these problem-solving actions, and there is no linearity to the order in which they are employed. Sometimes an action may not even be observable when it takes place solely in a programmer’s head. At the same time, literature has documented that all of these actions do occur and play an important role in how programmers arrive at a solution to the programming problem at hand.

2. Challenges

We believe that it is necessary to fundamentally rethink IDEs, so that they seamlessly and intrinsically support programming as problem solving. Supporting the full set of actions comprising problem-solving in programming, however, involves numerous challenges that must be addressed. These challenges span all six categories of activities, and even those activities that have traditionally been supported by IDEs (A4) exhibit gaps when reviewed through the lens of problem-solving requirements.

1. How to support programmers’ formulation of problems and reflection on potential solutions?

Programmers do not just arrive at a solution out of nowhere. They need to first contextualize the computational problem in terms of what they know and how they can progress towards a possible solution. This involves exploration, articulation, and reflection on different alternatives, with these actions being interleaved, sometimes even happening at the same time (e.g., developers are known to reflect on a code solution while they articulate it) and typically encompassing several relatively quick iterations. Often there is no single correct solution, and the best solution requires mixing and matching elements from multiple alternative solutions.
### Table 1 – Activities and Actions of Programming as Problem-Solving

<table>
<thead>
<tr>
<th>Activities</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1 Understanding the situation</td>
<td>Identifying goals</td>
</tr>
<tr>
<td></td>
<td>Recalling prior knowledge</td>
</tr>
<tr>
<td></td>
<td>Constructing models</td>
</tr>
<tr>
<td></td>
<td>Interpreting code artifacts</td>
</tr>
<tr>
<td></td>
<td>Filling knowledge gaps</td>
</tr>
<tr>
<td>A2 Externalizing thoughts &amp; ideas</td>
<td>Representing relevant information</td>
</tr>
<tr>
<td></td>
<td>Contextualizing information</td>
</tr>
<tr>
<td></td>
<td>Preserving contextual information</td>
</tr>
<tr>
<td>A3 Developing strategies</td>
<td>Generating alternatives</td>
</tr>
<tr>
<td></td>
<td>Articulating and refining alternatives</td>
</tr>
<tr>
<td></td>
<td>Understanding and assessing alternatives</td>
</tr>
<tr>
<td></td>
<td>Recombining aspects of alternatives</td>
</tr>
<tr>
<td>A4 Enacting change</td>
<td>Translating strategies to actions</td>
</tr>
<tr>
<td></td>
<td>Tracking progress</td>
</tr>
<tr>
<td></td>
<td>Evaluating and assessing change</td>
</tr>
<tr>
<td>A5 Collaborate</td>
<td>Feedback solicitation</td>
</tr>
<tr>
<td></td>
<td>Team work</td>
</tr>
<tr>
<td></td>
<td>Group think</td>
</tr>
<tr>
<td></td>
<td>Leverage group knowledge</td>
</tr>
<tr>
<td></td>
<td>Synchronization</td>
</tr>
<tr>
<td>A6 Retrospect</td>
<td>Reflect on work</td>
</tr>
<tr>
<td></td>
<td>Preserve work</td>
</tr>
</tbody>
</table>

2. **How to provide programmers access to the relevant context in a problem space?** Programming solutions must exist in the context of the rest of the codebase and its related artifacts. Programmers need to understand where a code snippet fits in a code base, what it calls out to, and what calls into it (de Souza & Redmiles, 2008), the desired behaviors of the existing code and the code to be produced (e.g., computational speed, usability, features), organizational policies (e.g., licensing, process standards, code style), and historical development (e.g., has a solution previously been tried and rejected). Information that defines this context is not always readily available and, instead, must be cobbled together from multiple different types of and sources of artifacts. A developer needs to know where these individual pieces of information reside and how to obtain just those items that actually pertain to the problem at hand.

3. **How to support different information processing style and workflow of programmers?** Programming is a creating activity; no two programmers arrive at the same solution in the same way. For example, female programmers tend to process information comprehensively, seeking a more complete understanding of the problem before starting (Grigoreanu et al., 2012), whereas male programmers are known to use a more heuristic (or selective) approach. As another example, visuospatial reasoning is critical for abstract knowledge and inference, and is a core component of how we view the world, but it has been observed that programmers bring their own personal visuospatial reasoning and information processing style to evaluating and organizing artifacts, solutions, and ideas (Tversky, 2005).

4. **How to support programmers in relying on past experience?** Problem-solving in programming is not a one-off activity performed in isolation. Instead, programmers rely upon past experiences with similar problems, knowledge gained from previous artifact interactions, and prior mental models. The question arises of how this prior experience can be brought to bear, easily, when a programmer faces a new situation. Apart from the programmer memorizing what they might have done in the past and looking this up, current IDEs provide no support in this regard. The challenge lies in how a new IDE can provide this kind of assistance.

5. **How to enable collaboration between programmers across all artifacts involved in problem solving?** As already highlighted, programmers contextualize their work with all sorts of different ar-
tifacts. Yet, current IDEs generally only allow sharing of the code being worked on. Successful collaboration, however, requires sharing of all different artifacts so all participants have access to the full context. Moreover, it requires ongoing programmers to know how their changes may affect others, and who may be making changes that affect their own work (de Souza & Redmiles, 2008). Finally, they also need to understand the provenance of design decisions, and how and why these decisions were made.

6. **How to utilize different pieces of information and context to support the act of coding?** Solutions to computation problems must eventually be represented in code. Converting a conceptual solution into actual lines of code is a non-linear activity (coding sessions start and stop), occurs concurrently with other problem-solving activities, and is loosely organized (e.g., solutions are partially implemented, abandoned, and recovered). Creating a simple, elegant software solution hinges on complex, exploratory coding sessions, and the IDE must support this process fluidly.

**Figure 1 – Cards-based User Interface of a Problem-Solving IDE**

3. **Toward A New IDE**

We are in the initial stages of rethinking what an IDE might look like when it is designed from the ground-up to support programming as problem solving. We particularly are exploring new metaphors for how users would interact with the IDE (compared to the bento-box paradigm underneath today’s IDEs), as rooted in the specific actions that we know programmers engage in and that must thus be supported (conform Table 1). At present, the leading metaphor for our design is that of cards (as inspired by code bubbles (Bragdon et al., 2010)): the IDE presents all information and enables the developer to make changes through individual cards (see Figure 1). We believe cards, and the affordances that they have, allows the envisioned IDE to better match what programmers actually do. We briefly highlight five key elements in this regard together with which challenges (C#) we identified they would address.
- Cards serve as containers for different entities created during problem-solving sessions. They span different artifacts (e.g., an open issue that is being worked on (Card 1), sketches that capture designs (Card 2), or code snippets for reviewing and editing (Cards 3–6)). This diversity of cards is crucial, as it unifies how programmers interact with different types of information that they need in formulating problems and reflecting on them (C#1).

- Cards can be spatially arranged or grouped by programmers in the open canvas. Since cards can be grouped in different ways, this supports different information processing styles and workflows (C#3). For example, cards can be placed side-by-side (Cards 4–6) allowing for the exploration, reflection, and mixing and matching of alternative solutions (C#1). Such comparative reasoning has been shown to be critical for creating good solutions (Hartmann et al., 2008).

- Cards can be organized in groups or stacks, which allows programmers to organize the different entities to create the context that is needed in the problem solving exercise (C#2). For example, cards can be organized spatially (Cards 3–5) such that all relevant cards for a given context are visible at the same time, or they can be stacked (Cards 7–9) to consolidate cards that are relevant, but are not immediately needed.

- Card groupings (in groups or stacks) are type-agnostic, that is, cards of different types can be combined together (Cards 7–9). Cards can also reflect programming elements at different points in time (Cards 4, 5), allowing comparisons between different versions of the same artifact to provide historical context to the current change (C#4).

- Cards are created within an individual programmer’s workspace, but are easily shared between different programmers (e.g., Cards 3–5 are shared with two teammates: Mark and Sally). Further, each card is dynamic, automatically updating itself to reflect the latest changes, which provides an awareness of what other programmers are currently working on (C#5), thereby allowing developers to create some form of workspace awareness (Da Silva et al., 2006).

To date, we have merely engaged with the 'theoretical' exploration of these ideas. Our next step is to construct an actual prototype implementation to take them forward toward supporting programming as problem solving.

4. References


On the Nature of Programmer Expertise

Chris Parnin
NC State University
Raleigh, NC, USA
cjparnin@ncsu.edu

Janet Siegmund
University of Passau
Passau, Germany
siegmunj@fim.uni-passau.de

Norman Peitek
Leibniz Institute for Neurobiology
Magdeburg, Germany
npeite@lin-magdeburg.de

Abstract
Many experts in fields such as mathematics, medicine, and chess display intellectual marvels undiminished with age. However, software engineers, much like athletes, seem to have a limited lifetime for applying their expertise. Compared to other areas of expertise, the elements of which programming expertise is built upon is unstable, short-lived, and often non-transferable. In this position paper, we derive insights from psychology, cognitive neuroscience, and decades of software engineering research on expertise. Using these insights, we strive to understand what representations, strategies, and cognitive processes and mechanisms experts use when performing exceptional programming feats. In particular, we want to understand how expertise shapes an expert’s mind, and understand the intricate patterns and strategies that expert programmers hone over the years. To answer these questions, we propose to use several brain-imaging techniques to study expert software engineers. Finally, based on these results, we wish to derive guidelines in order to help companies and teachers in identifying and training programmers to quickly adapt to changes in terms of languages, projects, teams, and techniques.

1. Introduction
Some of the most amazing feats are performed by human experts, who demonstrate a mastery of skills obtained through many years of deliberate practice (Ericsson & Lehmann, 1996). As an expert, programmer’s eyes glide across source code of a program, and in just seconds, they can extract a deep understanding from abstract symbols and text arranged in program files, which may take a novice programmer a good part of a day to truly understand. Experts’ eyes dance around source code, finding points of interest, such as method signatures, and follow relationships such as data-flow relationships between program elements. They can spot typical errors like an off-by-one error incredibly fast, demonstrating that they have abstracted the comprehension process to be more efficient. Expertise in programming, as in any other areas, requires years and years of continuous and deliberate practice. But even after extensive experience in programming, there are still tremendous differences in programmer expertise: So called 10xers can be 10 or more times as productive as other programmers who have spent an equal amount of time with programming (McConnell, 2010; Sackman, Erikson, & Grant, 1968). Likewise, there are novice programmers who can grasp the concepts much faster than their peers, but until today, we can only hardly explain why such differences occur.

Ever since the emergence of software engineering as a discipline, having programmer expertise on board of a project has tremendous importance. However, still today, it is difficult to evaluate whether a programmer will be a good addition to a software project, especially when different software technologies, architectures, and products may be involved. Despite decades of research on expertise and program comprehension, there is a general lack of understanding of what programmer expertise actually is. Solving this problem is of fundamental importance. We believe that an understanding of the fundamental human limitations associated with gaining expertise in programming skills will enable developers and companies to more effectively invest in technology choices and training strategies. For example, the ten-thousand rule¹ is a common rule of thumb for estimating how long it takes to gain the highest level of expertise. If an expert Java server-side developer needs to transition into a new role developing front-end web technologies, does the developer also need to reinvest ten years of effort before regaining proficiency or is it just six months?

¹The ten-thousand rule states that it takes 10,000 hours of practice of a skill to become an expert.
Why is it so difficult to understand programmer expertise? In this position paper, we will bridge together several insights into expertise and also define several research experiments that will help us find a way out of this dilemma. To this end, we will first look at expertise from different angles:

- **Cognitive Psychology:** Cognitive psychology describes how humans create information from all the data surrounding them. It matured as a discipline over many decades, so it offers lots of relevant insights, for example, how learning strategies differ and how experts are created. Expertise in many areas, including chess and tetris, is a well-studied phenomenon. Thus, this mature research will help us to better understand the nature of programmer expertise.

- **Neuroscience:** Neuroscience uses neuro-imaging techniques, such as fMRI, EEG, CT, to relate brain activation to cognitive processes. This area is relatively new, but it extends from cognitive psychology, in that it can interpret the brain activation in relation to the well-studied cognitive processes. For example, results show that experts use their brain more efficiently than novices, such that they use a specialized brain area. Expert golfers need just one small area to perform a golf swing, but novices use a large network of brain areas (Milton, Solodkin, Hlustík, & Small, 2007).

- **Software Engineering:** Also in software engineering, expertise has been studied, but rather superficially in comparison to cognitive psychology and neuroscience. Specifically, researchers have been studying how tasks, tools, and complexity affect programming productivity, but the relationship to programmer expertise has not been well-studied. Thus, re-evaluating this research in its potential to shed more light on programmer expertise will give us valuable insights, also into why there was and is so little progress on programmer expertise.

Eventually, we expect that the research that we outline in this paper may eventually enable better guidelines for training and measuring expert programmers. For example, with a set of validated fMRI biomarkers of effective programming expertise, we can improve educational interventions by identifying deviations from desired brain activation when reasoning about a particular programming problem or concept such as recursion.

### 2. Expertise

In this section, we highlight the insights from cognitive psychology, neuroscience, and software engineering. Afterward, we describe how these different viewpoints can help us answer about expertise, and establish a roadmap toward understanding programmer expertise.

#### 2.1. Insights from Cognitive Psychology

##### 2.1.1. Deliberate Practice

Research in cognitive psychology revealed that experts demonstrate a mastery of skills obtained through many years of deliberate practice (Ericsson & Lehmann, 1996). In understanding what makes an expert an expert, scientists have studied the training strategies, cognitive representations, and problem-solving strategies used when performing tasks with exceptional skill. Initial research focused on the training strategies and researchers found that there is a consistent difference between experience (in terms of years spent) and expertise (performance levels) (Camerer & Johnson, 1991; Shanteau & Stewart, 1992)—the primary difference in performance arises from how experts were trained and not necessarily how long. For example, when comparing chess experts who have spent equal time in gaining experience, the consistently best performers are the ones who repetitively studied specific chess positions and scenarios, as opposed to lower performers who just practiced in tournaments (Charness, Krampe, & Mayr, 1996).

##### 2.1.2. Problem Representation

Cognitive expertise involves chunking of information, or organizing a stream of perceptual cues into a more meaningful pattern (De Groot, 1978). Experts use more effective problem representations and generate better “next steps or moves” (in chess) (Simon, 1990) or select the best diagnostic option (in medicine) (Elstein, Shulman, & Sprafka, 1990). Experts differ from novices in how they process infor-
mation and arrive at an answer, such that they look a bit deeper and process next steps faster (Holding, 1992), resulting in improved qualities of answers (Elstein et al., 1990).

2.2. Insights from Neuroscience

2.2.1. Neural Efficiency and Cortical Differences

Studies of brain activity find that experts demonstrate more efficient neuronal firing patterns than novices with the same tasks (Neubauer & Fink, 2009). Expert brains work differently than non-expert brains. When novices are compared with experts performing the same kinds of tasks, the differences can be remarkable. When novice golf players try to perform a golf swing, their brains are alight with activity throughout many areas of the brain as they clumsily try to coordinate the swing in their mind, whereas experts have conceptualized the movements of a golf swing into a simple, focused, and energy-efficient action in the brain (Milton et al., 2007). Not only does an expert’s brain act more efficiently, experts sometimes also have a larger brain mass in these areas. A larger right posterior parietal cortex is seen in expert video game players (Tanaka et al., 2013). Experienced London taxi drivers have larger parahippocampal regions with size correlated with years of experience (Maguire, Woollett, & Spiers, 2006).

2.2.2. Specialization

Although the brain appears to have specialized areas for specialized tasks, when humans develop new skills, there is often no specific area of the brain that supports that skill. Instead, learning processes often recruit existing information-processing networks of the brain in support of the new skill. For example, the fusiform face area is strongly associated with face perception, but it also gets recruited when identifying a specific object, such as for bird experts who can distinguish between a vast variety of bird species or for car experts who can identify distinct differences between many models and makers of cars. Interestingly enough, bird experts who are not car experts do not use the fusiform face area when observing cars, and vice versa (Gauthier, Skudlarski, Gore, & Anderson, 2000). Other studies also show the involvement of the fusiform face area in experts of tasks that require visual perception, including the categorization of chest radiographs (Harley et al., 2009) or understanding chess positions (Bilalić, Langner, Ulrich, & Grodd, 2011).

2.3. Insights from Program Comprehension Studies

Over the past few decades, several theories of program comprehension have been proposed and empirical studies attempting to validate them have been performed.

2.3.1. Syntax vs. Semantics

Software engineering researchers have proposed that programmers use knowledge structures that encode semantic (Shneiderman & Mayer, 1979) and domain information (Brooks, 1983) about a program as well as prime structures (Linger, Mills, & Witt, 1979), that include elements of syntax, control-flow and data-flow (Pennington, 1987b) of the program. These knowledge structures (Rich, 1981) have been formalized referred to as programming plans. Motivation for programming plans was inspired from theoretical constructs in text comprehension, such as scripts, which are mental representations of common activities (e.g., eating in a restaurant) and can aid humans in understanding and remembering narrative text (Bower, Black, & Turner, 1979). Programming plans act like schemas that are first instantiated and then its slots are filled with concrete values as a programmer builds an understanding of the code (Soloway, Ehrlich, & Bonar, 1982). Plans may help programmers fill in the “gaps” when trying to understand code. Finally, it was proposed that programs follow basic rules of discourse and that any violation to “accepted conventions of programming” should as a result hamper an expert’s ability to use programming plans (Soloway & Ehrlich, 1984).

Evidence that expert programmers have different mental representations from novices has been described in several studies, however not all evidence is consistent with the theory of programming plans. In a series of studies, participants were asked to understand a piece of code and later recall text of the program. Experts recall programs better than novices when the order of presentation is correct (Shneiderman, 1976), but performance difference disappears when programs are presented in random order. Further, when examining the details of what is recalled (Shneiderman & Mayer, 1979),
researchers found that experts could recall semantic information about source code, but incorrectly recalled the exact details. Novices did the opposite: They could more accurately replicate the source code syntax, but often mistook the meaning of the source code. In another study, when categorizing related code snippets, experts and novices differed in their organization (procedural similarity vs. syntax similarity) (Adelson, 1981). Finally, Soloway and Ehrlich (Soloway & Ehrlich, 1984) evaluated the theory of rules of discourse by varying the style of the snippets, such that there were versions that followed typical coding conventions (plan-like) and versions that explicitly violated such conventions (unplan-like). For example, they changed the variable naming, such that in the violated version, the naming did not convey the purpose of the variable, but rather the opposite (max was renamed to min). The results showed that novice programmers were not affected by the violated coding conventions. However, experts were significantly slower with these version and made significantly more errors—specifically, experts became as slow and as incorrect as novices.

However, another series of studies cast doubt on the nature of programming plans. Gilmore and Green failed to replicate Soloway and Ehrlich’s previous results (Soloway & Ehrlich, 1984) when using programming plans from Pascal programs with Basic programmers (Gilmore & Green, 1988). They suggested that programming plans may not generalize across different languages, and that plans cannot represent the underlying deep structure of programs. Bellamy and Gilmore (Bellamy & Gilmore, 1990) examined the protocols generated from experts in different languages as they created programs. Using two different models of programming plans, they found neither model was well supported by protocols; further, different programming language experts generated different types of representations. Finally, Pennington (Pennington, 1987b) theorized that if programmers form plan-based mental representations, then they should recognize lines faster when preceded by lines from the same plan structure. Unfortunately, in the study, stronger priming effects were observed from syntax structure vs. plan structure. Subjects also made fewer errors on control-flow questions, compared with data-flow and functional questions. Pennington concluded that:

While plan knowledge may well be implicated in some phases of understanding and answering questions about programs, the relations embodied in the proposed plans do not appear to form the organizing principles for memory structures.

2.3.2. Strategies

Several software engineering researchers have noted that expert programmers make better use of strategies and are more fluid in adapting strategies when compared to novice programmers. Previous theories of program comprehension have proposed two primary mechanisms: top-down and bottom-up program comprehension. **Top-down comprehension** is a hypothesis-driven process, in which developers initially form hypotheses about source code and, by looking at more and more details, refine these hypotheses subsequently, until they form an understanding of the program (Brooks, 1978). The programmer is guided by using cues called beacons that are similar to information scents in information foraging theory (Pirolli & Card, 1999). With **bottom-up comprehension**, developers start with details of source code and group these details to semantic chunks, until they have formed a high-level understanding of the program (Shneiderman & Mayer, 1979). With opportunistic and systematic strategies (Littman, Pinto, Letovsky, & Soloway, 1987), programmers either systematically examine the program behavior or seek boundaries to limit their scope of comprehension on an as-needed basis. Von Mayrhauser and Vans offered an integrated metamodel (von Mayrhauser & Vans, July 1993) to situate the different comprehension strategies in a single model scheme. Finally, Murray and Lethbridge (Murray & Lethbridge, 2005) have proposed that programmers develop and make use of specific strategies for specific types of problems rather than conforming to a general framework (e.g., the strategy needed for understanding the cultural practices of a team is distinguished from getting the big picture of a program).

Evidence for different strategy usage by experts and novices has been observed in several studies. Shaft and Vessey evaluated how program-comprehension strategies depend on expertise in a program’s domain, and found that depending on familiarity, programmers use completely different strategies (Shaft
& Vessey, 1995). These results were consistent with top-down (familiar domain) and bottom-up (unfamiliar domain) comprehension models that were state of the art during that time. There are several studies that evaluated these comprehension models and how context drives the comprehension process that developers choose (Brooks, 1978; Widowski, 1987; Pennington, 1987a). In fact, some studies suggest that the ability for an expert to switch strategies is a more important factor than knowledge representation. For example, Widowski (Widowski, 1987) compared novice and expert programmers working on unplan-like and plan-like programs of varying semantic and syntax complexity. Again contradicting Soloway and Ehrlich’s results (Soloway & Ehrlich, 1984), experts actually did better on unplan-like programs. Further analysis of protocols found that experts were able to adjust their strategy (structure-oriented vs. variable-oriented) more than novices in order to do account for unplan-like programs. Finally, Vessey (Vessey, 1987) observed when debugging a program, unsuccessful programmers used an erratic mix of reinspections and navigations, whereas successful programmers maintained a smooth progression through the program’s execution. Further, effective strategy use could account for 74% of debugging time, whereas chunking ability could only account for 31%.

In conclusion, when considering programming plans and strategies, Gilmore (Gilmore, 1990) has argued that “plans” are insufficient for explaining expertise alone, and instead, that multiple factors, including practice and strategy acquisition must be examined:

Expertise is not as simple as we might sometimes think. Although high-level, efficient representations of programming knowledge develop with experience, it seems that this knowledge is not the sole determinant of programming success. Besides chunking of knowledge structures, experts seem to acquire a collection of strategies for performing programming tasks, and these may determine success more than does the programmer’s available knowledge...Programming may be rather like riding a bike, or some other motor skill, without practice it cannot be mastered.

3. How to Find a Way into the Light

Thus, with everything that we learned, we can sketch a roadmap toward a thorough understand of programmer expertise. We learn from cognitive psychology the role of deliberate practice, indicating that how novices work with code plays a crucial role in their learning process. Neuroscience shows that experts use their brain differently when completing a task they have experience in. Research in software engineering revealed different aspects of code (e.g., plan-like vs. unplan-like) and knowledge of programmers determine how programmers work with code. Thus, combining these three areas will give us unique opportunities to understand the nature of programmer expertise.

Based on the insights from the different disciplines, we have derived hypotheses that we think are the most important to address in future projects. In this section, we discuss these hypotheses and outline experimental designs to evaluate them.

3.1. How Does Programming Shape the Expert’s Mind?

**Research Hypothesis:** Experts and novices have a different neuroanatomical representation, such that novices use a different network of brain areas.

**Rationale:** From brain imaging studies of experts in various fields, we know that experts in different occupations use different neural mechanisms and the insights we get from one kind of expert may not generalize to other types of experts. Different studies suggest that experts have encoded the process more efficiently in a single/few distinct brain area, possibly one that is typically used for other processes (like the fusiform face area, which is used by bird experts also to recognize birds). Programming is a more complex task, which consists of, among others, understanding code and writing code. Both processes are different, so we expect differences in brain activation. Thus, we expect that the study of expert programmers will reveal unique patterns of brain activations.

**Design:** To understand the neuro-anatomical representation of programmer expertise, we need to study
novice and expert programmers. A large portion of programming activities consists of, in essence, reading code. For this reason, we focus initially on code comprehension. A promising option is to build on the snippets that have already been designed for neuro-imaging studies, such as in reported in several brain imaging studies (Siegmund, Kästner, et al., 2014; Crk & Kluthe, 2014). Another way is to let participants categorize source code snippets based on functionality (Adelson, 1981), or identity correct or wrong implementations. Finally, another option is to use set of validated assessment instruments called concept inventories (Adams & Wieman, 2011), which provides the ability to assess if someone grasps the necessary concepts for expertise in a given field or domain.

We will collect block-based fMRI brain scans of expert programmers and novice programmers with programming and control tasks. Block designs are well suited to localize functional areas and study steady state processes, such as attention and comprehension, because they maximize the hemodynamic responses associated with cognitive processes, allowing for improved ability to locale brain areas. Using control tasks improves the ability to further isolate essential areas of cognitive for programming rather than supporting areas (e.g., eye movement while reading or locating syntax errors).

**Objectives:** The study and analysis will attempt to achieve the following objectives.

- **Identify brain regions associated with expertise in programming.** Understanding how programming shapes the expert’s mind will help us to get a deeper understanding of the nature of programmer expertise. We expect that we will see differences in activation between novices and experts, such that experts recruit one or a few areas to understand and create source code, while novices require a wide-spread network of areas. Furthermore, we expect also a difference between experts and 10xers, for example, regarding activation in an even more concise area. This will help us to spot experts and 10xers more easily, and light the path that novice programmers have to follow to become an expert.

- **Identify cortical differences associated with expertise in programming.** Structural MRI brain scans of expert programmers and control participants will be analyzed using voxel-based morphometry, a method that allows an automatic whole-brain analysis of gray-matter volume. Compared with controls, we expect that expert programmers will have greater gray-matter volume in the previously identified brain areas. This analysis will further validate our results, as it will demonstrate that expert programmers not only use a distinct set of brain areas, but that years of programming experience has indeed shaped the expert’s mind.

3.2. Which Theories of Programmer Expertise Can Be Supported?

**Research Hypothesis:** Top-down comprehension and bottom-up comprehension are represented in different networks of brain areas.

**Rationale:** Different representation of top-down and bottom-up processes have been observed in cognitive psychology and neuroscience. Since both processes share similarity with their psychological and neuro-scientific counterparts, we expect to observe a similar distinction in brain activation. Specifically, we expect memory-related activation, because top-down comprehension requires concepts to be stored in memory, and these should be retrieved during comprehension.

**Design:** To differentiate between top-down and bottom-up comprehension, we need to decompose each process into its stages. For example, bottom-up comprehension requires perceiving and processing a line of source code (visual perception, reading comprehension), retrieving meaning from the source code (problem solving), chunking statements to semantic units (synthesis), and finally concluding the purpose of source code (induction). To observe these stages in novice and expert programmers, we will show both groups obfuscated code, preventing experts from applying their knowledge about plans, strategies, and domains. Regarding top-down comprehension, programmers need access to their knowledge (memory retrieval), deduce the high-level purpose of source code (deduction), and confirm the purpose by examining details (memory retrieval, matching). To enable novices to actually use a top-down
approach, we intend to train them with specific snippets and according plans (e.g., that a decreasing loop counter often indicates a reversing operation) and then observe both novices and experts with the same snippets. This will also allow us to observe how expertise and the comprehension strategies interact with different brain activation. Possibly, we observe four different patterns depending on expertise (novice, expert), and strategy (top-down, bottom-up).

Objectives: The study and analysis will attempt to achieve the following objectives:

- **Identify strategy differences between experts and novices according in the comprehension strategy.** We will use multivoxel pattern analysis and hidden semi-Markov models (Anderson, Pyke, & Fincham, 2016) to identify distinct stages associated with program comprehension. This analysis will allow us to find differences in how long experts and novices spend time in different cognitive stages, and whether novices or experts introduce any distinct stages or ordering in their problem solving.

- **Identify differences in neural efficiency between experts and novices.** We examine deactivation strengths between different cortical areas in order identify neural efficiency (Durning et al., 2015). For example, during cognitive tasks, the default network is often deactivated in order to enable concentration on the task; the strength of the deactivation correlates with the difficulty of the task (Buckner, Andrews-Hanna, & Daniel, 2008). Analyzing the deactivation, such as the default network, allows us to observe when participants need to concentrate more in order to solve a task. Hence, we expect differences in deactivation patterns between experts and novices. Furthermore, we expect a stronger deactivation for bottom-up comprehension, compared to top-down comprehension.

4. Discussion

We discuss several implications and challenges for future research on programmer expertise.

4.1. Implications

In software engineering, in a classic study, Sackman evaluated typical tasks during programming, such as implementing and debugging code, and measured the time and correctness developers needed to complete such tasks (Sackman et al., 1968). He found that even for expert programmers, there are huge differences in performance, up to a factor of 25. On average, the efficiency varied by a factor of 10, leading to the term “10xer” for developers who are exceptionally good at their job. If we can understand how experts deliberately practice programming, we can devise more efficient training strategies for novice programmers and professional programmers to adapt their skills to new technologies.

There are several ideas that are promising. In the Pragmatic Programmer (Hunt, 2000), Hunt proposed the idea of practicing a code kata, which was an exercise in programming that can help a programmer practice a programming skill on a daily basis. Similarly, there has been a proposal to teach software engineering concepts via athletic “cross-fit” programming exercises (Hill, Johnson, & Port, 2016). The insight is that in cross-fit training, you try to best your previous performance record for a particular workout. Likewise, students learn to practice programming exercises (workout of the day), but as they practice the exercise, they measure their performance relative to an “expert”. In class, they be assessed with a new challenge and only receive credit if they could complete exercise within 20 minutes. Overall, the idea of targeted practice on a set of programming skills is consistent with effective training many other domains, such as chess, where deliberate practice on specific openings, and practicing generating moves for critical board positions was found to be more effective than just straight game play.

4.2. Challenges

**Measures of Expertise:** It is critical for the success of the study to choose appropriate and representative programmers for the novices and experts group in order to clearly distinguish cortical and strategic differences in expertise. However, it is a challenge to establish a sound measure of programming expertise, which does not simply rely on a rudimentary measure of experience. Many studies have used rather superficial and ad-hoc measures to define programmer expertise. For example, Soloway and Ehrlich categorized undergraduate students as novices, and graduate students as experts. However, a deeper investigation into the expertise levels of their participants did not happen. This makes it rather difficult
to fully understand how expertise actually influenced the study results.

In industry, technical interviews are an expensive and often unsatisfactory process. Alternative methods, such as standardized tests, requires considerable effort to develop and validate. For example, Bergerson and others describe an approach to develop such an instrument, which is based on established techniques that are rooted in psychology (Bergersen, Sjøberg, & Dybå, 2014). This test comprises several programming tasks in Java and has been validated with 65 programming experts over two days. Extending approaches for standardized measures of experience levels of programmers (Siegmund, Kästner, Liebig, Apel, & Hanenberg, 2014) is a start. Additionally, screening participants with technical interviews and standardized tests of programming expertise (Bergersen et al., 2014) can be a further filter. Conservatively, only programmers who pass all screening measures should be admitted to the study’s experts group.

A related challenge is to select 10xers from expert programmers. We expect the comparison between 10xers and experts will show further differences beyond the contrast of experts and novices. Thus, it is interesting to also include 10xers and compare them to average expert programmers. Of course, finding 10xers under experts is a similarly difficult task. The previous selection process might not be enough to identify 10xers from a group of expert programmers. To this end, recruiting participants with high ranks from leaderboards, such as HackerRank, or through supervisor’s recommendations can be promising.

**Material and Task Design:** For creating code, specifying the programming problem will be challenging. Focusing on too small problems (e.g., factorial), it may be difficult to see differences between novices, experts, and 10xers, but too large problems may again be impossible to implement, because they require too much time or lead to source code that is too long. Further, depending on the presentation and context of the programming problem, some programmers may adapt a different strategy when solving, which should be anticipated and controlled for. Using pilots to design programming tasks with a known set of strategies and suitable difficulty outside a fMRI scanner is a good initial step.

### 5. Conclusion
Different insights from cognitive psychology, cognitive neuroscience, and classic studies in software engineering can be combined in order to answer important challenges in understanding expertise of software developers. Research on cognitive psychology tells us that deliberate practice plays an important part toward becoming an expert. Neuroscience tells us that experts have specialized areas that support a very efficient representation of cognitive processes. Software engineering studies have demonstrated the importance of mental representation and strategies used by experts.

Studies in expertise are useful in guiding the program-comprehension research community. For example, we could examine more closely how overloading the working-memory capacity affects program comprehension or whether 10xers have above-average working memory, language skills, can better concentrate, or whether they use completely different approaches when working with source code. We may be able to find alternative ways to identify experts and even support training. For example, we could expect that programming experts have a certain activation pattern when working with code. Thus, if we find such a pattern in a job applicant, we can assume a certain level of expertise. Furthermore, this could facilitate educational interventions to improve desired regional brain activation in order to reduce cognitive errors and increase programmer expertise. In the long run, ideally these kinds of studies can finally contribute to answer long-asked and sometimes hotly-discussed questions: Should we start programming education with objects or functions first or teach generic language skills? What effect does the mother tongue have on learning programming? What makes programmers efficient? Can anyone become a good programmer? What does it take to become an exceptionally good programmer? How can we help novices to overcome the typical obstacles of learning to program? Of course, fMRI and also other neuro-imaging techniques are not a “silver bullet” that can definitely answer these questions, but they give a promising new and complementary perspective to our current understanding of program comprehension and the ongoing endeavor to improve the life of programmers.
6. References


Investigating high-achieving students’ code-writing abilities through the SOLO taxonomy

Ayman Qahmash
Department of Computer Science
University of Warwick
A.Qahmash@warwick.ac.uk

Mike Joy
Department of Computer Science
University of Warwick
M.S.Joy@warwick.ac.uk

Adam Boddison
National Association of Special Educational Needs
Adamb@nasen.org.uk

Abstract
Computer Science Educationalists have implemented educational taxonomies which enhance the pedagogy for introductory programming modules. The SOLO taxonomy has been applied to measure students’ cognitive abilities in programming by classifying students’ exam answers. However, SOLO provides a generic framework that can be applied in different disciplines, including Computer Science, and this can lead to ambiguity and inconsistent classification. In this paper, we investigate high-achieving students’ coding abilities and whether they tend to manifest specific SOLO categories. We address the challenges of interpreting SOLO and the limitations of code-writing problems by analysing three specific programming problems (Array Creation, Linear Search and Recursion) and solutions to those problems presented by a group of nine students. Results for the first programming problem show that six students’ responses fell into the highest possible category (Multistructural) and the remaining three were categorised in the second highest category (Unistructural). For the second problem, eight students’ responses fell into the Multistructural category, while only one response was categorised as Unistructural. For the third problem, two students provided Multistructural solutions and five students’ solutions were Unistructural, but two further students showed a lack of understanding program constructs in their solutions, which were then categorised as Prestructural.

Keywords: programming, code-writing, SOLO

1. Introduction
Educational taxonomies have been implemented in many educational domains to enhance pedagogy, assessments and teaching methods, all of which affect students’ learning, knowledge and skills. There have been many attempts to apply different taxonomies, and these have been valuable in providing insights into computer science education (CSE) to understand different educational factors. Well-developed educational taxonomies, such as Bloom, revised Bloom and SOLO (Bloom, 1956; Krathwohl, 2002; Biggs, 2014), have been applied to measuring students’ outcomes as well as to classifying exam questions based on what they are supposed to measure. Although an educational taxonomy provides a generic framework that can be implemented in various disciplines, educators may not always come to a constant agreement on classifications (Fuller, 2007). In this study, SOLO has been chosen for classifying students’ learning outcomes as SOLO provides a hierarchy for measuring assessments and classifying students’ responses.

This paper is structured as follows. A brief background of educational taxonomies are introduced, followed by a discussion of taxonomies within the context of Computer Science and our justifications for applying an educational taxonomy are discussed. Research questions, methods, procedures and analyses are outlined in the methodology section, and finally, results are presented in the discussion section.

2. Background
The structure of the Observed Learning Outcome (SOLO) taxonomy (Biggs, 2014) aims to distinguish students’ cognitive levels, which are required during their learning process. The first level is Prestructural (P), where a student is provided with a new problem and irrelevant information. At this stage, the student has not understood the problem and tries to use simple information to solve it. The second level is Unistructural (U), as the student starts to focus on one single aspect that can be used to solve the problem. The third level is Multistructural (M), where the student starts to understand more than one factor that may help to solve the problem. The fourth level is Relational (R), which focuses on the qualitative development as the student starts to understand and identify relations between several aspects. The fifth level is Extended Abstract (EA), where the student manifests the ability to...
hypothetically think about other new factors that may help to solve the problem. In addition, the student may show the ability to generalise, evaluate and/or apply the knowledge to other problems. Another widely used educational taxonomy is that of Bloom, which focuses on three main domains: cognitive, affective and psychomotor (Bloom, 1956). The first level is Knowledge which refers to a student’s ability to recall basic knowledge, facts, concepts and terms, whereas the second level, Comprehension, describes a student’s ability to understand, translate and interoperate facts and concepts. A student can demonstrate a meaningful description of a problem in their own words. The third level is Application which indicates that a student can apply abstract knowledge to a new problem. The fourth level is Analysis, where a student exhibits the ability to decompose a complicated problem into integral parts and to understand the relationships between all parts. The fifth level is Synthesis which describes a student’s ability to compose integral elements into a new meaningful solution. The highest level is Evaluation, which refers to making judgements based on acquired knowledge and experience. A revised Bloom taxonomy has since been introduced by Krathwohl (2002), and which provides a two-dimensional framework consisting of knowledge and cognitive processes. The revised knowledge dimension includes an extra fourth subcategory, compared to the original taxonomy. Similar to the original taxonomy, the cognitive process dimension consists of six levels. However, the revised taxonomy renames the categories as verbs, and Synthesis swaps places with Evaluation and is renamed to be Create as shown in Fig. 1.

Figure 1: Original and revised Bloom’s taxonomies.

2.1 Bloom in Computer Science

In the computer science field, several studies have been conducted to apply Bloom’s taxonomy to curriculum design, assessment design, and student outcome measurements. Dolog (2016) used Bloom’s taxonomy to revise a software engineering curriculum to meet required and desirable student skills and competencies. Johnson (2012) evaluated the assessment in a Linux course using a revised Bloom taxonomy by analysing all verbs that had been used in 10 quiz questions and 39 assignments. The results indicated that 99% of the quiz questions were about memorising, whereas 11 out of 39 assignments were categorised as knowledge (recall) and only two assignments were at the evaluation level. Johnson and Fuller (2006) conducted a study that applied Bloom’s taxonomy in order to investigate computer science students’ cognitive abilities. The study encountered a problem of inconsistent categorisations, for two reasons. Firstly, determining a cognitive ability, which required the student being assessed, needs a deep understanding of how a course can be taught. Secondly, it has been claimed that applying Bloom’s taxonomy to programming problems proves to be a challenging process due to insufficient frameworks and a lack of CSE knowledge on how to apply Bloom’s taxonomy (Whalley et al., 2006).

2.2 SOLO in Computer Science

SOLO has been applied to computer science and education, where student performance has been assessed in a few specific aspects of programming, e.g. assisting with students’ code comprehension, code writing and algorithm design. Lister et al.’s study (2006) introduced a taxonomy which provides an interpretation of how SOLO could be applied to students’ answers to code comprehension problems using multiple-choice questions (MCQs). However, MCQs were not adequate to elicit responses at the Relational level. Therefore, the study was extended to analyse different types of questions in which 108 students were asked to explain a segment of code in plain English, allowing students’ responses to be categorised (based on SOLO) by three academics. In addition, eight expert academics were asked to
answer the ‘explain in plain English’ questions in order to compare both students’ and experts’ responses on each SOLO level. Results showed that half of the students provided Multistructural answers, in which students were only able to explain the code line by line without indicating the purpose of the code. Meanwhile, seven out of eight experts provided answers that can be categorised at the Relational level. Later, Lister et al. (2010) applied SOLO to measure student performance in code writing, relying on Biggs (1999) verbs descriptions that are suitable for each level. In addition, Hattie and Purdie’s study (1998) provides examples of how SOLO can be applied to language translation. SOLO levels can be determined by how certain phrases are interpreted rather than by translating words in isolation without understanding either the relation between the words or the context. For example, word-by-word translation, which is Unistructural, might provide meaningful translation that does not reflect the purpose of the original phrase. In the context of code-writing questions, a student may provide a direct translation of a certain program specification which does not result in correct code, whereas applying some changes to produce translation which is close to a direct specification might result in valid code. Based on Hattie and Purdie’s theoretical framework, SOLO categories for code writing were proposed as shown in Table 1.

<table>
<thead>
<tr>
<th>phase</th>
<th>SOLO category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qualitative</td>
<td>Extended Abstract – Extending [EA]</td>
<td>Uses constructs and concepts beyond those required in the exercise to provide an improved solution</td>
</tr>
<tr>
<td></td>
<td>Relational – Encompassing [R]</td>
<td>Provides a valid well structured program that removes all redundancy and has a clear logical structure. The specifications have been integrated to form a logical whole.</td>
</tr>
<tr>
<td>Quantitative</td>
<td>Multistructural – Refinement [M]</td>
<td>Represents a translation that is close to a direct translation. The code may have been reordered to make a valid solution.</td>
</tr>
<tr>
<td></td>
<td>Unistructural – Direct Translation [U]</td>
<td>Represents a direct translation of the specifications. The code will be in the sequence of the specifications.</td>
</tr>
<tr>
<td></td>
<td>Prestructural [P]</td>
<td>Substantially lacks knowledge of programming constructs or is unrelated to the question.</td>
</tr>
</tbody>
</table>

Table 1: SOLO categories for code-writing tasks (Lister et al., 2010).

Initial analyses of 30 students’ code-writing answers were conducted to develop the proposed taxonomy. Students were asked to write code involving three conditional statements in which providing a direct translation for sequenced conditional statements was considered Unistructural. However, when students considered removing redundancy, solutions tended to increase on the SOLO scale, becoming Relational. The students’ responses fell into only Unistructural and Multistructural. However, a second analysis of a different code-writing question was conducted for 59 students. The question related to theatre ticket sales, and was more complicated than the previous question. In this case, two students’ responses were categorised as Relational. Although the proposed SOLO taxonomy provides a theoretical basis for analysing students’ approaches to answering code-writing questions, it is evident in the study results that levels of questions may limit students’ responses to certain SOLO categories. If a student is asked to write a program to assign a value to a variable and print out the value, it is clear that the student’s response will be Unistructural — there will be no chance to provide a response at any upper level. Thus, it has been recommended that further replications of this study applied to different code-writing questions be conducted (Lister et al., 2010).

Whalley et al. (2011) proposed a refined SOLO taxonomy which overcomes previous research limitations in which mapping a very contextual code-writing question to the previous SOLO taxonomy resulted in difficulties in maintaining consistent mappings (Lister et al., 2010). In this study, a grounded theory approach had been adopted to analyse nearly 750 students’ responses to three code-writing questions (Discount problem, Average calculation, and Printing a box of asterisks) in order to conduct a SOLO mapping. The mapping process started with developing empirical categories consisting of silent programming elements (SPEs) to extract program constructs, syntactical elements and code features by conducting constant coding of students’ codes. Coding process allow expert computer science educators to identify silent programming elements which could emerge from students’ code. Producing SPEs could be advantageous and is practical for different code-writing questions. The next stage was to extract broad features that reflect a general code quality which can appear in most code, such as code redundancy and efficiency. The extracted features can indicate the level of code abstraction
based on subjective evaluations. Finally, based on the SOLO taxonomy proposed by Lister et al. (2010), three researchers categorised students’ responses to investigate whether using SPEs makes the mapping process efficient.

The study produced a refined taxonomy because an issue regarding the definition of the Multistructural level had been raised during the analysis stage. A previous definition of Multistructural indicated that a ‘response represented a translation that is close to a direct translation. The code may have been reordered to make a “valid” solution’ (Lister et al., 2010). However, during the analysis of the Average calculation problem, some responses managed to provide a direct translation that was a correct solution, but which could be less integrated. While the response is categorised as Multistructural, it tends to be over-categorised and should be Unistructural. Therefore, Multistructural was redefined as ‘a translation that is close to a direct translation. The code may have been reordered to make a more integrated and/or valid solution.’

It is clear that Whalley et al. provide a rigorous methodology, conducting a grounded theory approach to analyse a large set of data which requires a constant coding process to produce SPEs that can be reproduced for different code-writing questions. The mapping process requires expert computer science educators who are capable of identifying multiple alternative solutions or SPEs in which common features can be extracted. Students’ responses might be classified as Unistructural, which should indicate at least a single concept or SPE, whereas a Multistructural response should indicate a student’s understanding of multiple concepts or SPEs, which may or may not provide an integrated solution. However, a Relational response should indicate that all concepts and SPEs have been integrated, manifesting a comprehension of the relationships between all elements and features. Computer science educators should understand that classifying students’ responses is based on the level of translated specifications that are required to satisfy code implementations. In other words, the level of required specifications in a certain question affects students’ response classifications but not necessarily that the classification could measure student knowledge.

It has been claimed that the mapping process used in previous research (Whalley et al., 2011; Jimoyiannis, 2013) has not been consistent in defining programming constructs at the Unistructural level. Therefore, developing the building blocks may overcome the previous research limitations in order to identify programming constructs for the Unistructural level only. The building blocks should be derived from the current course curriculum while considering the knowledge that has been acquired by students. Iterative and vector questions were analysed while applying the proposed building blocks and results showed that 44% of students’ performances achieved a Relational level and 3% were at a Unistructural level.

3. Methodology

Content analysis provides a systematic approach to understand and analyse documents, transcriptions, audios and videos. Bryman (2015) defines content analysis as an approach to quantify content based on predetermined categories in which analysis procedures should be systematic and replicable. Another feature of content analysis is that it can be integrated with other approaches (Bryman, 2015) such as, in our case, the SOLO taxonomy.

3.1 Research questions

- How to assess students’ cognitive abilities for code-writing problems?
- Do high-achieving students tend to manifest specific SOLO categories for code-writing problems?

Data consisting of nine students’ exam scripts from a level 1 programming course were selected based on the students’ performance in programming and mathematics. Students proved to achieve high performance based on their grades, therefore, we were interested to analyse their responses based on the SOLO taxonomy. The programming course covers programming fundamentals, Object Oriented Programming, design, constructions, and testing, using the Java programming language. Three code-writing questions were selected, each of which included different programming constructs. We adapted Whalley et al.’s (2011) analysis approach, as shown in Fig. 2, to develop the SPE for each question, to which students’ responses were coded by three independent researchers. The SPE could be identified based on syntactical elements. Then, each researcher extracted general constructs, elements and
features. Those features could be abstract and imply certain code quality. The final step was to use the developed SPEs to categorise students’ responses using the SOLO taxonomy. To ensure that all researchers followed the same analysis steps, the analysis procedure was developed and distributed. At the end, each researcher consolidated their findings and discussed issues that might have affected the mapping process.

Figure 2: Bottom-up analysis approach.

3.2 Code-writing problems
3.2.1 Array Creation
The first problem was about writing a method that takes a single integer, \( n \), as an argument to create an array of size \( n \) with random values between 0 and 100. Students should write valid code to demonstrate their knowledge of array declaration, initialisation and iteration. In addition, the code can be implemented using Java built-in Math or Random objects and their functions to generate random values to be stored in the array. We assumed that those objects had been introduced to the students in the course. However, the question included a non-direct translation of the specifications as the array must include values between 0 and 100 inclusive. In this case, Math objects have a Random method that returns a double value, greater than or equal to 0.0 and less than 1.0, which needs to be multiplied by 101 and converted to integer. Thus, the array will include values from 0 to 100. We decided that the question should be categorised Multistructural. Sample of Student solution are shown in Table 3.

<table>
<thead>
<tr>
<th>N</th>
<th>Code</th>
</tr>
</thead>
</table>
| 1 | public static int[] createArray(int n){
|   |   int[] newArray = new int[n];
|   |   for(int i=0; i<n; i++){
|   |   newArray[i] = (int)(Math.random() * 101);
|   |   }
|   | return newArray; |

Table 2: Sample of student solution for Array Creation problem.

Coding the students’ answers is the next step to develop the SPEs derived from students’ program code to identify program constructs and syntaxes that be used to implement such a program to solve a problem. As shown in Table 4, the main program constructs consist of method declaration, array declaration, iteration, initialization with random values, and a return statement. In most solutions, the methods were declared correctly to return the created array. However, some methods were declared to be ‘static’, which was not required in the question specifications. For the array declaration, all solutions declared the array with size \( n \) in a one line statement, which is more efficient than using two line statements. All solutions implemented the array iterations using one finite loop whereas there were two options to generate random values to be stored in the array. Both Math and Random Java objects were implemented, however some solutions were not able to generate random values between 0 and 100 including 100 as specified in the question.
Based on the resulting SPEs and features extracted from students’ codes, Table 5 shows the SOLO mapping.

<table>
<thead>
<tr>
<th>Construct</th>
<th>Feature</th>
<th>Solutions by Student’s number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method declaration</td>
<td>Typical</td>
<td>x x x x x x x</td>
</tr>
<tr>
<td>Array declaration</td>
<td>Efficient</td>
<td>x x x x x x x x</td>
</tr>
<tr>
<td>Array iteration</td>
<td>Finite loop</td>
<td>x x x x x x x x</td>
</tr>
<tr>
<td>Random value generation</td>
<td>Inclusive range</td>
<td>x x x x x x</td>
</tr>
<tr>
<td>Return statement</td>
<td>Included</td>
<td>x x x x x x x x x x x x x x</td>
</tr>
<tr>
<td>SOLO mapping (1st researcher)</td>
<td>M U M M U U U U U</td>
<td></td>
</tr>
<tr>
<td>SOLO mapping (2nd researcher)</td>
<td>M U M M U U U U U</td>
<td></td>
</tr>
<tr>
<td>SOLO mapping (3rd researcher)</td>
<td>R U R M U U U U U</td>
<td></td>
</tr>
<tr>
<td>Final and agreed SOLO mapping</td>
<td>M U M M U U U U U</td>
<td></td>
</tr>
</tbody>
</table>

Table 4: SOLO mapping for Array Creation problem.

### 3.2.2 Linear Search

The second problem was to write a method that takes an array and an argument, s, as arguments, and performs a linear search on the array finding the index when s is found or returning -1 if s is not found.

We agreed that the question’s specification can be translated directly and should be categorised as Multistructural.

All students demonstrated a clear understanding of the question and produced code that included main constructs. As shown in Table 6 in code number 2, the student’s code tended to have a redundant declared variable to be returned, thus we consider it as redundancy in the return statement.

Different constructs extracted from students’ code included method declaration, array iteration, selection and the return statement. Students’ solutions were then categorised based on derived features as shown in Table 8.

<table>
<thead>
<tr>
<th>N</th>
<th>Code</th>
</tr>
</thead>
</table>
| 1 | ```public int linearSearch(int [] searchArray, int s){
|   for(int i=0;i<searchArray.length;i++){
|     if(searchArray[i] == s){
|       return i;
|     }
|   }
|   return -1; ``` |

Table 5: Sample of student solution for Linear Search problem.

<table>
<thead>
<tr>
<th>Construct</th>
<th>Element</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method declaration</td>
<td>public int linearArray(int []) array, int s)</td>
<td>Typical</td>
</tr>
<tr>
<td>Array iteration</td>
<td>for(int i=0; i&lt;s; i++)</td>
<td>Finite loop</td>
</tr>
<tr>
<td>Selection</td>
<td>if statement</td>
<td>Valid condition</td>
</tr>
<tr>
<td>Return statement</td>
<td>int find = -1; Return find;</td>
<td>Redundant</td>
</tr>
</tbody>
</table>

Table 6: Constructs and features for Linear Search problem.
3.2.3 Recursive method

The third question was about writing a recursive method that calculates the sum of the differences between opposing pairs (i.e. the difference between A[0] and A[n-1], A[1] and A[n-2], etc.). The question aimed to measure a student’s ability to implement a recursive method, which is considered to be a difficult concept to be understood by novice programmers. Thus, we agreed to categorise this question to be Rational, as the question included additional complex constructs along with applying the recursion concept. A typical solution passes to the method an array together with a variable that keeps track of the array index that traverses incrementally from left to right. Then, it is important to have a second variable which keeps track of the array index that traverses in the opposite way. In addition, edges of the array must be checked, in order to calculate differences between the edges. Table 9 shows student code which meets the question’s specifications and is considered to be valid code, and Table 10 shows constructs, elements and features extracted from students’ code. Most important constructs which differentiate students’ solutions for the SOLO mapping are edges, difference calculation and recursive method invocation. Given the fact that the nature of recursion involves a degree of abstraction, novice students encounter difficulties implementing recursive methods (Wirth, 2014). Therefore, students’ solutions manifest different levels of SOLO categories ranging from the lowest to the highest (which is Rational in this question). Two students were not able to understand the question requirements and provided solutions lacking constructs related to the question. Table 11 shows students’ SOLO categorisations.

<table>
<thead>
<tr>
<th>N</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>public int oppPairs(int[] array, int pos){</td>
</tr>
<tr>
<td>2</td>
<td>int pos2 = array.length() - 1 - pos;</td>
</tr>
<tr>
<td>3</td>
<td>if(pos2 &lt; pos)</td>
</tr>
<tr>
<td>4</td>
<td>return 0;</td>
</tr>
<tr>
<td>5</td>
<td>else</td>
</tr>
<tr>
<td>6</td>
<td>return array[pos2] - array[pos] + oppPairs(array, ++pos);</td>
</tr>
<tr>
<td>7</td>
<td>}</td>
</tr>
</tbody>
</table>

Table 8: Sample of student solution Recursive problem

<table>
<thead>
<tr>
<th>Construct</th>
<th>Feature</th>
<th>Element</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method declaration</td>
<td>Typical</td>
<td>Public int oppPairs(int [] array, int pos)</td>
<td>Typical</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Public int oppPairs(int [] array)</td>
<td>Missing argument</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Public void oppPairs(int [] array, int pos)</td>
<td>Void method</td>
</tr>
<tr>
<td>Variable assignment</td>
<td>Efficient</td>
<td>int pos2 = array.length() - 1 - pos;</td>
<td>Efficient</td>
</tr>
<tr>
<td>edges</td>
<td>Valid</td>
<td>if (pos2 &lt; pos)</td>
<td>Valid</td>
</tr>
<tr>
<td>Difference calculation</td>
<td>Invalid</td>
<td>int diff = array[pos2] - array[pos] + array[pos2-j] - array[pos+i];</td>
<td>Invalid</td>
</tr>
<tr>
<td></td>
<td>Efficient</td>
<td>int diff = array[pos2] - array[pos] + oppPairs(array, ++pos);</td>
<td>Efficient</td>
</tr>
<tr>
<td>recursive invocation</td>
<td>Valid argument</td>
<td>oppPairs(array, ++pos)</td>
<td>Valid argument</td>
</tr>
<tr>
<td>Return statement</td>
<td>non-redundant</td>
<td>return array;</td>
<td>invalid argument</td>
</tr>
</tbody>
</table>

Table 9: Constructs and features for Recursive problem.
Table 10: SOLO mapping for Recursive problem.

<table>
<thead>
<tr>
<th>Construct</th>
<th>Feature</th>
<th>Solutions by Student’s number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>49</td>
</tr>
<tr>
<td>Method declaration</td>
<td>Typical</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>Void method</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>Missing argument</td>
<td>x</td>
</tr>
<tr>
<td>Variable assignment</td>
<td>Efficient</td>
<td></td>
</tr>
<tr>
<td>edges</td>
<td>Valid</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Invalid</td>
<td></td>
</tr>
<tr>
<td>Difference calculation</td>
<td>Efficient</td>
<td>x</td>
</tr>
<tr>
<td>recursive invocation</td>
<td>Valid argument</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>Invalid argument</td>
<td>x</td>
</tr>
<tr>
<td>Return statement</td>
<td>Non-redundant</td>
<td>x</td>
</tr>
<tr>
<td>SOLO mapping (1st researcher)</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>SOLO mapping (2nd researcher)</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>SOLO mapping (3rd researcher)</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>Final and agreed SOLO mapping</td>
<td>P</td>
<td>P</td>
</tr>
</tbody>
</table>

4. Discussion

Despite the effort applied to developing a SOLO taxonomy for code-writing questions, mapping students’ responses based on a specific SOLO taxonomy has a degree of ambiguity and inconsistency. SPEs had therefore been introduced by Whalley et al. (2011) to minimise the mapping ambiguity and inconsistency. In addition, limitations of code-writing questions affect the mapping of students’ responses as certain types of question do not allow for high order thinking to be manifested in the students’ code. For example, if the question tends to measure student knowledge on how to declare a variable and assign a value to the variable, the student makes direct translations of what is required. Clearly, the student’s code cannot be categorised EA as the question is limited to specific requirements.

We find that identifying program constructs and extracting the quality features allow more constant mapping provided by all researchers, and we held a consolidation meeting to refine extracting program constructs and features for question three. In addition, we evaluated the importance and the weight of certain constructs that might have affected the mapping process. For example, there was a concern raised by one researcher questioning method declaration using the modifier static, and thus the mapping had to be slightly changed. For instance, in Table 11, student 36 manifested three main constructs (edges, difference calculation and recursion invocation) and had a slight error while checking the edges, and the student’s response categorised Relational the same as responses that manifested all three constructs. The student’s response that was categorised Relational should manifest all main constructs and features showing understanding of the relationship between them (Whalley, 2011).

Another challenge was the choice of the questions as we had been limited to only three code-writing questions that had been included in the exam script. Limitations of questions prevented students’ ability to be manifested and categorised in a higher category. The three questions had been categorised Multistructural, Multistructural and Relational respectively, thus those categories represented the highest categories for each question. In addition, we agreed to consider SOLO categories for each question when mapping students’ responses, so if a student’s response had been categorised higher than the question level, the category should be degraded. We agreed to categorise the questions based on the level of translations and concepts needed to be measured. Therefore, mapping students’ responses for code-writing questions should be accorded to the level of translations of required specifications in the code-writing questions (Whalley, 2011).

Our aim was to investigate high-performing students’ responses according to the SOLO taxonomy. Despite the limitations and challenges addressed earlier, results show that high-performing student manifest the ability to understand code-writing problems and provide solutions that might be categorized at the highest possible SOLO category. In question one (Array Creation), six students’ responses fell into the highest possible category whereas the rest of students’ responses were categorised in the second highest category (Fig. 3). In question two (Linear Search), eight students’ responses resided in the Multistructural category, which is the highest category for question two, while one
responses were categorised as Unistructural as shown in Fig. 4. As we mentioned earlier, question three focused on recursion which is one of the most difficult concepts for novice programmers. Therefore, question three was categorised as Relational where students’ responses might manifest a degree of abstraction that might vary from one student to another in which responses could be categorised at different levels. Fig. 5 shows that two students provided solutions categorized at the highest possible level where five students’ solutions manifested direct translations with invalid solutions categorized as Unistructural. Two students showed a lack of understanding program constructs in their solutions which had been categorised as Prestructural.

5. Conclusion
Educational taxonomies provide a framework for CSE to categorise students’ cognitive abilities in the computer science field. Several attempts have been made to apply Bloom’s taxonomy to categorise student code, but have resulted in a great deal of ambiguity as Bloom does not provide descriptions that can be interpreted easily in computer science. However, the SOLO taxonomy has been applied to classify students’ codes and algorithm designs. In this paper, we have adapted Whalley et al.’s (2011) framework, which has allowed us to code students’ responses for code-writing questions and to develop SPE and quality features which have assisted us to categorise students’ responses. Including the first author, two researchers have replicated the analysis procedures to ensure that analysis has yielded consistent results. The number of high-achieving students’ responses were categorised at the highest possible level for two of the three questions which were analysed, although only two students’ responses were categorised at the second highest and highest levels for the remaining question (which focused on the complex concept of recursion).
References


Abstract
This paper presents a prototype virtual reality debugging environment, aimed at novices, that provides a 3D visualisation of code and supports gestural navigation through the visualisation. This prototype is implemented for the HTC Vive, and has the capability to expand into a variety of 3D visualisations, programming languages, and VR platforms. We describe how our design builds on previous research on debugging to provide support for observing, exploring and hypothesizing activities by focusing on flow-of-control and data visualisations. We present preliminary results from pilot user testing and highlight key areas for future development.

1. Introduction
Debugging is an inevitable and challenging aspect of programming, and there is evidence that visualisation tools can provide useful support for users in the task, particularly for novice programmers. With the recent growth in virtual and augmented reality environments, there is potential for development of 3D representations that allow programmers to ‘walk through’ their code as part of the debugging process. This work explores the potential for alternative methods of displaying data and control flow in a virtual environment, with a view to identifying fruitful future directions for virtual reality debugging support. We present a prototype system, H.I.D.E (Humanised Interactive Development Environment), and some preliminary user testing findings.

The following section gives a brief background on how this work builds on existing research on debugging support. Following this, an overview of H.I.D.E is given, highlighting the key aims and functionality of the system. Finally, we explain our initial findings from pilot testing and their relevance before concluding, and pointing to key areas for future work.

2. Background
Debugging is an arduous task that all programmers at any level must undertake at some point, and a significant quantity of time allocated to it on any project. The motivation behind our system was to explore how a virtual reality environment could provide support to increase debugging efficiency, by building on existing understandings of common behaviours exhibited by programmers. Empirical studies have produced several relevant findings on debugging behaviour. As highlighted by Ko and Myers [1], there is a broad consensus that debugging is an exploratory task that can be broken down into six distinct and interleaving activities: hypothesizing what went wrong; observing runtime data; restructuring data; exploring the restructured data; diagnosing code; repairing code.

Ko and Myers [2] found that around half of the errors novice programmers made with the novice programming environment Alice were due to false assumptions in hypotheses made while debugging, noting that “85% of questions were about a single object. The remaining concerned multiple objects’ interactions” [1, p. 153]. This set the foundation for our thinking. If programmers are producing incorrect assumptions, and are primarily focussing on single objects or multi-object interactions, we asked how data could be displayed in a fashion to better facilitate their understanding.

Romero and colleagues noted in [3] that “experienced programmers, when comprehending code, are able to develop a mental representation that comprises different perspectives … as well as rich mappings between them”. However, this mental representation places a cognitive load upon programmers, increasing the quantity of data they need to process. Ideally, a system should form these mappings in a fashion that can be easily read without inducing additional cognitive load. Such a system would be
especially useful for novices, as they do not have the experience to form these mappings. Romero et al.
highlight the ‘double challenge’ faced by novices:

“... As well as trying to learn abstract concepts about programming, they have to master the
decoding, representation coordination and step-and-trace skills required to use debugging
environments.” [3, p.993].

Traditionally, debugging environments are deployed onto standard hardware. These environments use
the standard interfaces; keyboards, mice, and monitors. However, there are many rapidly evolving
technologies that present alternative methods of interfacing with computers. Virtual reality and
Augmented reality devices (Such as the HTC Vive and Microsoft Hololens) are but a few examples.
These devices present interaction possibilities not present in traditional input devices, such as haptic
control, true 3D environments, and depth perception. Exploring how these new possibilities could be
used to support novice debugging is the foundation of our system.

3. H.I.D.E Overview

H.I.D.E (Humanised Interactive Development Environment) is a system developed in Unity, deployed
to the HTC Vive that visualises LUA scripts in a 3D virtual environment. H.I.D.E focusses on providing
support for the observing, exploring and hypothesizing stages of debugging. These activities are the
ones most likely to benefit from visualizations, and there is potential to reduce the cognitive load of programmers by
making mappings visible.

H.I.D.E represents each lexical token within a program’s
lexical token tree as a 3D object within the space. When
looking directly at, or highlighting a token with an input
device, the system will show the data value of that token at
that point in the program execution. For example, when
looking directly at the token representing ‘x’ on the first
line, the value shown in the display will be ‘7’. However, when observing ‘x’ on the third line, the
value shown in the display will be ‘12’. Similarly, when observing the ‘if’ statement, if the
condition evaluates to true, the system will display ‘Is Entered’ to alert the user that this
statement’s body will be executed.

Figure 2 shows a view from an early
build of the HIDE system, displaying
an equality test. The user in this
instance is looking directly at the
equality symbol in the middle, while
highlighting the two data values on
either side. As can be seen, this
equality is testing if ‘n == 0’. The
left-hand sphere represents ‘n’ in this
instance, and its current value at this
point in the program execution is ‘1’.
As such, the display shows false.

Below is an example use case of H.I.D.E:

“A programmer is struggling to discover why her factorial function is returning incorrect values.
However, while looking at the code alone she cannot discover the error. She loads the program into
HIDE, and follows the manipulation of the input variable through in HIDE, using it’s highlighting
and context-sensitive readouts. She discovers that 5! is returning 121 instead of 120. After following
the execution through visually, she quickly spots that the base case is returning 2 instead of 1”

Though the example is simplified to an extreme, it demonstrates the potential benefit of the tool. HIDE
is designed to help programmers spot logical errors or human errors, that are difficult to see when
looking at traditional readouts. Program code can be navigated using the touch pads on the Vive controller, and code objects can be highlighted by pointing a controller at them and holding the trigger button. The system casts rays from the headset and controllers to detect which code object is currently being held in focus, and displays the appropriate readout.

4. Evaluation

We conducted a pilot user test, aimed at exploring how users interacted with the system, and their ability to understand the data displayed. The testing session was designed to assess how quickly participants comprehended what a small script was doing in two different environments – notepad++ and our system.

4.1 Method

Two undergraduate Computer Science students took part in testing. They each had 2-3 years programming experience, but neither had programmed in the Lua language before. Participants were given a brief orientation period, and then asked to debug two different scripts in the two different environments, with the ordering reversed. The time allowed for attempting to debug the short script in each system was 5 minutes. We asked participants to narrate their thought process throughout the experiment, and collected their informal observations about the system by audio recording the session. At the end of the 5 participants were asked to explain what they believed the program did, if they had not tried during the 5-minute period.

4.2 Results

Table 1 shows the time it took each participant to debug the scripts in each environment, whilst Table 2 shows participant feedback comments (transcribed from audio files).

Table 1 – Preliminary test results

<table>
<thead>
<tr>
<th>Participant</th>
<th>file1</th>
<th>Notepad++ time</th>
<th>file 2</th>
<th>H.I.D.E time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>test1.lua</td>
<td>4m 33s</td>
<td>test2.lua</td>
<td>Unsuccessful</td>
</tr>
<tr>
<td>2</td>
<td>test2.lua</td>
<td>Unsuccessful</td>
<td>test1.lua</td>
<td>End of test</td>
</tr>
</tbody>
</table>

Table 2 – Participant comments

<table>
<thead>
<tr>
<th>Participant 1</th>
<th>Participant 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Within a short amount of time, it’s easy to understand what is happening</td>
<td>Some parts made sense, but there is a lot going on</td>
</tr>
<tr>
<td>Extra-dimensional indentation is useful</td>
<td>Layout is too spread out</td>
</tr>
<tr>
<td>Large code bases may cause it to fall apart in terms of readability</td>
<td>Having to look at objects instead of data being explicitly displayed is frustrating</td>
</tr>
<tr>
<td>Could be useful in a teaching environment, or a small tech company</td>
<td>It could be difficult to understand</td>
</tr>
<tr>
<td>It is another way of sharing or debugging code</td>
<td>Unsure of where to use the system, but could be useful in smaller companies</td>
</tr>
<tr>
<td>Is more intuitive than some ways of displaying code</td>
<td>More of a secondary approach to traditional debugging</td>
</tr>
<tr>
<td>Difficult to judge what detail is needed</td>
<td>System is an alternative method of viewing code, from a new perspective</td>
</tr>
<tr>
<td>Main issues currently are colour and layout oriented. The layout is well presented, but rough around the edges</td>
<td>Vive is a good hardware platform to deploy the system to</td>
</tr>
<tr>
<td>It’s good fun, and would be a good teaching aide</td>
<td>Colour and transparency was not clear</td>
</tr>
<tr>
<td>Good for exploring a code base</td>
<td></td>
</tr>
</tbody>
</table>

131
4.3 Discussion
Neither participant successfully found the bug using the HIDE environment. Future versions of this study should allow more time for familiarisation with the system and the debugging task to allow further investigation of whether the system can support bug identification. However, participant comments gave some useful initial feedback. Both subjects described the system as intuitive, especially the dimensionality, which bodes well for developing the idea further. The system naturally leans towards collaborative efforts, as the visualisation could be seen by multiple parties simultaneously which would aide in mutual understanding of information. Colour, however, was highlighted as counter-intuitive, as such it will need further consideration.

The broader implications of this system for virtual reality are apparent. Users found having to look at an object to bring up information frustrating, and that the data was too spread out. This seems to stem from the low resolution of the headset, paired with a narrow field of view. The data can seem overwhelming with such a narrow perspective. In future systems, data must be very carefully filtered and abstracted to prevent cognitive overloading.

5. Conclusion
Our system creates a 3D environment within which users can view program code, to aide in their understanding of program structure. The visualisation aides in the formations of hypothesis regarding program errors, and assists users in comprehending the flow of data and control through a program. The current system presented requires additional refinement at the implementation level.

The system utilises a virtual reality headset to provide a novel approach to viewing program code, and presents opportunities for teaching inexperienced programmers the foundations of programming in a more interactive and engaging way.

The current system is limited to LUA, and does not fully support cross-file dependencies. However, the system has the potential to grow into a multitude of areas. Areas that require data input include; Running a script of function with a set of inputs, taken from natural language input, Modification of program code inside the environment, and integrated automated testing (such as Monte-Carlo or Unit testing). Areas that do not require input that could be expanded upon are; Multiple users in the same environment, Cross-platform compatibility (Such as the Occulus rift and Hololens) and Multi-language support for both programming and natural languages.

The system has identified a promising field for further study regarding alternative methods of displaying data. The inherent ease of use when dealing with multi-sensory environments has great potential for increasing efficiency, as well as communicating data between entities faster.

6. References
Abstract
Programmers acknowledge the difficulty of naming things, whatever their experience level and wherever they work, but relatively few use explicit naming guidelines. Various authors have published different kinds of identifier naming guidelines, but these guidelines do little to make naming easier, in practice. Meanwhile, professional programmers follow diverse conventions and disagree about key aspects of naming, such as acceptable name lengths.

Although few teams write their own coding standards, let alone naming guidelines, many teams use code review and pair programming to maintain code quality. We believe that these teams could use third-party naming guidelines to inform these reviews, and improve their coding style.

This paper examines various sources of naming guidelines, in the context of the first author’s twenty years’ experience as a professional programmer. This paper then presents a consolidated set of naming guidelines that professional programmers can apply to the code they write.

1. Why naming matters
Several researchers have explored the importance of naming. For example, Deissenbock and Pizka conclude that identifier names are crucial to program comprehension:

“Research on the cognitive processes of language and text understanding show that it is the semantics inherent to words that determine the comprehension process” [20].

Other authors agree; Caprile and Tonella say that identifiers provide important information about program entities, because they give the programmer an initial idea of these entities’ roles within the whole program. Deissenbock and Pizka [20] not only present their opinion on naming, they also performed measurements. They found that in the Eclipse code base, which consists of about 2 MLoC, 33 per cent of the tokens and 72 per cent of characters are devoted to identifiers.

Better identifier names have been known to correlate with program comprehension. For example, [23] reports on a study performed with over 100 programmers, who had to describe functions and rate their confidence in doing so. Their results show that using full word identifiers leads to better code comprehension than using single-letter identifiers, measured by both description rating and by confidence in understanding. However, they also found that in many cases there is no difference between words and abbreviations. Interestingly, this study also found that women comprehend more from abbreviations than men do.

Naming might have been found to matter for source code quality. Butler *et al.* evaluated the quality of identifiers in eight large Java projects according to a number of naming style guidelines. They found that the occurrence of naming violations correlated with code issues as reported by FindBugs, a static analysis tool for Java [22]. In particular, capitalisation errors, using non-dictionary words and using more than four words were correlated with issues.

Developers agree that naming matters. In an ethnographic study among twelve professional developers and eighteen third-year students [24], researchers found that both students and professional developers find the use of naming guidelines important. The study also found a remarkable difference between professionals and students: professional developers pay more attention to the name of the identifiers...
than to source code comments. Could this be due to the fact that computer science courses tend to emphasise the importance of comments but largely neglect naming?

While developers agree that guidelines are important, we have observed that software development typically turns out to cost more and take longer than anyone expects.

As Bugayenko writes [9], software development is "a never-ending process' that will cost 'All of your money, and it won’t be enough'. We see that the cost of continuous software development includes the cost of debugging, fixing and maintaining code. These activities clearly require programmers to read and understand existing code. As programmers, we can only understand code if we know what it means.

A thought experiment further illustrates that we rely on naming to understand what code means. Imagine trying to read code after someone has replaced every identifier name with an underscore followed by a random number. Although an identifier like 'result' communicates relatively little intent, a name like 42 explains even less.

2. Purpose of naming guidelines

In the above, we have established that naming is important, but also hard. As Karlton famously said: 'There are only two hard things in Computer Science: cache invalidation and naming things.' We think some programmers make the mistake of focusing too much on the executability of the code, rather than on the value of the code as a thing for humans to read, forgetting that other famous quote by Knuth: Programs are meant to be read by humans and only incidentally for computers to execute." A good name helps a future reader of code to quickly understand what a value means, thus making code more readable and easier to understand. However, programmers don’t always try write code to be maintainable, and when they do they typically find it difficult to achieve. The very idea of 'maintenance' lacks a common industry definition that doesn’t assume a specific (usually waterfall) software development method or software services business model. Similarly, computer science does not have a clear definition of 'maintainability', and instead focuses on proxies such as code comprehension and programmers’ ability to discover and correct code errors. These related measures reduce 'maintainability' to 'readability'. Readability requires good naming, because bad names obscure the programmer’s intent. We claim, above, that naming affects programmers’ ability to read and understand source code. Unfortunately, programmers struggle to write readable code because they struggle to avoid using bad names. Naming guidelines aim to help programmers identify and avoid bad names, and to guide programmers towards good names. We see naming guidelines as a means to help programmers write more maintainable code, and to reduce the cost and difficulty of software development. Crucially, these benefits potentially apply to all software development, not just long-term maintenance of legacy systems.

3. Existing guidelines

In our experience, professional software developers don’t use explicit naming guidelines extensively. The few written coding standards in common use, such as [6], limit their guidelines to formatting and name length, but offer little to clarify the difference between good names and bad names. Some books for software developers include a section on naming. Code Complete [4] includes a 30-page chapter on The Power of Data Names. This includes fourteen guidelines for how to write better names, a discussion of various naming conventions, a list of eleven naming smells and a checklist that summarises these guidelines. For this chapter alone, we recommend that every professional programmer own a copy of this book. Clean Code [5] also has a whole chapter on Meaningful Names, which consists of eighteen guidelines. Most of these guidelines directly address the hardest part of naming: semantics. More recent programming books tend to devote fewer words to naming, perhaps because they have little to add. Computer science research sometimes includes naming guidelines. Papers by Relf [2] and Arnaodova et al. [3] include collections of naming guidelines, which they evaluate in different ways. A thorough literature review would no doubt turn up more naming guidelines, but professional programmers rarely have access to published papers, which makes them less directly useful in the software industry than books.
4. Importance of different guidelines to professional programmers

Professional software developers benefit more from some kinds of guidelines than from others. Guidelines like "Variable names should be short yet meaningful" [6] sound reasonable, but offer little practical help, either when choosing a name when coding or when evaluating a name during code review. Some academic studies, such as Binkley [10], have compared the relative readability of different formatting conventions, such as camel-case (capitalised words) and snake-case (words separated by underscores). In principle, programming language designers could use this research when setting these conventions to design programming languages with a more productive developer experience. Ken Arnold typifies the view that the responsibility for using this kind of research to choose a coding style lies with language designers rather than programmers. In his essay Style is substance, he argues that a programming language’s specification should fix all aspects of coding style, so that compilers reject all violations: “For almost any mature language? coding style is an essentially solved problem, and we ought to stop worrying about it. ? the only way to get from where we are to a place where we stop worrying about style is to enforce it as part of the language. I want the owners of language standards to take this up. I want the next version of these languages to require any code that uses new features to conform to some style.” [11]

He argues that programmers follow the name formatting convention that a programming language community adopts, and that they have nothing to gain from this kind of research.

“For any given language, there are one or a few common coding styles.... There is not now, nor will there ever be, a programming style whose benefit is significantly greater than any of the common styles.”

However, Binkley [10] concludes that not all ‘common coding styles’ deliver the same productivity, and that “it becomes evident that the camel case style leads to better all around performance once a subject is trained on this style”.

Fortunately, some research has directly addressed different guidelines’ usefulness. Relf, for example, concludes that:

> The identifier-naming style guidelines that proved to be the most useful to programmers required that > identifier names should be composed of from two to four Natural language words or project accepted acronyms; > should not be composed only of abstract words; > should not contain plural words; > and should conform to the project naming conventions. [8]

Professional programmers can apply guidelines that are stated this clearly more readily than they can access and read the original scientific research that contains these conclusions. We therefore aim to present naming guidelines from a number of sources in a form that makes them accessible to professional programmers.

5. Guideline styles

People who write naming guidelines phrase their guidelines in different ways. Some authors write prescriptive instructions, e.g. Use intention-revealing names [5], while some phrase them as code smells or naming problems, e.g. Meaningless names [1]. The written naming guidelines [1,2,3,4,5,6,7] that we examined include one or more of the following.

- Prescriptive instruction
- Naming smell name
- Correcting refactoring name
- Example guideline violation
- Example name that follows the guideline
- Explanation of why the guideline matters or how it works
Naming smells are "code smells" that come from bad names. A code smell indicates where you can improve your code, and often points to some deeper problem. A particular code smell often has a corresponding refactoring that removes that particular smell, improving the code. Naming smells appear in many forms, but all have the same refactoring: *Rename*.

Needless to say, programmers find consistently-written guidelines easier to understand and apply. As well as consistency, multiple explanations help programmers apply a guideline in different scenarios. Naming smells help programmers identify violations during code review, while prescriptive instructions are easier to follow while writing code. Examples serve to explain both smells and instructions, whose abstract nature can make them hard to understand.

The remainder of this paper presents and discusses specific guidelines.

### 6. Syntax guidelines

Syntax guidelines address how identifiers are constructed from words and formatted. These guidelines are not concerned with which words names use, except for the guideline to use words in the first place.

#### 6.1. Use naming conventions

**Guideline.** Follow the programming language’s conventions for names. Programming languages usually have some conventions for how to write identifier names, or at least their specifications or communities do. Java programmers, for example, follow Sun Microsystems’ original guidelines [6] for how to use upper and lower-case, nouns and verbs, in the names of classes, interfaces, methods, variables and constants [2], [6].

**Refactoring.** Apply standard case with rigorous consistency, and use language-specific code inspection tools to enforce it.

**Example violations.** ‘appleCOUNT’, ‘applecount’ (when camel-case is standard).

#### 6.2. Replace numeric suffixes

**Guideline.** Don’t add numbers to multiple identifiers with the same base name. If you already have an ‘employee’ variable, then a name like ‘employee2’ has as little meaning as ‘anotheremployee’[1], [2], [4].

**Refactoring.** Replace the numbers with additional words that describe the difference between multiple identifiers that might otherwise have the same name.

**Example violations.** ‘employee2’

#### 6.3. Use dictionary words

**Guideline.** Only use correctly-spelled dictionary words and abbreviations. Make exceptions for ‘id’ and documented domain-specific language/abbreviations. Spelling mistakes can render names ambiguous, and result in confusing inconsistency. Abbreviations introduce a different kind of ambiguity that the original programmer does not see because they know which word the abbreviation stands for, even if multiple words have that same abbreviation.[1], [4], [5].

**Refactoring.** Spell words out in full and define abbreviations for the bounded context. Use tools that identify spelling errors in identifier names.


#### 6.4. Expand single-letter names

**Guideline.** Don’t make exceptions to using dictionary words for single-letter names; use searchable names. Single-letter names, when used as abbreviations, introduce the maximum possible ambiguity. They end up being used with specific meanings, usually by unwritten convention, which makes the code harder to read for programmers when they first encounter the convention or who have to switch between conflicting conventions in different contexts[1], [2], [4], [5]. One study of over 100 programmers that compared comprehension for single letters, ‘well-formed’ common abbreviations and full words supports this guideline: “The results show that full-word identifiers lead to the best comprehension; however, in many cases, there is no statistical difference between using full words and abbreviations.” [12]
Refactoring. [Use dictionary words](use-dictionary-words).


6.5. Articulate symbolic names

Guideline. Don’t use ASCII art symbols instead of words, in programming languages that support it. Make very limited exceptions for documented domain-specific symbols, e.g. ‘+’ in arithmetic. Ironically, programmers who encounter symbolic names in third-party libraries may invent their own names, but choose names based on what the symbol looks like, rather than what it means.[1]

Refactoring. [Use dictionary words](use-dictionary-words).

Example violations. ‘=>’, ‘<*>’ - valid function identifiers in Scala, for example, colloquially named fish and space ship.

6.6. Name constant values

Guideline. Name what the constant represents, rather than its constant value. Don’t construct numeric constant names from numbers’ names.

Refactoring. Extract constant, for the Magic number code smell. Replace number names with either domain-specific names, such as ‘pi’, or a name that describes the concept that the number represents, such as ‘boilingpoint’[2], [4].

Example violations. ‘radius * 3.142591’, ‘ONEHUNDRED’

6.7. Only use one underscore at a time

Guideline. Don’t use more than one consecutive underscore. Multiple underscores usually appear as a single line, which makes it hard to count them.[2]

Refactoring. Replace with a single underscore.

Use tools that warn when names contain multiple underscores. Example violations. ‘APPLECOUNT’

6.8. Only use underscores between words

Guideline. Don’t use underscores as prefixes or suffixes. Underscores lack visual prominence, which makes them good word separators, but easy to misread before or after a word.[2]

Refactoring. Trim underscores. Use tools that warn when names do not start with a letter.

Example violations. ‘APPLECOUNT’

6.9. Limit name character length

Guideline. Keep name length within a twenty character maximum. [2] The results of one experiment involving 158 ‘programmers of varying degrees of experience’: “reinforce past proposals advocating the use of limited, consistent, and regular vocabulary in identifier names. In particular, good naming limits individual name length and reduces the need for specialized vocabulary.[13]”

Refactoring. Simplify name, Extract variable.

Example violations. ‘ForeignDomesticAppleCount’

6.10. Limit name word count

Guideline. Keep name length within a four word maximum, and avoid gratuitous context. Limit names to the number of words that people can read at a glance. Don’t unnecessarily use the same prefix, such as the software system’s name, for all names[2], [4], [5], [8], [22].

Refactoring. Simplify name, Extract variable.

Example violations. ‘NewRedAppleSizeType’, ‘MyAppSizeType’

6.11. Qualify values with suffixes

Guideline. Use a suffix to describe what kind of value constant and variable values represent. Suffixes such as ‘minimum’, ‘count’ and ‘average’ relate a collection of values to a single derived value. Using a suffix, rather than a prefix, for the qualifier naturally links the name to other similar names[2], [4].

Refactoring. Move the qualification to the end.

Example violations. ‘MINIMUMAPPLECOUNT’ (replace with ‘APPLECOUNTMINIMUM’).
6.12. Make names unique

**Guideline.** Don’t overwrite (shadow) a name with a duplicate name in the same scope. In Java, for example, a local variable hides a class field that has the same name. Adopt a convention that prevents ambiguity in which name the programmer intended to refer to[2].

**Refactoring.** Add words to one of the names clarify the difference between contexts.

7. Vocabulary guidelines

Vocabulary guidelines address word choice, with the rationale that using the right word matters.

7.1. Describe meaning

**Guideline.** Use a descriptive name whose meaning describes a recognisable concept, with enough context. Avoid placeholder names that deliberately mean nothing more than ‘avariable’[1], [4], [5].

**Refactoring.** Describe what the identifier represents.

**Example violations.** ‘foo’, ‘blah’, ‘flag’, ‘temp’

7.2. Be precise

**Guideline.** Identify a specific kind of information and its purpose. Imprecise words might apply equally to multiple identifiers, and therefore fail to distinguish them[1].

**Refactoring.** Replace vague words with more specific words that would only be correct for this name.

**Example violations.** ‘data’, ‘object’

7.3. Choose concrete words

**Guideline.** Use words that have a single clear meaning. Like imprecise words, abstract words might apply equally to multiple identifiers[1], [2].

**Refactoring.** Replace with more specific words that narrow down the concept they refer to.

**Example violations.** ‘Manager’ suffix, ‘get’ prefix, ‘doIt’

7.4. Use standard language

**Guideline.** Avoid being cute or funny when it results in a name that requires shared culture or more effort to understand. Like deliberately meaningless names, cute and funny names require the reader to understand some implicit context. While humour often relies on indirect references and ambiguity, these qualities do not improve code readability [5].

**Refactoring.** Replace indirect references and colloquial language with the corresponding explicit and standard language.

**Example violations.** ‘whack’ instead of kill.

7.5. Use a large vocabulary

**Guideline.** Use a richer single word instead of multiple words that describe a well-known concept. Use the word that most accurately refers to the concept the identifier refers to[1].

**Refactoring.** Replace multiple words that describe a concept when ‘there’s a word for that’.

**Example violations.** ‘CompanyPerson’ (replace with ‘Employee’).

7.6. Use problem domain terms

**Guideline.** Use the correct term in the problem domain’s ubiquitous language, and only one term for each concept. Consistently use the correct domain language terms that subject-matter experts use[1], [4], [5].

**Refactoring.** Rename identifiers to use the correct terminology.

**Example violations.** ‘Order’ when you mean ‘Shipment’, in a supply-chain context, where it means something different.

7.7. Make names differ by more than one or two letters

**Guideline.** Don’t use a name that barely differs from an existing name. Avoid words that you will probably mix up when reading the code[2], [4], [5].

**Refactoring.** Make the difference more explicit by adding or changing words.

**Example violations.** ‘appleCount’ vs ‘appleCounts’
7.8. Make names differ by more than word order

**Guideline.** Don’t use a name that only differs from an existing name in word order. Don’t use two names that both combine the same set of words.  

**Refactoring.** Make the difference more explicit by using different words rather than just different word order to communicate different meanings.

**Example violations.** ‘appleCount’ vs ‘countApple’

7.9. Make names differ in meaning

**Guideline.** Don’t use names that have the same meaning as each other. Avoid names that only differ by changing words for their synonyms.

**Refactoring.** Rename both variables with more explicit names.

**Example violations.** ‘input’/‘inputValue’, ‘recordCount’/‘numberOfRecords’

7.10. Make names differ phonetically

**Guideline.** Don’t use names that sound the same when spoken. Aim to write code that another programmer could write down correctly if you read it out loud. Even though don’t transcribe code like that, as a rule, they often talk about code.

**Refactoring.** Replace a homophone with a synonym.

**Example violations.** ‘wrap’/‘rap’

8. Data type guidelines

Data type guidelines extend vocabulary guidelines by addressing data type names in identifier names. Some of these guidelines only apply to languages whose type system allows code to explicitly identify data types, separately from identifier names. Code in other languages cannot always avoid the need to indicate types.

8.1. Omit type information

**Guideline.** Don’t use prefixes or suffixes that encode the data type. Avoid Hungarian notation and its remnants. Don’t prefix Boolean typed values and functions with ‘is’.

**Refactoring.** Remove words that duplicate the data type, either literally or indirectly.

**Example violations.** ‘isValid’, ‘dateCreated’, ‘iAppleCount’

References:

8.2. Use singular names for values

**Guideline.** Don’t pluralise names for single values.

**Refactoring.** Replace the plural with the singular form.

**Example violations.** ‘appleCounts’

8.3. Use plural names for collections

**Guideline.** Pluralise names for collection values, such as lists. Technically, this contradicts the guideline to avoid encoding type information in names, but English grammar requires it to make it possible to read the code normally, or out loud.

**Refactoring.** Use the plural form.

**Example violations.** ‘remainingApple’ for a set of apples.

8.4. Prefer collective nouns for collections

**Guideline.** If a collection’s type has a collective noun, in the name’s context, use it instead of a plural.

**Refactoring.** Use the collective noun, when possible, instead of a regular plural form.

**Example violations.** ‘appointments’ (replace with ‘calendar’), ‘pickedApples’ (replace with ‘harvest’).

8.5. Use opposites precisely

**Guideline.** Consistently use opposites in standard pairs with naming conventions. Typical pairs include add/remove, begin/end, create/destroy, destination/source, first/last, get/release, increment/decrement,
insert/delete, lock/unlock, minimum/maximum, next/previous, old/new, old/new, open/close, put/get, show/hide, source/destination, start/stop, target/source, and up/down[4].

**Refactoring.** Use the correct opposite, and use it consistently.

**Example violations.** ‘first’/’end’

8.6. Use Boolean variable names that imply true or false

**Guideline.** Use names like ‘done’ or ‘found’ that describe Boolean values. Use conventional Boolean names, possibly from a code conventions list [4].

**Refactoring.** Replace Boolean names with names in the correct grammatical form.

**Example violations.** ‘status’ for e.g. ‘started’

8.7. Use positive Boolean names

**Guideline.** Don’t use negation in Boolean names. Don’t use names that require a prefix like ‘not’ that inverts the variable’s truth value [4].

**Refactoring.** Invert the meaning and remove the prefix.

**Example violations.** ‘NotSuccessful’

9. **Class name guidelines**

Class name guidelines specifically address names for classes in object-oriented programming languages.

9.1. Use a noun-phrase name

**Guideline.** Name a class with a noun phrase so you can use the class name to complete the phrase This class’ constructor returns a new?. Follow object-oriented programming’s grammatical conventions[5,6].

**Refactoring.** Add the missing noun, remembering to [Choose concrete words](choose-concrete-words).

**Example violations.** ‘Calculate’

9.2. Use a name that allows all possible states

**Guideline.** Don’t use class names that assume a particular state. If a class models something that can have multiple states, then avoid a name that would be inconsistent with the state that results from calling a method that changes that state [3].

**Refactoring.** Make the class name less specific to accommodate all possible states.

**Example violations.** ‘disable’ method that returns a ‘ControlEnableState’ (rename class to ‘ControlState’).

9.3. Choose a name consistent with possible values

**Guideline.** Don’t use a name that appears to contradict certain possible values. Some types aggregate multiple values of the same type, such as a line that has a ‘start’ and an ‘end’, so use a name that applies equally to both values, such as ‘Extremity’, rather than naming the type after just one possible value, such as ‘Start’[3].

**Refactoring.** Make class name inclusive.

**Example violations.** ‘start’ field has type ‘MAssociationEnd’ (rename class to ‘MAssociationExtremity’).

10. **Method name guidelines**

Method name guidelines specifically address names for methods in object-oriented programming languages. Several of these guidelines apply to Java in particular, due to the bad habits the JavaBeans Specification [6] encouraged.

10.1. Use a verb-phrase name

**Guideline.** Make the method name an active verb phrase, except for accessor methods and some conversions. As with the guideline to use noun phrases to name class, follow object-oriented programming’s grammatical conventions. Some coding styles omit the verb from accessor methods, changing ‘Parcel.getWeight()’ to ‘Parcel.weight()’. Another common style is to omit the verb from conversion methods, changing ‘Discount.convertToPercentage()’ to ‘Discount.asPercentage()’[5,6].

**Refactoring.** Add the missing verb, remembering to [Choose concrete words](choose-concrete-words).
Example violations. ‘calculation()’

10.2. Don’t use ‘get’, ‘is’ or ‘has’ prefixes for methods with side-effects

Guideline. Use a verb phrase that suggests the side-effect, if there is one. Verbs like ‘create’ and ‘convert’ suggest a side-effect, while others suggest idempotence[3].

Refactoring. Replace ‘get’ with another verb.

Example violations. ‘getImageData’ method that constructs a new object.

10.3. Only use ‘get’, ‘is’ and ‘has’ prefixes for methods that only perform field access

Guideline. Only use the conventional accessor method name prefixes for accessor methods that directly return a field value. In Java, the JavaBeans specification [7] requires these prefixes for certain methods. When some methods require a certain prefix, don’t use the same prefixes for methods that do not require them.

Refactoring. Replace ‘get’ with another verb.

Example violations. ‘getMethodBodies’ populates the method bodies but doesn’t return them.

10.4. Only use ‘get’ prefix for field accessors that return a value

Guideline. Don’t use the ‘get’ field accessor method name prefix for methods that don’t return a value [3].

Refactoring. Replace ‘get’ with a verb that describes the side-effect.

Example violations. ‘getScore’ that performs calculation or accesses external data.

10.5. Only use ‘is’ and ‘has’ prefixes for Boolean field accessors

Guideline. Don’t use the conventional Boolean accessor method name prefixes for methods that don’t return a Boolean value[3].

Refactoring. Replace prefix with ‘get’ or remove the prefix altogether.

Example violations. ‘isValid’ returns an ‘int’ value.

10.6. Only use ‘set’ prefix for field accessors that don’t return a value

Guideline. Don’t use the ‘set’ field accessor method name prefix for methods that return a value[3].

Refactoring. Replace ‘set’ with another verb, or remove it in a ‘fluent API’ that chains method calls.

Example violations. ‘setBreadth’ creates and returns a new object, or updates and returns ‘this’ (fluent API).

10.7. Only use validation verbs for methods that provide the result

Guideline. Only use verbs like ‘validate’, ‘check’ or ‘ensure’ to name methods that either result or throw an exception when validation fails[3].

Refactoring. Return result. Example violations. ‘validateSnaps’ and ‘checkCurrentState’ that return ‘void’.

10.8. Only use transformation verbs for methods that return a transformed value

Guideline. Only use verbs that suggest transformation, like ‘convert’, for methods that return the result [3].

Refactoring. Return result, or change the verb to indicate what the method transforms.

Example violations. ‘javaToNative’ with return type ‘void’.

11. Further research

While developers agree guidelines are important [24], they remain underused in the software industry. In our experience, professional software developers do not always agree on which guidelines to use, or even that they are worthwhile. Our industry would benefit from more rigorous answers to the following questions.

1. Which naming guidelines apply universally to all kinds of code?
2. Which naming guidelines have the most positive impact on code readability and maintainability?
3. Can we usefully reduce naming guidelines to a short checklist for use in code review?
4. Can mining large collections generate ‘crowd-sourced’ standard vocabularies for specific domains?
5. How should software developers write naming guidelines?
6. How should software developers use naming guidelines?
7. Is it possible to measure identifier name quality or naming guideline effectiveness?
8. What can we learn from a cost-benefit analysis of naming guidelines?
9. How do naming and naming guidelines relate to software documentation?
10. Does better naming reduce the need for code comments?
11. Do pair programming and mob programming significantly improve naming quality?
12. How can an English dictionary and thesaurus help programmers choose effective names?
13. Which techniques have a positive effect on improving programmers’ naming skills?
14. Which tools to support naming have programmers not yet tested in an industrial setting?
15. How does the programmer’s native language affect their approach to choosing English identifier names?

Needless to say, we hope that software engineering researchers address these questions in the future.

12. References
1. Peter Hilton - [Naming smells](http://hilton.org.uk/blog/naming-smells) (2016)
12. Dawn Lawrie, Christopher Morrell, Henry Feild and David Binkley - Effective identifier names for comprehension and memory (2007)
13. Dave Binkley, Dawn Lawrie, Steve Maex and Christopher Morrell - Identifier length and limited programmer memory (2009)
User experiences in a visual analytics business

Mariana Mărășoiu
Computer Laboratory
Cambridge University
Mariana.Marasoiu@cl.cam.ac.uk

Alan F. Blackwell
Computer Laboratory
Cambridge University
Alan.Blackwell@cl.cam.ac.uk

Abstract
We report on an ongoing ethnographic study conducted at a data analytics company and discuss the multiple facets of user experience observed in this context. We describe in detail two interaction episodes of analysts working with visual analytics software and characterize them through the Patterns of User Experience framework. We discuss the implications of our observations and make some recommendations for future tool design.

1. Introduction
The way in which businesses use data is changing, with more and more companies relying on visual analytics for monitoring, improving or shaping their business. This can be done either in-house, or it can be contracted out to specialized visual analytics consultancies. Together with the increased interest in the larger population for analysing data, it is becoming ever more important to build tools that support this work.

Ethnography and ethnographically informed methods have been previously used in studies of software engineering in order to understand and describe the social context and work practices of engineers in a variety of settings, for example agile development (Sharp & Robinson, 2004) and professional end user development (Prior et al., 2008). At PPIG, ethnomethodologically informed ethnography has been used to discuss programmers reading code by Rooksby and colleagues (2006). They have also been used to study end user programmers (e.g. Nardi & Miller, 1990). Sharp and colleagues highlight the importance of ethnographic studies in the context of empirical software engineering research (Sharp et al., 2016), but their discussion can be extended to other studies of professionals that work with computers on a daily basis. One such group are data analysts.

Recent work studying data analysts includes an interview study of 35 data analysts from various industries (Kandel et al., 2012). The study characterized the process of data analysis in a real-world industrial context, by categorizing three types of analysts on the dimension of tool use, and classifying the activities that the analysts engage in with regards to data processing. We add to their work through an in-depth study of a small group of analysts that specialize in visual analytics for exploring, understanding and reporting data.

In this paper, we use the Patterns of User Experience framework (Blackwell, forthcoming; Blackwell & Fincher, 2010) to characterize ethnographic descriptions, highlighting the experiences we observed the data analysts having whilst they worked working with visualisation tools. We also observe the commonalities between the work of the analysts and previously studied programming-related activities and behaviours. We present these observations alongside ethnographic “thick descriptions” (Geertz, 1973), a format which allows us to discuss the experiences of the analysts in the context of their work.

2. Methodology
The study started in August 2016. The first author was the single observer, and visited the office of Atheon Analytics one day a week, almost every week, for a total of 27 times until the submission of this paper. The observer aimed to take part in day-to-day activities, by attending meetings, having informal discussions with the analysts, observing them at work by sitting next to them, and working at her own laptop at a nearby desk in order to capture the shared office environment.

In the chosen setup, due to the researcher not being trained as an analyst and the relatively sparse visits, there was limited opportunity for more in-depth participation in the work activities of the team, such as working on models and dashboard development alongside the analysts. However, the ability
to observe the team over several months offered insights into their work that a shorter but more dense study wouldn’t have provided. For example, we were able to observe a wide range of activities and projects, as well as how the projects evolved through various stages of requirements gathering, planning, design, implementation and customer support.

3. The background
To situate the observed episodes that are described in Section 4, we first describe the context of the ethnographic study, the setting and the people observed, and then give a short description of Tableau1, a commercial software for data visualisation used by the analysts.

3.1 The setting
The office is located in the Cranfield Innovation Centre, a one-storey office building within the Cranfield University Technology Park. The Park is located outside the Cranfield village, next to the Cranfield Airport, on the road that leads from the village to the university. The team first moved here in 2012 from Luton, and since then they have expanded to larger offices twice more, but within the same building. One such move happened during the study reported here, in December 2016.

Atheon initially started as two separate businesses, Atheon Consulting and Atheon Software Products, which were merged in 2010 under a single name, Atheon Analytics. However, the dual nature of the company has been maintained over the years, as Atheon Analytics offers both consulting services and products for retailers and suppliers. At present, the company has around 22 employees, 4 of which are formally part of the consulting team, alongside the managing director. The study reported here has been conducted by observing this team.

Figure 1. A photo of the office2 and a sketch of the office plan.

The office is open plan, with desks having partition screens on the long edge. The partition screens were added after the last move and they weren’t present in the office when the study started. Each desk of the team members has one or two monitors, but no desktop computers - all the work is done on laptops, with the external monitors connected for additional screen space. A photo and a sketch of the office can be seen in Fig. 1.

The office has a kitchen and a meeting room, both of which got larger when the company moved offices. Besides the usual appliances, the current kitchen has a whiteboard and a table, where people eat, chat or sometimes even have meetings (when the meeting room is occupied).

3.2 The consulting team
The consulting team is located in the far left corner, the 4 analysts working at a four-desk island (highlighted in Fig. 1). The managing director and CEO, who organizes most of the consulting projects and also contributes with analyses sits at an adjacent desk on the right side. The other technical teams are on the other side of the common play area.

1 https://www.tableau.com/
The team has one weekly meeting, usually scheduled for Monday morning, sometimes moved later in the week, where they go through the projects that they’re working on, update each other on progress, decide the state of the project for the upcoming week, and add new projects. Usually, there are between 5 and 8 projects that the team is actively working on, with another 5 to 8 in wait, depending on other people, and around 20 more possibilities of future projects.

The projects are diverse, ranging from one-off analytical pieces where the output is a presentation of the insights, to building bespoke “tools” for clients that will be used in business decision making, and to maintaining and adding new features to SKUtrak\(^3\), a Tableau-based product that the company provides as a service to a number of retailers and suppliers.

This wide range of projects is interesting for multiple reasons. First, the analyst takes on multiple roles. One is that of the typical data analyst where they’re asked to analyse some data and report on it. Another set of roles are those related to building a software product, so the analysts themselves are gathering requirements, designing an interface, develop it and then maintaining it. From our observations, the latter is more prevalent in the work of the team we studied.

3.3. The software used by the analysts

The main visual analytics tool used by the team is Tableau, a commercial software package that allows rapid creation of interactive visualisations and dashboards of visualisations. Tableau is based on previous work visualisation techniques for data cubes and relational databases (Stolte et al., 2002).

The result of a project is often a Tableau workbook. Tableau’s file organization is similar to that of Excel: a workbook is a file that contains one or more sheets, which can in turn be worksheets, dashboards, or stories. A worksheet contains a single “view” of the data, for example a table or a chart. A dashboard consists of one or more views laid out on a canvas area. Fig. 2 shows a screenshot of the interface for creating a visualisation worksheet, to aid understanding of the user interaction with the software we describe in the next section.

![Figure 2. A screenshot of the Tableau data “view” interface.](image)

Whilst most of the observations were conducted when the analysts were using Tableau, they are using a large number of other tools that help them for specific tasks. For example, Alteryx, a data flow programming language, is being used for data processing and data blending, Excel is being used for some analytical tasks and tracking requirements, PowerPoint is used for creating anticipatory sketches of a data visualisation or for presenting results, SQL for querying databases, and Python scripting within Jupyter notebooks for predictive analyses.

\(^3\) https://www.skutrak.com/
4. Findings

In this section, we discuss two observed instances of analysts working with Tableau and other tools. We offer rich descriptions of each observed episode, and, in parallel commentary, a) discuss them using the Patterns of User Experience framework in order to describe the experience of the analysts as they’re using their tools (green text), and b) highlight programming-related activities (blue text). The names in the descriptions are fictional.

Due to the context in which our analysts work, we can describe two types of experiences, depending on who the user is. First, we can analyse the designed customer-experience: the experience that the analysts aim to design in their tool for their clients. Second, we can observe and discuss the analyst-experience, that is, the experience of the analyst as they’re building the tool. This dichotomy of experiences appears since both the analysts and the clients use the same underlying software, with the analysts having the knowledge to “program” against it (to create complex visualisations and dashboards, program filters, actions and interactions, etc.), whereas the client is using the resulting dashboards in an interactive, but read-only mode (they’re interested in understanding today’s data with the given visualisations, not in creating new dashboards).

4.1 Preparing for a workshop with a client

John explains to me what he’s working on today: “I have a training session coming up later this week [so I’m now figuring out what they need]. [...] They also gave me a shopping list of stuff [that they want to do]” during the training session, so John is going through that list to make sure he is prepared for the meeting. I only notice this later, but he has a spreadsheet with the requirements from the client team, where he also takes notes of the things that should be brought up during the training session.

For now, he’s adjusting the width of some Filter widgets in a dashboard. He does this by following the same set of steps for each of the filter: from the menu dropdown of the filter widget he selects “Fixed width”, then enters 180 as the new pixel width of the widget.

The spreadsheet of requirements fulfills an equivalent role to that of a feature tracker in professional software engineering.

Following this, he renames the sheet by adding “- Working” at the end, then duplicates it and starts to change the previous one more substantially.

This activity could be described within the PUX framework as a modification activity - the widgets already exist, and the analyst is only changing their appearance. With regard to the designed customer-experience, the goal of this activity is to improve the experience of meaning, and in particular, ME3: Similar things look similar. From the analyst-experience perspective, the interface enables to some extent IE2: Actions are fluid, not awkward and PE2: The steps you take match your goal - the sequence of steps is easy to remember, but it is not supporting PE5: Repetition can be automated.

We also notice a strategy for version control: the analyst creates a checkpoint by duplicating a sheet, renaming it with a description of it’s current state, in order to make more substantial changes to the initial sheet.
Starting with a dashboard that has a table on the left side and a bar chart on the left, he goes to the sheet describing the bar chart. He duplicates a pill (Tableau’s name for field names which comes from their visual representation, e.g. ) that already exists in the column shelf, defining the bars of the chart. This creates a new visualisation side by side with the existing visualisation. He then makes the marks of this new visualisation fully transparent, and then merges the two visualisations into a single, dual-axis chart. He then hides the axis and title at the bottom (which corresponds to one of the charts) and he also hides the axis at the top, but maintains its title (this corresponds to the other chart). He explains that he did this so that the chart design could match the table in the dashboard where they are put side by side. John mentions that “in Tableau 10 you can have a grand total in a table and you can chose to have that at the top. But they haven’t introduced it for charts.”

Another request from the clients is for him to give them an explanation for blending data sources, and John mentions that he’s going to use a blog article he wrote, which discusses blending vs. joins as a starting point for the training session. John mentions at this point that the client team uses an Excel spreadsheet to select the stores they want to look at. He mentions that one of the disadvantages of using Tableau is that you can’t enter data into it - data comes from external sources. However, “for something that works with databases, that’s probably a good thing”. The way that John is getting around this limitation is to use a spreadsheet file as a data source, join it with the other data sources, so changes to the spreadsheet followed by a refresh in Tableau updates the visualisations.

Even though this is a more complex situation, the end goal is still a change to an existing dashboard - a modification activity. Analysing the designed customer-experience, the change would enable the experience ME3: Similar things look similar. The experiences of the analyst are similar to that of performing a work-around in the system to get what they want, which is reflected in their last comment. As such, the relevant patterns of experience that are not supported by the interface are IE1: Interaction opportunities are evident, IE2: Actions are fluid, not awkward, IE3: Things stay where you put them and TE1: You don’t need to think too hard. However, the ability to employ such hacks to achieve a desired goal could be an encouragement for creativity (patterns CE1: You can extend the language and CE4: Anything not forbidden is allowed apply).

In this case, the analyst performs an incrementation activity, by adding a new data source and combining it with the existing ones. This would later enable modification as an activity for the client (changing the data in the spreadsheet and visualising the change in Tableau), as well as sense-making (analysing the new data).

Before discussing the patterns of experience, we should note that there is one feature of Tableau has the highest effect on the other patterns. This is the ability to work with data only in read-only mode. On one hand, this enables pattern IE4: Accidental mistakes are unlikely, as data cannot be changed accidentally in Tableau. On the other hand, it limits the things one can do in Tableau.

Looking at the example above, in the context of modifying data in Excel and visualising the change in Tableau, the experience pattern IE2: Actions are fluid, not awkward is hindered, as the user (whether analyst or client) needs to switch between multiple applications to generate changes in the visualisation.

4.2. Implementing feature requests from clients
Today I’m watching Emily work. Eric, the team manager who sits next to her, mentions that they have some quick fixes for a project that they need to do, asking if she would like to do them. She says “Yeah sure” and Eric continues that some of it is tooltip changes, and a few other things. He says that they
don’t have to do them, they’re not urgent, but Emily switches and starts working on this immediately. Eric sends her the spreadsheet link on Hangouts that contains the list of the requested changes to the dashboard tool. They chat a bit about the changes she needs to make - the spreadsheet has several columns, including a description of the desired change, estimated time, status and notes.

Emily starts with the first requirement, a tooltip edit. She goes to the visualisation whose tooltip needs changing, then opens the “Edit tooltip” window and makes some changes to the text and the format. She closes the window, then hovers over a few data points to see how the tooltip looks. She then opens the tooltip editor window again and makes some more changes to the formatting. She does this several times, until she’s happy with how the tooltip looks.

Once she has decided on a format and content structure, she goes through multiple sheets that need to have a similar tooltip and makes the same changes there as well, one by one.

Once she’s done with the tooltips, she marks it as done in the requirements sheet and moves to the next one.

In the meantime, the project manager of another team comes over to Eric’s desk and they chat for a bit about another project. Emily takes a short break and goes to make herself a tea.

Once Eric finishes the chat, Emily asks him about some of the tasks in the spreadsheet which she’s marked with “???” in the “Notes” column.

For the first one, Eric points to a chart which needs changing on a specific dashboard from the workbook. Emily taking notes of her progress is an incrementation activity, with the most relevant experience being ME5: You can add comments.

For the second one he tries to explain what the client wants. He takes a piece of paper from his desk and draws a chart saying that this is what the thinks that the client expects. Emily seems slightly confused and comments that in the dashboard it’s a rolling week, so the chart shows a full week of data, and that “[she hasn’t] seen anything with missing data”. She opens the calculation of the field that is shown on the chart, and they discuss what the formula is trying to do.

After some discussion, Emily says: “This is basically saying, it’s doing the difference between the dates, and for some reason it’s adding a 2”.

Emily and Eric are talking about the calculation looking at We again notice the use of spreadsheets for keeping track of feature requests, similar in purpose to the feature trackers used by software engineering teams.

Within a PUX description, the activity Emily engages in is modification: she desires to change the tooltip text and format. The experience of interacting with the tooltip when hovering over a data point is characteristic of IE1: Interaction opportunities are evident and IE2: Actions are fluid, not awkward. However the experience of editing the tooltip is more problematic. There is a lack of PE3: You can try out a partial product - the interface doesn’t allow her to interact with the tooltip in the visualisation whilst she’s editing the contents of the tooltip. This also inhibits TE5: You are drawn to play around, as the back and forth between the tooltip editor and the visualisation can be perceived as frustrating.

PE6: Repetition can be automated is relevant here: she doesn’t have the ability to automate the change across all other similar tooltips, she has to manually edit each of them.

We observe here Eric sketching a chart in order to support the conversation (an exploratory design activity pattern), and the use of a different medium than Tableau: the pen and paper.

When the Tableau visualisation is brought back into the focus of the discussion, the analysts collaboratively aim to understand the calculated fields (a sense-making activity within the PUX framework). From a programming perspective, the analysts are debugging.
their calendars, trying to figure out why “it’s adding a 2”. Emily opens the formulas for “<Project> week” and “Calendar week” repeatedly, so that they appear on top of the other successively, and they discuss the differences between them.

Eric suggests to see how the fields actually look against a daily date. Emily makes a table in a new sheet, with “Date” as first column, then “Max date”, and then the three similar calculated fields. This creates a table similar to the one below.

<table>
<thead>
<tr>
<th>Date</th>
<th>Max date</th>
<th>Week</th>
<th>&lt;Project&gt; week</th>
<th>Calendar week</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1/11</td>
<td>1/28</td>
<td>-1</td>
<td>-2</td>
<td>-2</td>
</tr>
<tr>
<td>1/12</td>
<td>1/28</td>
<td>-1</td>
<td>-1</td>
<td>-2</td>
</tr>
<tr>
<td>1/13</td>
<td>1/28</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1/14</td>
<td>1/28</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1/15</td>
<td>1/28</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1/16</td>
<td>1/28</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1/17</td>
<td>1/28</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1/18</td>
<td>1/28</td>
<td>0</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1/19</td>
<td>1/28</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>1/20</td>
<td>1/28</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1/21</td>
<td>1/28</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1/22</td>
<td>1/28</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1/23</td>
<td>1/28</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1/24</td>
<td>1/28</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1/25</td>
<td>1/28</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1/26</td>
<td>1/28</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1/27</td>
<td>1/28</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1/28</td>
<td>1/28</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

After Emily and Eric look at the table and have discovered what the current behaviour is, they then discuss how the client wants it and what changes Emily should make. They discuss both how the visualisation at hand should be changed (Emily says that the clients probably want the weeks with value 0 and 1 to be in the chart, rather than -1 and 0 as it is at the moment), as well as how changing this visualisation would change the others that might depend on the same formulas.

After deciding how the visualisation should look, Emily spends some time working out how she can implement the change in the calculated fields.

After some investigation, Emily turns to Eric and says “I’m going to put in a parameter for this screen”. She explains that many of the other charts in the rest of the model use the 0 and -1 check for the weeks to be displayed, so if she changes the formulas that compute “<Project> week” and the other fields, it would affect the rest of the model. She then pauses and comments that she would still have to edit all of them to put the parameter in.

After a sidetrack in the discussion, Emily says that in order to make the parameter work, she will have to duplicate all the calculated fields on the current sheet to put the parameter in. Or she will have to rewrite all calculated fields with the parameter.

Eric says: “I suppose there’s no harm there, is there? That’s Eric’s suggestion to display the fields can be compared to state tracing in programming: displaying the internal state of the program with the purpose of debugging it.

The PUX activities are a combination of Exploratory design (exploring solutions for understanding the calculations) and Transcription (creating the table once the decision to create the table has been taken).

The relevant patterns of experience for the creation of the table are: IE2: Actions are fluid, not awkward, IE3: Things stay where you put them, IE5: Easier actions steer what you do, as Emily only took a few seconds to create the table.

However, the need to create the table in order to understand the results of a calculation suggests that visibility may be an issue (VE1: The information you need is visible).

This can be described as a design discussion, where the analysts evaluate what the expected behaviour of the visualisation is and how it can be implemented.

This is a discussion that reflects experiences of structure, and in particular SE1: You can see relationships between parts - Emily had to take several minute to find out how the calculations were interrelated, so the relations were not immediately visible. Also, the fact that changing one sheet would affect the rest of the workbook signals a problem for SE2: You can change your mind easily - the information structure in the software makes it difficult to isolate changes to one sheet.
the more elegant way [the second option].”

They go briefly through the other dashboards to see what would be impacted by the change. They decide to put the parameter in. Emily finishes the discussion mentioning again why using a parameter would be a good idea: otherwise she would have to go through and change 1 to 0, and that would affect the whole model and it wouldn’t give her any flexibility.

Emily returns to editing the dashboard. She creates a new “Current/Full toggle” parameter. She then edits a calculation using a large “if then/else” statement that depends on the new parameter. She writes the code in, then checks it against the helper table that she built, which is now on her secondary screen. Her work now involves editing the calculations, looking at the visualisations and seeing if anything changed, taking some notes on her notebook, and toggling parameters for testing.

Within the PUX framework, Emily performs incrementation and modification activities. She is familiar with the interface for adding parameters and editing calculations, so IE2: Actions are fluid, not awkward and IE5: Easier actions steer what you do apply for her.

She is also engaging in testing of the feature she just built, often by comparing the behaviour of the tool with her expectations (comparison activities and SE4: You can compare or contrast different parts and VE1: The information you need is visible are relevant here).

5. Discussion

We can observe that the analysts can be described both as end user programmers and professional programmers - on one hand they are end users of Tableau, and are building visual analyses for specific purposes, to answer analytical questions: the tool only matters to the extent that it allows them to achieve their own goal. On the other hand, the dashboards that they are building are then used by others, and the interactivity and ability to react to new data gives the analysts the ability to generalize the models and “productionise” them.

This creates an opportunity for building tools to support such visual analysis activities, by analogy to those which are traditionally found in software engineering. Previous research into introducing such support for spreadsheets (e.g. for debugging and testing (Reichwein et al., 1999), and for code smells (Hermans et al., 2015)) might be a useful starting point for improving such support for visualisation tools.

Further reflecting on our analysis above, we can observe that the current tool used by the analysts, Tableau, is sophisticated on some axes, but not on others. For example, the ability to do state tracing by displaying data in a table has a higher throughput than stepping instruction by instruction, as in a typical IDE for a textual programming language, and this can result in a quicker understanding of the code being debugged. In this case, the interface supports IE2: Actions are fluid, not awkward and PE2: The steps you take match your goals when the analyst creates the table, as well as TE1: You don’t need to think too hard when interpreting the table. Capabilities for finding out where a measure is used do exist in Tableau, as in modern IDEs. However, the global extraction of a parameter in the second episode is made harder by the lack of ability to search and replace through all calculated fields, resulting in poor experiences of structure and interaction (in particular SE1: You can see relationships between parts and IE2: Actions are fluid, not awkward are problematic). Refactoring features are available in most modern IDEs. Another area in need of improvement is version control -
as we have seen, duplication of a sheet is one strategy, but it is limited and prone to errors (as we have observed on a different occasion, a sheet was duplicated and renamed, but the modifications that followed were done in the wrong sheet of the two).

The analysts we studied were often engaged in maintaining software (Tableau dashboards) that they previously shared with others (clients, other team and company members). However, while clearly engaging in programming activities, the analysts refer to themselves not as “visualisation developers”, but as “data animators”⁴. This suggests that they view their practice more creatively, and that the purpose of their work is not the tools they create, but helping their users gain insights into their data through the tools.

6. Conclusion
We presented two ethnographic descriptions of work typical for a visual analytics consultancy, taken from an ongoing study. We used the PUX framework as a tool for characterizing these rich descriptions, in order to discuss the experiences of the analysts as they go about their work. We discussed two types of experiences - the designed-customer experience and the analyst experience, and observed that these are often distinct. We also observed that the analysts engage in a number of programming-related activities, which could be better supported by future tools.

7. Acknowledgements
Mariana is a Vice-Chancellor’s Scholar and is supported by an EPSRC industrial CASE studentship co-sponsored by BT. She is also supported by a Qualcomm European Research Studentship in Technology.

We would like to thank Atheon Analytics and their consulting team for agreeing to participate in this research and welcoming the first author to conduct the study in their office. We would also like to thank Luke Church for discussions of ongoing work and for his comments on this paper.

8. References

⁴ https://dataanimators.com/


Abstract

The non-acceptance of end-user computing, especially end-user programming leads to misconceptions and consequently to the underdevelopment of the wide public in computer-supported problem solving. To find methods in connection with end-user computing, the available sources and their application and usage had to be analysed, from which a selection is presented in this paper. On the contrary to the surface navigation approaches, we provide the essence of Sprego programming, which is a high mathability, computer-supported real world problem solving approach in spreadsheet environments, along with its theoretical background and tools. Sprego heavily relies on the previously published results which proved that functional languages serve novice programmers better than imperative languages, the functional data flow modelling, Technological Pedagogical Content Knowledge in spreadsheets, the functional language built into spreadsheets, and their simple interface which lessens the coding burden. We have found and proved that Sprego is an effective programming approach in end-user computing and beyond that it supports knowledge transfer between the subfields of computers sciences and other traditional sciences.

1. The state of art

1.1. The acceptance of end-user computing

“This paper presents a model for recognition of errors in documents … This is a shame: I like the paper's subject matter; but this does not mean I believe that it belongs in a computing education conference.” (private collection)

The above citation clearly describes the non-acceptance of end-user computing within computer science education, which is in accordance with the statement of Panko & Port (2013), who found that end-user computing seems invisible to IT professionals, corporate managers, and information system researchers. Beyond that we are faced with another misconception, namely that computer science is identified with computer driving license (Freiermuth et al., 2008; Hromkovič, 2009; Csernoch, 2017). Consequently, end-user computing is led by profit-oriented software companies, mostly proposing their belief in the fix nature of sciences (Wolfram, 2010, 2015; Chen et al., 2015; Csernoch, 2017), focusing on marketable software interfaces, which leads to secluded and mindless usage of tools, and ultimately to an extremely high number of error prone documents (Ben-Ari, 2011; Csernoch & Biró, 2014b, 2015c; Bewig, 2005; Burnett, 2009; EuSpRIG horror stories, 2015; Panko, 2008; Pemberton & Robson, 2000; Spreadsheets, 2015; Thorne & Ball, 2008). In general, knowledge-transfer between the subfields of informatics/computer sciences and other traditional sciences is barely detectable. The wide public – end-users, not rarely over confident end-users (Panko 2015; SCF, 2016) – are not trained to use computers in real world problem solving but they are guided and specialized – if not self-taught – to pass exams and to carry out limited, problem oriented tasks.

1.2. Functions: a link between end-user programming, programming, and mathematics

In the course of analysing the recommendations of the different national curricula and the practices, we have come to the conclusion that much less attention is paid to the notion of function in spreadsheet environments. The functional language of spreadsheets, the concept of function and the general behaviour of functions are not emphasized. This negligence of function has a noticeable consequence in that spreadsheets are not considered as a practicing field of calling, applying functions when dealing with real world problems. We can find sources which claim that functional modelling and functional languages can better serve as introductory languages than imperative languages, but these findings have

Spreadsheet functions in general are n-ary, and in special cases variadic functions. Teaching them both in maths and in ICT/CS classes would be a great opportunity to introduce n-ary functions in practice, to give examples of how the arguments can be filled in, what role the order of the arguments plays, how the different data types can be taken care of, how composite functions can be created and evaluated, and how the arguments and the values of the embedded functions are connected. This approach is in accordance with long expressed results of research in developing functional thinking “Research, including early algebra research, suggests that students’ flexibility with multiple representations both reflects and promotes deeper mathematical insights (Behr et al., 1983; Brizuela & Earnest, 2008; Goldin & Shteingold, 2001). Brizuela & Earnest note that ‘the connections between different representations help to resolve some of the ambiguity of isolated representations, [so] in order for concepts to be fully developed, children will need to represent them in various ways’.” (Blanton & Kaput, 2011).

1.3. Technological Pedagogical Content Knowledge (TCPK)
Our arguing considering the acceptance and utilization of end-user programming is in close connection with TCPK which clearly states that “…it is not sufficient for teachers to be knowledgeable about technology or pedagogy in order to use technology efficiently in the classroom. … teachers need also to know how technology can be integrated with specific content in meaningful ways.” (Mishra & Koehler, 2006). Angeli went one step further and detailed the conditions in which TCPK can be used effectively in spreadsheet environments: “teachers need to (1) develop educational rationale about why spreadsheet are important to teach, (2) understand the educational affordances of spreadsheet in teaching particular content domain, (3) identify content domains that can benefit from the use of spreadsheets, (4) be knowledgeable of students’ learning difficulties with spreadsheets, and (5) teach spreadsheets within the context of a meaningful curriculum topic” (Angeli, 2013).

2. Programming vs. “user-friendly” spreadsheet management
“I think computers are the greatest tool for conceptually understanding math. … they liberate you from calculating to think at a higher level. But like all tools, they can be used completely mindlessly…” (Wolfram, 2010).

What we primarily experience in spreadsheet environments is the mindless usage of tools. Spreadsheet programs in general do not support functionality and programming. The software companies prefer to communicate the “user friendly” aspect of these programs and environments. They declare that there is no need for any background knowledge in order to use these programs, since the available software tools can fulfill the users’ aims perfectly. Unfortunately, this approach is almost unconditionally accepted in education, where most teachers and teaching materials communicate the software companies’ profit oriented slogans, instead of focusing on the algorithmic approach to problem solving in spreadsheet environments. The software companies’ user friendly slogans emphasize the role of the environments, and they introduce more and more novel surface tools which enchant end-users, without giving any further thought to the problems. Unfortunately, this approach is highly supported by the widely accepted ECDL exams (Csernoch, 2017), the recently published Spreadsheet Competency Framework (SCF, 2016), teaching and learning materials – including printed and online coursebooks, built-in wizards and helps, and various ICT/CS curricula.

However, within spreadsheets there are tools available, although they are not emphasized, which serve high mathability problem solving, for short end-user programming. In this framework we have introduced Sprego – Spreadsheet Lego –, which fulfills all the requirements of high mathability, computer-supported real world problem solving in spreadsheet environments (Csernoch & Biró, 2013, 2014a, 2015a, 2015b, 2015c, 2015d, 2015e; Biró & Csernoch, 2014, Biró et al., 2015a, 2015b). Sprego matches the requirements of functional data flow modelling, takes further advantage of functional languages (Booth, 1992, Sestoft 2011; Hubwieser, 2004; Warren, 2004; Schneider, 2004, 2005), relies heavily on real world artefacts – authentic tables in the present environment –, the concept of function, Boolean expressions, n-dimensional vectors, and discussion and debugging.
We claim that these concepts can be introduced as early as primary maths and ICT/CS classes, and show, on one hand, how teaching mathematics and ICT/CS can be reformed, in accordance with both Wolfram (2010) and Gove (2012), on the other hand, how we can provide various examples which can develop the students’ rule-recognition and rule-following skills, help building their concept of function (Blanton & Kaput, 2011; Skemp, 1987, Vuorikari et al., 2016), and teach them programming in end-user environments. These are the tools which end-users need to make fast but reliable decisions (Csérschyh, 2017).

3. Sprengo: end-user programming in functional languages

3.1. Problem solving in Sprengo
Sprengo (Csérschyh, 2014; Csérschyh & Biró, 2015b, 2015c) focuses on the programming and data management aspects of spreadsheets, relies heavily on the concept of function, emphasizes the role of multilevel and multivariable – n-ary – functions, and applies them intensively. The concept and algorithmic aspects are at the centre of attention when using Sprengo, similar to mathematics (Pólya, 1954) and to other programming and data management environments (Vuorikari et al., 2016).

3.2. Sprengo functions
Sprengo declares that instead of the 500+ “user friendly” functions of spreadsheets, only a dozen general purpose functions – Sprengo12 – would serve as the basis for an effective high mathtibility problem solving method (Csérschyh, 2014; Biró & Csérschyh 2015a, 2015b, Csérschyh, 2017). This predicted number is in accordance with findings in programming (Hromkovič, 2014; Mayer, 1981) and in general spreadsheet use (Walkenbach, 2010).

Sprengo12 contains the following functions:

- handling strings: LEFT(), RIGHT(), LEN(), SEARCH() (in which students were found better than handling numbers (Szanyi, 2015)
- handling numbers: SUM(), AVERAGE(), MAX(), MIN()
- making decisions based on yes/no question(s)conditions, handling vectors, and handling errors: IF(), MATCH() and INDEX(), and ISERROR().

We have to emphasize here that the set of Sprengo functions is an open set; consequently, any general purpose function can be added, according to the problems which emerge (Csérschyh, 2014).

3.3. Authentic tables
It is obvious that spreadsheet environments and tables work as sources of both functional modelling and problem solving; however we have to be aware that setting school tasks in the context of ‘real world’ situations, for example through the use of word problems, is not sufficient to make them meaningful for pupils (Angeli, 2013; Ainley & Prat claimed (2005) that “…there is considerable evidence of the problematic nature of pedagogic materials which contextualize mathematics in supposedly real-world settings, but fail to provide a purpose that makes sense to pupils. … We see the purposeful nature of the activities as a key feature of out-of-school contexts which can be brought into the classroom through the creation of well-designed tasks.” However, badly-designed ‘realistic’ test items and tasks do not serve the original purpose (Cooper & Dunne, 2000) for various reasons. One of the main reasons is that students’ lack of background knowledge does not allow them to build the concept, and from that point on, solve the problem (Csérschyh et al., 2015; Csérschyh & Biró, 2016).

Burnett (2009) discussed similar findings focusing on end-user programming and programmers. She claimed that end-user programmers tend to focus on the content and the problems which they are interested in. The common ground of these different approaches is the use of authentic tables from real world contexts. Furthermore, research has clearly proved that one of the reasons for failure when teaching spreadsheets is the decontextualized and technocentric teaching methods (Angeli, 2013; Csérschyh & Biró, 2016; Mireault, 2016; Csérschyh, 2017).

Authentic tables allow students to carry out real data analyses: here the focus is not on knowledge brought into the class from outside, but on knowledge offered by the table. By becoming familiar with the table at the beginning of the process of problem solving, students can grasp the characteristics of
the data and can also reveal connections between the various items. This explains the students’ preference of text-based problems.

3.4. Array formulas
Beyond the Sprego functions, another hardly known spreadsheet tool is applied in Sprego, namely the array formula (Sestoft, 2011; Walkenbach, 2010; Wilcox & Walkenbach, 2003). Software companies recommend copying the formulas, even though this method has been recognized as one of the main sources of spreadsheet errors (Panko, 2013) along with the different reference types and their learning difficulties (Angeli, 2013), which are unavoidable when formulas are copied.

At the beginning of the learning process, array formulas rule out copying formulas and cell references, and leaving space for these concepts in advanced studies. On the other hand, we can introduce one-dimensional arrays, vectors, and use them intensively in problem solving. With this tool, we can introduce the n-dimensional vector in maths classes also, which is considered higher mathematics in most curricula, but essential in high level programming languages.

3.5. Data types
“Serious” programming and data management blame spreadsheets for negligent data-type-management (Panko & Port, 2013; Vágner & Zsakó, 2015). However, we argue that automated type recognition would serve as a convenient tool for novices. In this environment students do not have to handle the different data types manually, consequently, they can focus on the problem, the model, and the content of arguments instead of the coding details. The immediate output would also help students recognizing the data type of the output of the formulas (Data Sets, 2008). Beyond this, consciously pre-prepared tables – depending on the level of students and the goal(s) of the tasks and classes – would allow teachers to apply them with various purposes.

According to Schneider (2005), the subject of Boolean data types and Boolean functions has disappeared from some curricula in mathematics. However, in informatics it is a fundamental data type. In Sprego, Boolean expressions are introduced in the form of yes/no questions – the phenomenon transferred from language studies – and primarily used to as the ‘inside’ formula for making decisions based on the answers to the questions. Students at a very early age would form yes/no questions and decide on the output based on the answers.

Consequently, introducing Boolean data type and functions in a computer environment would also change the maths curricula. It is not only the output of formulas which can be considered in relation to data types but also the arguments and operands of these formulas. To make these formulas work the students have to be aware of the data types which the functions and the operators can accept. In spreadsheets we do this without the tiresome direct definition and declaration of variables and arrays.

3.6. Discussing, debugging
Spreadsheets are also blamed for not supporting discussion and debugging. It is true that built-in functions and the algorithms behind them are undetectable (Csernoch, 2014). However, by applying Sprego we are able to evaluate the formulas step-by-step, just as we do in “serious” programming (Csernoch, 2014, 2015; Biró & Csernoch, 2013, 2014, 2015a, 2015b; Csernoch & Biró, 2014a, 2014b, 2015b, 2015c).

However, the discussion and debugging of solutions involves far more than checking the syntactic correctness of the formulas. In spreadsheets, due to automatic type recognition, the data types always have to be thoroughly checked by the user. Automatic type recognition is both a blessing and a curse. On one hand, it helps beginners to recognize the assigned data types along with the types of the output values of the functions and reduces the coding burden, which plays a crucial role in end-user programming (Section 3.5). On the other hand, automated data type recognition would be irreversible. Undesired conversions have to be corrected or techniques have to be found to prevent automated data type recognition (Csernoch, 2014; Csernoch, 2015). Beyond these errors, similar to programming, semantic errors are the most demanding in spreadsheets.

In general, we can conclude that it is primarily the user’s responsibility to handle the errors in their work. In a teaching-learning situation, discussion and debugging, both as pre- and in-class activities, play a crucial role.
3.7. Sprego coding

Once the model and the algorithm (Csernoch, 2014; Csernoch & Biró, 2015b, 2015c; Schneider, 2005) are clarified to a problem, the coding is carried out in a spreadsheet environment (Csernoch, 2014; Csernoch & Biró, 2015b, 2015c; Schneider, 2005). Sprego supports the building of composite functions, however, it is always the user’s decision whether to introduce additional variables or arrays for displaying partial results or not – similar to traditional programming environments.

Building composite or embedded functions, we start with the innermost function. The output of the function is one of the arguments of the function outside the first one. This second function has an output again, and so on, until we reach the outermost function, whose output is the solution of the problem. At present, composite functions are barely taught in general education, however, with Sprego this concept can also be introduced at a very early age (for further details see Sections 3.8).

In Sprego the process of problem solving does not focus on browsing through the 500+ functions, but rather on the algorithm, and on calling with fast thinking a couple of familiar functions in the coding process (Kahneman, 2011). In the case of building the multilevel functions we advance from inside out, all the steps can be evaluated, the results can be displayed step-by-step, and as such, discussed and debugged. Consequently, the other advantage of this coding method is that it reduces the risk of creating erroneous spreadsheet documents. The application of array formulas allows us avoiding the repeated copying of the extracted formulas, so one frequent source of errors in spreadsheet documents.

Using additional variables and arrays and displaying all the outputs of the algorithms step-by-step has its advantages and disadvantages. Its advantages are that formulas are simple – holding only one step – and the output of each step is clearly presented. The disadvantages are that a vector or a variable has to be created for each step and the spreadsheet would be loaded with unnecessary data.

3.8. Unplugged and semi-unplugged Sprego tools

The unplugged tools invented for supporting Sprego programming would help youngsters and novice end-user programmers in building functional models and composite structures.

The simplest unplugged tools are shortened but enlarged text-based printed samples which students can cut into pieces according to the requirements of the tasks. The printed or semi-printed forms – prepared for printing but displayed on digital tools – of tables serve students as playgrounds where they can carry out the algorithms manually.

However, the main attraction of these tools is the hand-made or pre-prepared matrjoska dolls with all their accessories. A set of matrjoska dolls is a handy tool for building composite functions. The students can work with their own dolls and there is one additional set for the teacher to work on the board. Even the uncompleted sets can be used – pieces mysteriously disappear –, e.g. for displaying the mismatched pairs of parentheses.

In our experience the set of dolls are accompanied with paper balls, post-its, and stickers. The pieces of paper serve as the input variables or arrays, which can be inserted into the dolls. On the stickers the code, the steps of the algorithm, the output data, and/or the output data type can be written and placed around the doll. These stickers also serve as a tool to “close” a doll (a function), showing that whatever happens inside the doll, by closing it we only see the output, and this value serves as an input for the outside doll (function). For further specification, e.g. for distinguishing the different data types we can use different colours of pens or tapes. When students disassemble their dolls, there is an opportunity to go through again the functional model, the algorithm. Beyond that the tapes can be stickered into the students’ notebook in the proper order, without additional writing exercises.

However, the original matrjoska dolls are too valuable for frequent usage in classes. Consequently, we had to develop methods for creating dolls. We can buy readymade sets of barrels in stores. They would serve well, but still expensive. Beyond that we have experienced that the order of the colours of the sets is not necessarily the same, and usually does not match the colour of the teacher’s set. We found that our own 3D-printed sets serve us the best. There are no colour problems, the students can use their own sets from classes to classes, and they are available at a reasonable price. On further solution for creating dolls is origami. We have found that origami boats can be fold into closed objects, which can hold the smaller ones. The advantages of the paper boats are that students can create them at the beginning.
of the class – it takes only a couple of minutes –, the number of boats needed depends on the number of steps of the algorithm, there is no need for tapes and stickers – we can write on the side of the boats –, and finally, the sets can be inserted into the notebooks, as a complete task.

The pre-service teachers of our faculty are developing mobile applications with small animations for the fundamental Sprego problems (Csapó & Sebestyén, 2015). The animations display the steps of the algorithms in simple contexts, whose avatars are the matrjoska dolls. The application is under construction and development but the actual version is available and the students are open for further suggestions (Csapó & Sebestyén, 2017).

Functions can be introduced as early as the first grade maths classes (Blanton & Kaput 2004 cited in Blanton & Kaput, 2011). A typical example of dealing with functions is filling in the missing cells of a table, based on a word-task or the other way around, finding rule(s) based on the sample values of the table (Blanton & Kaput, 2011). Similar but interactive function machines can be created in spreadsheets. One example is presented in Figure 1.

![Figure 1 – A spreadsheet application and its accompanying formula to simulate a function machine.](image)

4. Summary
We have presented Sprego as a tool for developing the students’ notion of function and their programming skills in already existing spreadsheet environments.

In programming languages, handling functions is essential; however, in imperative languages, handling variables and arrays seems extremely demanding, especially for those who do not want to be professional programmers. In declarative languages the focus is much more on the problems than on the coding details. Being aware of the advantages of functional languages, the simplified environment of spreadsheets and the concept of functions introduced in maths classes, we developed a method which mathematics, ICT/CS, and end-user computing can benefit from.

5. Conclusions
It is generally understood that there is a great need for fundamental changes in both the maths and ICT/CS curricula and we are in great need of methods supporting computer-aided real world problem solving. However, it seems less obvious that the maths and ICT/CS subjects are interwoven and changes introduced in one should affect the other. It is claimed that programming would help students in developing their procedural thinking better than other previously accepted subjects, but further connections have hardly been made. Some of the approaches focus on imperative languages, while others search for less traditional programming tools. In spite of the obvious connection between the two subjects through functions, less attention has been paid to functional languages and the roles they would play in teaching functions in mathematics.

With Sprego we can revolutionize the teaching of composite and n-ary functions, can introduce vector and Boolean expressions in basic maths curricula, and offer a spreadsheet-based high mathability problem solving approach. Beyond supporting studies in mathematics, Sprego fulfils the need for an applicable method in functional languages; it takes advantage both of the functional languages and the familiar environments of spreadsheets.

Beyond the obvious connection to mathematics and programming, the extremely high number of available authentic tables would provide real world problems to solve in various classes. With this tool, Sprego is TCPK compatible, which emphasizes the importance of those contents which are interesting
for the students, connected to other subjects and generate questions, and lead to real discussions and debugging.

6. References


If What We Made Were Real
Against Imperialism and Cartesianism in Computer Science,
and for a discipline that creates real artifacts for real communities,
following the faculties of real cognition

Antranig Basman
amb26@ponder.org.uk

Abstract
I argue that we have not yet succeeded in constructing any real software, but merely simulations or cardboard prototypes of software. I criticise the pernicious imperialism of the so-called “computational thinking” which suggests that society should come to think like computer scientists, and instead suggest that computer scientists should come to think more like ordinary citizens. A greater appreciation of the cognitive underpinnings of everyday thought and everyday life will lead us to make durable, flexible things which are widely useful, as opposed to brittle, fragile things that disrupt everyday activities. We’ll make a case study of a paradigm element of traditional programming languages, function application, and consider how alternative building blocks and metaphors can lead to more successful artefacts for humans.

1. Introduction
Imagine if, when we made a piece of software for a particular community, we could be confident that there was a closely related piece of software that met the need of a closely related community. Imagine if the things we created had the status of a vigorous and imperishable characterisation of a need, rather than entering an unsustainable cycle of increasingly frantic maintenance and decay, doomed to be swept from the world in at most a couple of months or years? Imagine, correspondingly, if we could react to an “unexpected user requirement” or a change in technology or context joyfully, as a fresh opportunity to meet a freshly expressed need, as opposed to reacting with fear and despair as we wonder how much of the painful and intricate work we have done so far now needs to be undone?

1.1. Against Imperialism
Rhetoric similar to this has apparently motivated much of the products of Computer Science of the last 60 years, but as each new decade succeeds to the last, there is an increasing lack of recognition of how profoundly we are falling short of what ought to be possible. In a purely mental discipline, we suffer from none of the constraints of material and energy. Instead of delivering on these infinite possibilities, we fall back on not only complacency and cynicism, but active imperialism as we convince ourselves and our users that we have delivered not failure, but success — that rather than needing to try harder, we imagine that the rest of the world should adopt our own methods, most lately under the bandwagon of “computational thinking” (Wing, 2008), because of what we argue are their manifest success and suitability. Rather than redoubling our efforts to understand the nature of real thought and real communities, we spend our time trying to convince the world if it were “more like us”, it would be better off — and the way we choose to portray ourselves is as mechanistic, materialistic, unsubtle, inflexible and judgemental. The highest virtues of the new “computational thinking” are those most boring virtues of efficiency and correctness. Is it a wonder that most normal people are alienated by the products of the computational world which they see as an increasing stranglehold rather than an ally — a rising tide of barely functional pieces of “techno-junk” that rarely work properly, constantly promote frustrating interactions and are destined for landfill (both physical and virtual) in short order.

1.2. Towards Practice
As well as moral positioning, this paper comprises description of the aims of real communities that are doing real work steadily to bring about the change in thinking and practice that we need. The concrete results of this thought are being collected in Fluid’s Infusion framework (Fluid, 2017). In the last section are links for further reading and how to get involved. Characterising this work is not easy, since 60 years

1Fluid (www.fluidproject.org), the GPII (www.gpii.net) and others.
of increasingly entrenched thinking and practice make it increasingly hard for anyone to even see that there is a problem, or even to recognise what the work might look like that aims at a solution\(^2\). As mental horizons shrink, any work aimed at more than an immediate payoff is written off as a “boil the ocean” mission, and as each new generation of students appears, they face an ever-more complacent generation of mentors who believe that they hold the tools of solution rather than embodying the problem.

2. What Should Be Possible
Here are some more characterisations of what should be possible:

2.1. Software is worked on by means of itself
Scratch the surface of a physical product such as a chair or a wall, and you find something broadly similar underneath. The physical world is worked on by means of tools that are part of its own idiom — whether we cut a piece of wood into a smaller piece of wood, or make a hole to hold a bracket, we are using the affordances of the world itself to cause change. Contrast this with the nature of a modern piece of software or hardware — scratch the surface and underneath it is an incomprehensible world of blinking lights and mass of wiring that bears no resemblance to the physical form and affordances of the overall object. Now, hardware we can’t do much about — we are constrained by the requirements of real engineering. Software, being purely the product of the mind, should be able to be anything we like. In presenting to someone a “computational artefact”, we should simultaneously put everything they need into their hands in order to make choices about it, to work on it, to share it with others, and to find communities who have made similar (or even contrasting) choices. Instead we present them with a “locked box” with a limited number of dials to twiddle. The mystical philosophy of Sufism states that “Sufism is studied by means of itself” — what we want to bring about is a world of software that “is worked on by means of itself”. I wrote last year (Basman, 2016) on what could constitute durable, forgiving materials for software construction. The Sufi slogan is also closely related to the mantra of live programming “The thing on the screen is supposed to be the actual thing” (Ungar & Smith, 2013) which we treated in (Basman, Church, Klokmose, & Clark, 2016).

2.2. How we currently have no software
I argue that today we have no software — what we have is merely the simulation of software. What we have today bears the same relationship to real software as the set of a Hollywood movie does to the real places and scenes that it portrays. The movie set creates the impression of a particular scene which is good just for an observer in a carefully controlled place and for a limited set of purposes (the camera and its optics). Similarly, our software meets a set of needs which are good for a tiny set of users under a limited range of contexts — often this set is so idealised that the software doesn’t actually adequately meet the needs of any real users. A small change in perspective of the camera or a small change in usage (pushing against a prop wall that wasn’t designed to be rigid, for example) instantly exposes the sham of the movie set world. Similarly, a small change in requirements exposes the sham of the software we have — it may end up being treated as an entirely different piece of software with a different set of requirements — just as a complete movie is often shot with several different complete reconstructions of different scenes from different points of view or scales. We should be able to have software which is real, in that it behaves with the same continuity and consistency as real materials — real trees and real mountains expose a consistent and coherent set of linked aspects, affordances and appearances as we

\(^2\)Idries Shah, in his *Knowing How to Know* (Shah, 1998) has this to warn us in searching for a “Golden Age”: “How interesting that people think about a ‘Golden Age’ and hope for the coming or the return, of one. I have noticed that they never give any consideration to these concepts:

1. How would they know a Golden Age if they entered into one?
2. Could they survive in a Golden Age?
3. Have they been in a Golden Age, without recognising it?”
move from place to place, scale to scale and sense to sense³.

3. How we got into this mess

We got into this mess through 60 years of consistently drawing the wrong people into our field, and continuing to entrench its vices rather than reform them. In the “Garden of Eden” phase of Computer Science when such inspired products as McCarthy’s Lisp and Sutherland’s Sketchpad were plentiful as tabby cats⁴, it was easy to imagine that maturing to solve more ambitious problems for a wider class of people was just a step away⁵. In a world that has given us Java, Haskell and Ruby, success seems further away than it ever has been. Computer Science attracts people who are addicted to control — that is, their ability to have unilateral jurisdiction over some ever-increasing universe of effects and expressions. In George Orwell’s terms, such people are “power-worshippers” (Orwell, 1946) — enthusiasts of the strong simply because they are strong, and oppressors of the weak simply because they are weak. This tendency can be seen every day in the common rhetoric of the field — successful programmers are hailed as “wizards” or “ninjas” (and encourage others to do this) — glorying in power simply for the sake of power⁶. The push towards “computational thinking” is simply the same dysfunction dressed up in more respectable clothes — just as “intelligent design” is an attempt to create an acceptable, “highbrow” and intellectual packaging of the same worldview underlying creationism. In this worldview, the technologist is the one who “has power” and has mastered certain “mysteries” through the application of “correct techniques”. Others should aspire to be more like him, rather than the technologist humbling himself to put his gifts and worldview at the service of the public⁷.

Our current incarnation of this disease can be traced back at least to Newton. As argued by (Lakatos, 1978), Newton consistently falsified the nature of the methods he had used to achieve his startling results, in order to solidify his grip over the nascent community of natural philosophers. Newton argued that he had “deduced his theories from the facts”, which is a completely false account of the creative and inductive methods that he really used. Newton’s followers were completely convinced by his rationalistic account that he had achieved his results through deductive reason starting from the evidence, and went on to convince others. In this way Newton could be described as “the first wizard”⁸, in the tradition that software engineers today conceive themselves. In convincing themselves and the rest of their community to apply these methods, Newton’s followers ushered in two centuries of scientific darkness in England, in which no productive results were achieved again until Maxwell and Babbage arrived in the mid-19th century to sweep the stables clean. There are strong grounds for believing that our field is in the middle

---

³This imagery is treated in a more concrete way in (Basman, Lewis, & Clark, 2011) which describes the goal of a “homogeneous tower of abstractions” that is encountered when dealing with a single artefact from a variety of different viewpoints and scales. Our aim is to make these as closely related as possible, with as graceful, gradual and intelligible transitions between the different views, rather than the heterogeneous and unintelligible jumble that today’s “fake software” presents. It is also related to the crucial material property of “continuity” that we discuss in (Basman, 2016).

⁴Lord Chancellors were cheap as sprats, And Bishops in their shovel hats
Were plentiful as tabby cats (Gilbert, 1889)

⁵This great “retrospective hope” is the basis of an amusing and insightful presentation by Bret Victor (Victor, 2013) in which he purports to be addressing a 1973 audience comprised of “programmers of automatic computing machines”. Based on the inspiring achievements of 1973, he imagines numerous marvellous developments for the software of 40 years in the future, none of which have actually transpired.

⁶A good rhetorical example appears in (Hoyte, 2008): “but macro programming is, of course, not about style. It is about power.”

⁷(Winner, 1995), describing the UTOPIA project of the 1980s in its rare attempt to push back against the “ritual of expertise”, explains that

...those who came to the process with university degrees and professional qualifications explicitly rejected the idea that they were the designated, authoritative problem-solvers. Instead they offered themselves as persons whose knowledge of computers and systems design could contribute to discussions conducted in democratic ways.

⁸By contrast, Newton was in fact recently portrayed in (White, 1997) as The Last Sorcerer — a rewarding book which is rich in facts though thin in philosophy.
of a similar period of darkness for quite similar reasons — let’s hope we can bring it to an end in fewer than 200 years.

3.1. What’s wrong with efficiency and correctness?
Whose efficiency? What correctness? The elevation of these virtues reflects a dominant culture of accountants rather than creators. It imagines a single universal viewpoint from which these virtues can be consistently judged. (Feenberg, 1995) explains that technocratic criteria of efficiency, applied for example when considering environmental effects of technology, often result in less efficient end to end processes. But in fact, if we can’t even meet the needs of one user, what value could we ascribe to the consistency or correctness of the approaches we use to fail to meet them? It’s crucial to concentrate on positive virtues first, before turning to negative ones. Positive virtues include expressivity, the promotion of creativity, diversity of viewpoints and the understanding of relationships between them. These are the virtues that are appropriate for a young field that is not yet confident in its capabilities to do real work. As a field matures and becomes clearer about its engineering terrain, it then becomes appropriate to spend time consolidating our hold by turning to such negative virtues — negative because they involve the censoring or the restraint of expression rather than promoting it. Our field suffers right now from a kind of “premature senescence” where we somehow imagine a capability that we have not, and that it is already the time for expressing the virtues of senescence. In fact, we have merely “become old without becoming wise”.

4. What can we do about it?
Now we must seek out practical directions for achieving the aims of having real software. As we alluded to above, a significant part of this work will involve finding ways to give up power, rather than hungrily seeking it. This relinquished power will then be freed up to be delegated to our users.

4.1. Giving up power rather than accumulating it
Here are a number of kinds of power, widely considered traditional amongst software engineers, that we should give up:

i) The power to create grammars with infinite numbers of valid sentences
ii) The power to construct programs that might consume unbounded time and/or space, or perhaps never terminate
iii) The power to hide pieces of state behind abstractions (APIs or other kinds of interfaces)
iv) The power to construct pieces of software through irreversible or nearly irreversible machines such as compilers
v) The power to divide up a particular domain into a single hierarchical decomposition of entities with properties, connected by relations
vi) The power to prescribe the exact sequence of operations needed to achieve a particular result
vii) The power to import machinery, definitions or methodologies from related disciplines, without evaluating their tendency to result in appropriate products
viii) The power to change the form or behaviour of a program in an updated version, without giving a cost-free (to both users and developers) choice to retain the old form

From time to time there have been movements aimed at delegating at least a couple of these powers — for example the “sequence of operations” power vi) has a number of incarnations of technology aimed at delegating it, for example the logic programming language Prolog, or modern control flow packaging technology involving monads. But by and large the majority of these powers are not only considered sacrosanct, but also keeping hold of them has been made the basis of virtue in several major traditions.

---

9King Lear, Act 1, Scene 5: “Thou shouldst not have been old before thou hadst been wise”
10In (Clark & Basman, 2017) we discuss how this power can be relinquished in favour of working with an externalised state idiom.
11In (Basman, Clark, & Lewis, 2015) we refer to possessors of this power as entitled to express “excess artefact boundary intention”.
12In (Basman et al., 2015) we refer to possessors of power vi) as entitled to express “excess sequential intention”.
13In (Basman, 2016), this is listed as the first of our “house–clearing” tasks to be tackled as a prerequisite for establishing durable materials for software.
of engineering. For example, the power iii) of hiding state is the bedrock of Object Orientation (as is the power v) to create “entities”), and Functional Programming goes yet further in its insistence that state should not only be hidden, it should be claimed to not exist at all. Similarly it is considered axiomatic that a grammar without an infinite number of valid sentences can’t be interesting or worthwhile, and many accounts of human language try to shoehorn it into this view by claiming that these are realistic models of the kinds of languages we actually speak! Naturally this creates a number of purely factitious problems in trying to explain how learning works as a result of its blatant inaccuracy.

5. A case study: Function composition as the first evil
As an example of “importation” (the 7th power mentioned in the list of delegations), we can consider function composition, a seemingly harmless idiom imported from mathematics. One author might write the expression \( h(x) = f(g(x)) \) as a seemingly reasonable way to define a new function in terms of two pre-existing ones. This technique is actually at the foundation of the entire subdiscipline of functional programming. This kind of definition is invariably portrayed as virtuous, without a consideration of the costs incurred relative to the benefits achieved. And the costs are considerable: to the user of \( h \), the composition forever afterwards behaves as a “black box” — the inner details of \( f \) and \( g \)’s existence will never be revealed again. And the mere fact that it is such a black box is seen as the virtue rather than the vice — since \( h \) is now interchangeable for any other function achieving the same effects as the composition of \( f \) and \( g \), regardless of how they were created. The fatal difficulties that this “blind composition” poses for further creators in the same space as the original author are rarely considered. Should a second or third creator want to interpose themselves in this chain, and express some other choices relative to this application process, they have their work cut out for them. For example — imagine that what author 2 really wants is to adapt the creation of author 1 so that it reads, \( h'(x) = f(v(g(x))) \).

In many environments, this is impossible since the application point is simply notationally lost forever. In the Lisp programming language, uncovering the application point is technically straightforward since every function application is simply represented as a list data structure. However, although it is technically straightforward, it is not morally straightforward — since there is still no stable point representing the name or location of the application point of \( f \) and \( g \). That is, it has been, and can be provided with, no name that further creators could use to identify it. If the 2nd creator “happens to know” that they are faced with an expression that contains exactly 2 function applications, they can easily perform the list manipulation required (by means of a Lisp macro) to convert creator 1’s expression into the one they want. But this process is “informationally unstable” — 3rd and subsequent creators will struggle more and more with an increasingly disorderly terrain in order to find how to get their intentions expressed. This is because the 1st creator was facilitated in his crime of “creating new facilities without creating new landmarks” by the nature of the language he was provided with — the one imported from the language of mathematics.

Basing our expressions on such inappropriate idioms raises risks at all levels of design. For example, (Elliott, 2007) shows an example of this idiom surfacing at the user interface level of an application — when two elements are combined, they vanish, to be replaced by their composition, never to be recovered. Since the author, a programming language expert, regards this as a normal and virtuous design idiom, he sees no problem using this as the core building primitive exposed to users.

6. Some Practical Directions and Inspirations
Given that some of the central structuring idioms and metaphors for software construction, function composition and function application, are informationally faulty, we consider how they might be reformed.

6.1. Landmarks Rather Than Mazes
We argue that any author in the kind of terrain of “real software” that we are imagining should be facilitated by the natural modes of expression made available to him in his creative tools, to create new
landmarks that more or less keep pace with his rate of creating new facilities. This is a necessarily
imprecise statement — since it might well be burdensome for a new landmark (that is, a new named
feature) to appear for every act of composition in the environment. This would lead in the extreme to a
language similar to that used by Borges’ “Funes the Memorious” who used the entire catalogue of his
perceptions as counting numbers. However, the opposite extreme that we just considered, that of “blind
function composition” is clearly poisonous since it provides no means at all to create these landmarks
— the only possible landmarks are the functions themselves (such as \( h \) and \( f \)) rather than the application
point of the functions.

One way of seeing the problem and possible solutions can be taken from the world of web program-
moving, and the use of the DOM to represent a tree of nodes constituting the state of a web UI rendered
in a browser. The “blind function composition model” is analogous to the unreformed way in which de-
velopers of the 90s would be encouraged to navigate “blindly” around the DOM as a raw tree of nodes,
using constructs such as \( \text{myNode.parentNode.parentNode.parentNode} \) expressing the “in-
cidental knowledge” that the node of interest “just happened to be” 3 levels of containment higher in the
tree\(^\text{14}\). Compare this with the “incidental knowledge” of the Lisp programmer above who “happened to
know” that he was dealing with a composition of exactly two functions, the second of which he had an
interest in. This kind of “blind navigation” is extremely brittle in the face of acts by collateral creators in
the same space. In the “power-hungry” model we are describing in this essay, the natural response to this
situation is to try to seize more power, by finding ways to exclude other creators from the same space,
rather than trying to find ways of coexisting with them. The classic embodiment of this power-hunger
in the domain we chose for our analogy, the world of DOM programming, is the current drive towards
Web Components (W3C, 2014), an innocent-sounding name for a fascistic domain in which the rights to
navigation of the DOM by 3rd parties are eliminated. This is the form of solution that would be blessed
by a proponent of “computational thinking” — it tries to eliminate a problem by seizing more control.

6.2. Inspiration from the Web - CSS Selectors
A more appropriate kind of solution to this problem can be seen in the strategies actually chosen by
web designers in the last decade — who, unlike many computer scientists, have internalised the fact that
they must find ways of getting on with each other, as well as those with different skills and interests.
Web designers have by and large moved over to the use of (CSS) selectors in order to identify parts of a
document of interest, rather than either i) relying on blind navigation rules and/or ii) trying to find ways
of expressing unilateral control over all aspects of the document structure. These selectors are strings
with a reasonably simple format, which are able to express decisions about the identity of pieces of the
document that are of interest, that are expected to remain reasonably stable with respect to evolving
structure in the document at the hands of a community of related creators.

6.2.1. CSS Selectors as a Negotiated Space
I want to note three key aspects to the stability of reference of CSS Selectors as experienced within their
communities of use. Firstly that the stability is only “reasonably good” rather than being absolute or to
some provable standard — and that it is one that results from some process of “negotiation” with a group
of other creators. Secondly, it is enabled by certain kinds of substructure — in particular a facility for
supplying supporting names in an open way to an underlying collection of things — in this case these
names take the form of CSS class names which can be freely applied to the DOM nodes supporting
the space of selectors. These are “open” in that any creator can supply further names to any node they
are interested in — assuming that they are happy with their quality of communication with the other
creators that they are cooperating with. Thirdly, the stability is “opportunistic” — that is, each creator
can choose between a variety of tradeoffs in the strategies they use for writing selectors — ranging from
i) “chancing their arm” on existing aspects of the DOM structure without using class names, ii) piggy-
backing on some existing collection of names operated by another creator for some purposes which they
judge sufficiently related, to iii) deciding that they need to take control of a new collection of names of
their own.

\(^{14}\)Sadly as guides such as (Nativ & Fankhauser, 2009) demonstrate, this technique remains in vogue into our current decade.
Now, to a proponent of “computational thinking” this kind of messy negotiated process is simply anathema. A computational thinker is not satisfied with anything other than completely predictable results within previously agreed bounds — and is the kind of person that are seen regularly over the past 20 years declaring that “the web is broken” (Tiselice, 2015) when encountering these kinds of “negotiable solutions” rather than the “closed boxes” which their training and mentality have brought them up to expect. These negotiable solutions are in fact highly successful adaptations to the problem posed by a space in which multiple creators have to cooperate — the space of real software.

6.2.2. Selectors within Infusion
In the Infusion framework, we take a leaf out of the book of web designers and apply a highly similar solution to the problem of stably naming and identifying pieces of an implementation in an unstable or shared environment. Our IoC configuration system allows selectors in the form of IoCSS strings to match onto one or more pieces of an application, guided by their ability to match onto one or more context names as landmarks. Similar to CSS class names, these context names form an open system in that any creator may freely contribute any number of names of their own onto any existing artefact. We discuss our inspiration and use of IoCSS in (Basman, Lewis, & Clark, 2017).

In this way, we facilitate creators to create and employ landmarks, without which they would become lost in an unfeatured maze of expression trees or function applications. These “mazes without landmarks” are traditional features of languages which promote the use of unbounded recursion in the designation of artefacts — that is, those which allow grammars which permit an infinite number of sentences to describe a single artefact.

6.2.3. Landmarks as Secondary Notation
This notion and use of landmark names has an interesting status in the powerful Cognitive Dimensions of Notations framework (Green & Blackwell, 1998). Such landmarks are interesting because they could be said to occupy a position intermediate between what are called in that framework primary and secondary notations. They are intermediate because they are not primarily functional — in many cases, the entire edifice (considered as a single design) could function without them, encoding the same behaviour. This, as well as the fact that they can be freely added and removed from the structure supports the view of them as secondary. However, they are not purely secondary because without them, certain crucial uses of the artefact could not be made — that is, it could not be properly adapted into an ecology of related artefacts managed by related creators without them. They are a kind of “secondary notation with teeth”. Many of the cognitive dimensions come to have a freer meaning once one steps back from considering a single program written by a single creator (or a group compelled through Computational Thinking to behave as if they had no individuality), to considering an ecology of real software maintained for a real community — that thing which we imagine could be created. We have considered some of the implications of such collaborative dimensions in (Basman et al., 2015).

6.3. Inspiration from Optics and Reversibility
Another fruitful source of better analogies for building “real software” is the world of optics, rather than mechanics.

6.3.1. Lenses rather than machines
When dealing with light, we accept that it is going to go its own way, rather than trying to find ways of stopping it, packaging it, and manipulating it. In optical systems, components such as prisms and lenses are used to divert and redirect light as it passes from place to place — with the general expectation that the operation of the component is broadly, if not perfectly, reversible in that the effects of one such component can typically be undone by another one. In fact Newton’s Experimentum Crucis (Takuwa, 2013), proving that white lights are mixtures, and that only certain coloured lights are pure, directly took the form of “inverting” the action of one prism on a beam of light with another. This reversibility results from a crucial property guaranteed by the laws of optics, that the path traversed by any individual ray of light could be perfectly traversed by one travelling in the opposite direction. It is this interesting property which led to the centuries of confusion only dispelled by Alhazen as to whether the faculty of vision
operated by rays that were emitted from the eye in order to strike objects in the world, or conversely by rays collected by the eye which had been scattered off the objects.

This form of analogy currently has an embodiment in the Bidirectional Programming model (Foster & Pierce, 2009). We believe that such a model is crucial to delivering on many of the core facilities of real software. For example, the last power viii) from section 4.1 granted to users, the “power to resist change”, can be seen to require this kind of model — as well as our key idiom of “working on software by means of itself”. Let’s try to imagine what this entails in practice: in practice, the user is presented with a surface to a piece of software, that exists in both space and time. This surface constitutes the user interface of the software as the user operates it, as it exists from moment to moment. Presented with some behaviour on its surface, some real software would allow the user to express an intention directly coordinated with it: for example, the user might say “I don’t like this; make sure I never see this again” — or conversely, “I like this; make sure this never changes”. Without the ability to directly correspond all behaviour exposed on the surface of the software right down to the lowest-level pieces of state and configuration that the software was derived from, we could never deliver any real software. We must be able to always “reason from effects back to causes”. But what this implies is that the operation of the entire software has to be able to be conceived as the action of some kind of lens acting on these inputs — that is, that at any time we can trace the “rays” which lead out from the software to the user back in the other direction to discover their cause, and to allow the user to express their intention relative to them.

6.3.2. Free flow of information through externalisation

All of the “fake software” we have today is not like this, and does not allow this kind of reasoning. Instead, it consists of a number of “locks” through which water only flows in one direction: conducting power out from the worlds of developers into the worlds of users, and accepting no inflow in the other direction. This is precisely what APIs and abstraction boundaries, compilers and modules are designed to achieve — to concentrate power in the hands of those who have it, and to ensure that none of it leaks outwards. This falls into the “machine analogy” that we identified starting section 6.3.1: the precious resource is controlled by stopping its flow and allowing it to move only in controlled packages from place to place. Typically the person who defines the rules by which the resource is packaged has little motivation to draw up at the same time the inverse roles for unpackaging it and transmitting it in the other direction — because this involves extra work, as well as being anti-religious through giving up the control that they crave. As an example of this, consider how much pointless work is involved in every standard architecture when it is decided that at some point some crucial data structure doesn’t just have to exist privately in memory but in fact needs to be serialised to disk or wire in order to be shipped somewhere else. This normally involves a significant redesign as the same people who felt they were virtuous in designing abstraction boundaries have to suddenly scramble to discover how to circumvent them just to meet their own ends. That this work is endlessly being repeated is never interpreted as evidence that the entire enterprise of data hiding is completely misguided — developers are too well-trained in order to perceive this.

The Infusion system includes a direct embodiment of the lens model in its Model Relay and Model Transformation systems. Creators can set up publically advertised bodies of state which other creators are free to attach to, having their own copies of the data available for both reading and writing either in the original or a transformed form. End-to-end, this allows the “rays” of dependency to be traced in either direction across an entire application. We are currently working on the new Infusion Renderer which will allow this transparency to be extended by the final hop into the process of binding behaviour onto markup constituting the interface presented to users.

7. The Information Revolution Hasn’t Happened Yet

Wikipedia’s noble manifesto asks us to “Imagine a world in which every single person on the planet is given free access to the sum of all human knowledge. That’s what we’re doing.” Our mission is just the

15http://docs.fluidproject.org/infusion/development/ModelRelay.html
same, only broader. Wikipedia has now authoritatively won the battle against encyclopaedias constructed via centralised and authoritarian models. Its coverage is vastly broader, more up to date, and on average more accurate than that of any competition. But for all of its Internet-age wizardry, Wikipedia appeals to an ancient model of what knowledge is. The structure and content model of Wikipedia would have been completely comprehensible to the Emperor Xuanzong of Tang who ruled China between 712 and 756. The encyclopedia which he commissioned, the Tongdian, was itself a compilation of several previous works, and was part of an already established model of compendia which centuries later resulted in the Yongle Encyclopedia of 1408 with its 11,095 volumes occupying 40 cubic metres. This, impressive and useful though it is, is a model for “dead knowledge”—it sits on the page after it is written, and later it is read and perhaps remembered. This is the total of interaction offered by the “encyclopaedic model of knowledge”. What we aim to put into effect is a model for “active knowledge”—for which we currently have little name other than the bland catch-all term software—and it’s clear that not all software represents knowledge of this type. Active knowledge has behaviour, it is connected to communities and the real world, it has awareness of context and an individual’s faculties for producing and receiving information. A model for distilling active knowledge was described in (Winner, 1995), reporting on (Ehn, 1988)’s account of the UTOPIA project within the Scandinavian newspaper industry of the 80s, in which “key insights, lessons and prescriptions must arise from a process in which project members, regarded as equals, join to explore the properties of both technical artifacts and social arrangements in a variety of configurations.”

Much is made in the academic and journalistic literature of the so-called “Information Revolution” which is presumed to have coincided with the creation of the Web. However, I think this examination makes clear that this is no true kind of revolution since it has been accompanied by no revolutionary change in our model of what knowledge is, and how it is accessed and represented. Compare this with the Industrial Revolution, which created a model for a vast array of artefacts, modes of transport, machines—machines constructing materials, machines constructing other machines, converting and transmitting power from place to place, all products even whose categories would be hard to comprehend by the readers of the Yongle Encyclopedia. Instead of trying to push our methods into other disciplines, let us instead marvel at the incredible achievements of mechanical engineers who have produced far more substantial physical and cognitive progress even while saddled with the intractable limitations of the physical world. Rather than trumpeting our mental models, let us instead be humble and admit that we have not produced a fraction of a comparable achievement whilst being given a completely free hand to produce any imaginable structures without constraint. When the Information Revolution really comes, you can be sure we’ll know it.

8. What We Want

What we want is a new generation of Software Engineers and Computer Scientists, who are willing to give up all their imagined wizardry. Prepared to give up the recognition of their peers, industrial-scale salaries—prepared to work more slowly than they might, as a result of trying to produce work that still has a meaning 3 years in the future. Prepared to admit they have no real idea how to build software and have never seen any. Prepared to both study their colleagues and be studied, to understand what the real meaning of their work is. Prepared to read widely, both in other fields, and in the history of their own—that is, to accept that they are not wiser or have any surer models than their colleagues in other fields,

---

16This possibility is alluded to in a moving blog post from Jonathan Edwards (Edwards, 2013), responding to the selfsame Bret Victor talk that we referred to in section 3. Edwards subverts Victor’s ultimately naive narrative of how we might recover the stolen future of computer science through open-mindedness and out-of-the-box thinking, by candidly explaining that our community is going to have to incur real and substantial losses, both financial and moral, in order to reverse the directly wrong-headed course it in fact took since 1973. Edwards’ rendition of our point in the main text reads:

As always, disruption will come from our blindspot. From amateurs and children playing with toys, untainted by the sin of knowledge. Perhaps aided and abetted by a few turncoat hackers rejecting the dark side of super-intelligence.

We can also align our mission with the programme of “subversive rationalisation” announced by (Feenberg, 1995).
or in the past — and to take the time to rummage through the vast trash-heap of Computer Science to sift out the few scattered gems in it. We want a generation ready to build the true Cathedrals of software which will exist — rather than today’s imagined Cathedrals (Raymond, 1999) which to any but our own biased eyes are simply shanty-towns built out of any old trash we have to hand, destined to be swept away and built again after the first change in the weather. The builders of real Cathedrals were happy to begin on work that they knew would never be completed in their lifetimes, or even their grandchildren’s — how did we come to think we could measure ourselves against these, with our surest building blocks compared to soap bubbles? ²

9. Further reading

This paper has described the top-level motivations for our approach, and sources and models for inspiration. As we said at the outset, as well as philosophical positioning, this is the blueprint for a practical system that we are in the process of building. The core framework, Infusion is documented at (Fluid, 2017), with its externalised form, the GPII Nexus explained at https://wiki.gpii.net/w/The_Nexus. More forward-looking, speculative framework development documents are available on the wiki at https://wiki.fluidproject.org/display/fluid/Development.

10. References

²Alan Perlis: “Is it possible that software is not like anything else, that it is meant to be discarded: that the whole point is to see it as a soap bubble?”, quoted in (Abelson & Sussman, 1985).


Developing a Systematic Approach to Evaluate the Usability of Security APIs -
Doctoral Consortium Submission

Chamila Wijayarathna
Supervisors - Dr. Nalin A. G. Arachchilage and Prof. Jill Slay
Australian Center for Cyber Security, School of Engineering and Information Technology
University of New South Wales - Canberra
c.diwelwattagamage@student.unsw.edu.au

1. Introduction
Modern software development is primarily driven by the use of Application Programming Interfaces (APIs). Rather than developing functionalities from scratch, programmers find and reuse APIs that provide the functionality they are looking to implement in their code (Wurster & van Oorschot, 2009) (Myers & Stylos, 2016). APIs allow programmers to embed various functionalities to applications they develop without requiring them to know the underlying implementation details of the functionality.

APIs that provide security related functionalities, such as encryption, decryption and hashing, are categorized as security APIs. Due to the complexity of security concepts, security functionalities are implemented by developers who are experts in computer and information security (Wurster & van Oorschot, 2009). Security APIs allow programmers who are not expert in security concepts to embed security functionalities into the applications they develop.

Even though APIs are important in software development process, often they are not very easy to learn and use in software development environment (Myers & Stylos, 2016). Less usability of APIs reduces efficiency of programmers where they have to spend significant time to learn the APIs. Furthermore, less usable APIs lead programmers to incorrectly use them, which causes unintended behaviors in resulting systems.

The situation is worse with less usable security APIs. When a programmer uses a security API incorrectly, that causes security vulnerabilities in the system s/he develops. In a study Fahl et al. (2012) carried out using 13500 popular free android apps, they found that 8% of the apps are vulnerable to attacks like man in the middle attack, due to improper use of the Secure Socket Layer (SSL)/Transport Layer Security (TLS) APIs (Fahl et al., 2012). The authors have identified that the cause for this is not only the carelessness of the programmers, but also the usability issues of the SSL/TLS APIs used by programmers for developing those apps. For example, some SSL APIs provide lower level of abstraction to programmers that makes it hard for them to understand, where they get tempt to use it in unintended manners.

On the other hand, programmers who use security APIs are not security experts in most cases. They are task oriented, which sometime negatively affects the security aspects of the application they develop with the use of security APIs (Wurster & van Oorschot, 2009). The software development style can affect security of the applications developed by programmers in many ways. Therefore, it is worth designing and developing security APIs with usability in mind so that non security experts can also utilize them within their applications.

If the usability of security APIs can be improved, those will be less prone to erroneous usage and therefore, will be less subjected to introduce security vulnerabilities to the applications (Myers & Stylos, 2016) (Wurster & van Oorschot, 2009). Currently there is no existing methodology to evaluate the usability of security APIs (Acar, Fahl, & Mazurek, 2016) (Myers & Stylos, 2016). Thus, my research will contribute to develop a systematic approach to evaluate the usability of security APIs. I am planning to identify solutions to following research questions in order to achieve this goal.

1. What are the usability aspects that need to be considered when evaluating the usability of security APIs?
2. What are the steps that need to be followed when evaluating the usability of security APIs?

2. Research Methodology
By doing a comprehensive literature survey, I identified five methodologies that are being used to evaluate the usability of general APIs, which are:

- Empirical evaluation
- Heuristic evaluation
- API walkthrough method
- API concept maps method
- Automated evaluation

By considering the strengths and weaknesses of these methodologies, I identified that empirical evaluation will be the most suitable methodology to start my experiments (Wijayarathna, Arachchilage, & Slay, n.d.). Furthermore, through the literature survey, I identified that using a generic cognitive dimensions questionnaire as proposed by Blackwell and Green (2000) is the most suitable methodology to collect feedback from participants who involve in the empirical evaluation process (Wijayarathna et al., n.d.). However, existing cognitive dimensions framework and the questionnaire for API usability proposed by Clarke (2004) will not be sufficient to evaluate the usability of security APIs (Wijayarathna, Arachchilage, & Slay, 2017). Therefore, I proposed that this framework and questionnaire need to be improved to evaluate the usability of security APIs and I proposed an improved version of the cognitive dimensions framework and a questionnaire (Wijayarathna et al., 2017). Suggested improvements were derived by referring to literature on usability of security APIs (Green & Smith, 2016) (Gorski & Iacono, 2016).

First Experiment: I am in the process of conducting my first study to evaluate the proposed framework and the questionnaire. For this experiment, I selected four security APIs which cover different contexts and domains, and designed four programming tasks where participants will have to develop a program using one of these APIs. I recruited programmers to participate in this study where they did one of these tasks. While completing the task, they had to follow a think aloud study. Their computer screen and the think aloud output were recorded. Once they completed the task, they had to complete the cognitive dimensions questionnaire. Once I collect the data, I will analyze the screen recording and the think aloud results, and identify usability issues that each participant come up with. Then I will analyze the questionnaire answers and identify usability issues of the security API identified there separately. From the results, I am planning to identify answers to following questions.

- Is the cognitive dimensions framework proposed by Clarke (2004) sufficient to evaluate the usability of security APIs? If not, what are the aspects it does not cover?
- Is our proposed cognitive dimensions framework (Wijayarathna et al., 2017) sufficient to evaluate the usability of security APIs? If not, what are the alternations that need to be done for it?
- Are there any security API usability aspects that can not be evaluated using empirical usability evaluation? If yes, what are they?
- Is using a generic cognitive dimensions questionnaire (Blackwell & Green, 2000) effective in identifying usability issues in API usability evaluations?

I am submitting a work in progress paper based on this study for the PPIG workshop.
Second Experiment: I am planning to conduct the second study to identify methodologies that can be used to evaluate usability aspects that will be identified in the third question.

Third Experiment: From the results of first two experiments, I will propose a systematic approach to evaluate the usability of security APIs. Then I will conduct the third experiment to identify the applicability of proposed usability evaluation methodology to the agile software development process. This will be conducted by interviewing software quality assurance leads of software development firms that develop and deliver APIs and security APIs.

3. References


Applying Asynchronous Gaze Sharing and Social Navigation for Problem Solving Principles in an Programming Environment

Fabian Deitelhoff
Computer Science Department
Fachhochschule Dortmund
Fabian.Deitelhoff@fh-dortmund.de

Abstract
Simplex is a simplified language with a reduced set of keywords and abstractions. It can currently be used to write programs for the LEGO® Mindstorms® EV3. The idea is to enhance this language and the programming environment as part of a PhD project to use gaze sharing data and social navigation for problem solving principles in programming education.

1. Language & Environment
Simplex is the result of a R&D-Project and a Master thesis. The goal is to develop and partially implement a text-based language and development environment. Simplex programs can run on the current version of the LEGO® Mindstorms® EV3 as some sort of proxy object to provide interaction with learners and to visualize the program execution. Figure 1 shows a screenshot of the programming environment. On the left a learner can write a Simplex program. The center shows an assembler like intermediate language and on the right the generated artifact in the RBF (Robot Byte Code File) format.

2. Gaze Sharing Ideas
One part of the PhD project is to use eye tracking data to capture learners gaze data to visualize and analyze how they interact with Simplex and the environment. For example, this data can be used to test the integration of learning material in a split or in an integrated design within the environment (Jarodzka, Holmqvist, & Gruber, 2017, P. 4f).

Other possibilities are to check how participants will interact with a dynamic help systems like animated images and videos, how they’ll interact with error messages within the programming editor and how gaze paths are affected by an integrated simulation as a virtual proxy object.

Furthermore, the gaze data will be collected, analyzed and aggregated to help other learners. Therefor the gaze data will be integrated in the programming environment to provide following learners social navigation of the gaze data of previous participants. The hypothesis is that learners will benefit from information previous participants have looked at within the programming environment while solving tasks. For example, switches between the Simplex editor and the simulation or specific parts of the
integrated programming documentation. This overall scenario is an asynchronous one. There’s the opportunity to test synchronous scenarios, too in which for example two learners try to solve a more extensive task in a cooperative session.

3. Programming Language Ideas
In addition to this gaze sharing ideas there are more programming language design related ones. The goal is to evaluate at least some of them within the PhD project.

One idea is to integrate a type-inference based type system in addition to the static type system currently implemented in Simplex. Combined with eye tracking it is worth a try to evaluate how missing types will affect the reading and comprehension of Simplex code. This idea can be extended to a "Unit of Measure" system known from many functional languages like F#. In such systems one can provide units like cm and km as well as rules for this units to further enhance the type system. Thereby the compiler can provide more precise type analysis and error messages to learners.

Additionally, it is possible to test ideas in the field of programming assistance like auto completion, icons for program structures like loops and conditions, and placeholder like input mechanisms for literals (Amorim, Luís Eduardo de Souza, Erdweg, Wachsmuth, & Visser, 2016) and sensors to mention just two examples. All supported by eye tracking data and analyses.

4. Analytic Ideas
The main goal on the analytic side is to provide useful insights and appropriate measures for the expertise domain of programming. Eye tracking data of a programming environment can be one part of a methodological triangulation (see (Denzin, 2012)). Eye tracking can be useful for the perceptual aspects but other data sources are needed to complete the picture. For example, this data sources can be performance and verbal data.

It is one goal of the PhD project to come up with results (analytic ways, data sources, ...) in terms of this methodological triangulation for programming in general and programming education.

5. References


www.ppig.org

www.ppig.org

PPIG 2017

PPIG 2017

178
Problem-Solving Applications in Developer Environments

Nicholas Nelson
Electrical Engineering & Computer Science
Oregon State University
nelsonni@oregonstate.edu

Abstract
Programming is inherently a problem solving exercise: A programmer has to create an understanding of the situation, externalize and contextualize thoughts and ideas, develop strategies on how to proceed with the task, enact changes according to the most appropriate strategy, and reflect to learn from each problem. Therefore, programming is clearly more than just code input, testing, and maintenance. However, modern development environments largely focus on the “writing code” parts of programming. To support all aspects of problem solving in programming, we propose a new Integrated Development Environment (IDE) which uses a dynamic, expressive, and human-centric cards and canvas paradigm.

1. Introduction
Modern Integrated Development Environments (IDEs), such as Eclipse, IntelliJ, and Visual Studio, rely upon a fairly uniform interface of panes and windows to contain code and support different development lifecycle tasks. However, recent shifts toward distributed, service-oriented, and highly parallel software development have seen IDEs struggle to cope with the needs of developers. These developers have increasingly demanded tools that provide real-time feedback which integrates with both collaborators and customers that are increasingly embedded within the development model.

The pressure to add features that accommodate these interconnected development models has forced several IDE developers to rethink the core architectural designs of their environments. We believe that an entire reimagining is necessary in order to develop IDEs into general-purpose Problem Solving Environments (PSE).

2. Research Approach
We first surveyed the literature from the perspective of programming as problem solving, and found several activities that developers employ when programming. These activities which can be partitioned into six categories (Activities), with specific actions that represent in more detail how the high-level activities manifest themselves in practice (Actions). Clearly, not every task involves all of these problem solving actions, and there is no linearity to the order in which they are employed. Sometimes an action may not even be observable when it takes place solely in a programmer’s head. To support programming as problem solving, we proposed a new kind of IDE; a card-based IDE.

Our preliminary work will be utilized as the foundation for our IDE, which will iteratively focus on providing support for problem solving activities as development matures. We are iteratively designing this cards-based IDE, with the final goal of comprehensively supporting problem solving in programming. While doing this we will build and extend existing work, namely Code Bubbles (Bragdon et al., 2010), Variolite (Kery, Horvath, & Myers, 2017), PatchWorks (Henley & Fleming, 2014).

The IDE will be composed of an open canvas containing cards, which come in a variety of different types (code, text, sketches, web resources, etc.) and can be stacked and grouped according to the problem solving needs of developers (see Figure 1). Each card include a top bar containing a title and controls for going into full-screen mode or closing a card. Within each card is multiple card faces representing different aspects of the same content (metadata regarding interactions with that particular card, static analyzer output for a piece of code, or links to references made throughout a GitHub project in regard to the issue contained within that card, different versions of a file saved within version control), which can be accessed using a swipe motion via touchscreen or mouse.
For example, as a user, Sally has a GitHub issue card open in order to gain an understanding of the problem that she is preparing to resolve (Figure 1-1). Swiping through the related faces of the card, she finds links to both a design document and an email that are related to this issue; she opens them into separate cards and groups them into a stack (Figure 1-2). She needs to alter the current design, so she makes some sketches directly onto the design card. While exploring potential solutions, Sally finds a StackOverflow posting that contains sample code that would be helpful (Figure 1-3). She moves that postings into a card and groups it with the GitHub issue and the previously stacked design and email cards, and adds a quick annotation to provide contextual relevance to other developers and herself.

Once she has developed a strategy, she opens the relevant code file into a card and begins enacting the changes necessary to resolve this issue (Figure 1-4). As her solution takes shape, Sally realizes that her changes are large enough to require a code review. She selects the two code cards that contain her current solution and shares them with Mark (Figure 1-5). During their review session, Mark and Sally both make changes to the code and are able to simultaneously keep track of each others progress and focus on their own code.

In this new environment, new cards are generated by opening files, selecting and extracting content from other cards, or via card sharing with collaborators. This sharing functionality provides a permissions model that allows developers to share specific cards (or groupings of cards) either in a read-only mode which retains control of the content for the owner, or a synchronized editing mode that allows simultaneous updates to occur from all collaborators.

Our goal with this IDE is to explore a new paradigm of interactions. The types of interactions that best facilitate problem solving is an open problem. Therefore, we plan to conduct several user studies regarding the different potential interactions made possible by a cards-based IDE. We hope to develop both a platform for future research, and a development environment that addresses the problem solving needs of real-world developers.

3. References
Block-based languages for professionals

Robert Holwerda
Delft University of Technology, The Netherlands
r.n.a.holwerda@tudelft.nl

Abstract
Block-based languages should no longer be regarded solely as a stepping stone to text-based languages. In fact, the design of block-based languages should evolve towards a programming-UI that (adult) professionals find productive and pleasant to use, even on a day-to-day basis. Current block-based languages are not there yet.

We intend to research the design-space for block-based language interfaces with these new users and usages in mind, focussing on web designers. An initial user-study, exposing design students to a block-based version of a programming language they already know, kicks off this research.

1. Introduction
Block-based programming languages such as Scratch, E-toys, Alice and various versions of Blockly have been used successfully to introduce children to programming (ref,ref,ref). Among the aspects that contribute to this success are:

1) A *playful domain* that is fun and motivating, such as simple games (Scratch, E-toys), interactive multimedia (Scratch), puzzles featuring characters from cartoons and games (Hour of Code version of Blockly), simple robotics (other versions of Blockly), and mobile phone applications (MIT AppInventor)
2) The *programming language is kept simple* and comprehensible by limiting the set of commands, using an intuitive semantics, and avoiding or simplifying complex programming concepts (such as object orientation, static typing or data structures more complex than lists). Alice is an exception here, but it is geared towards older children.
3) A UI-design that allows the *direct structural manipulation of program-components*, which is both intuitive (drag-and-drop) and eliminates the need to learn and edit syntax that delineates structure, such as bracket, braces, comma’s, (semi) colons etc.
4) The UI makes the *language highly discoverable*: A palette presents building-blocks (commands, data-literals and control structures) visually, and individual blocks display labelled edit fields and menus for their parameters and visual indicators for how to connect the block to others. Users can rely on recognition instead of active recall of available components, and explore available options while solving problems inside the programming environment.

Each of the block-based languages mentioned is an educational language, and they are often seen as a first step for children, serving two purposes: (1) training the children in computational thinking (ref), and (2) preparing some children to make the transition to “real” text-based programming languages (ref). In our view, however, block-based languages hold much promise for application to a broader range of end-user programming situations, and for many people who are not children. In particular, aspects 3 and 4 are desirable in software where language-like information is edited by users who can’t be expected to memorize the vocabulary and grammar of one, several of even many textual languages. Take, for instance the many languages used by system operators to build, configure, deploy and maintain complex systems. Or the proliferation, in the web development world, of mark-up languages, style-languages and JavaScript variants, libraries and frameworks.

Blockly, being a toolkit with which different block-based languages can be created, has already been used to create languages that aim to be both useful and easily accessible to different kinds of professionals who are not software engineers (ref,ref,ref). These efforts, however, have not investigated whether the block-based user-interface that Blockly supports (and which closely mimics Scratch’s UI) could be adapted or enhanced to further support the needs and preferences of
professionals such as sysops, interaction designers, artists, data analysts and other end-user programmers.

We aim to research the design-space for block-based language interfaces aimed at end-user programmers who:

- do not need aspects 1 (playful domain) and 2 (simplified language) of the listing above;
- do not intend to use the block-based interface as a stepping stone toward text-based computer languages, or as an introduction to regular programming language semantics (all educational block-based languages mentioned above, embody an imperative structured programming language);
- want to be able to easily use some language(s) causally and sporadically, partly due to aspect 3 (direct structural manipulation) and 4 (high discoverability of language);
- but also, might grow into power-users and experience the language-UI as highly productive in day-to-day use;
- work with languages that may not fit the programming language grammar style (e.g. markup-languages like HTML and Latex where elements and text content can be mixed quite freely);
- work with language vocabularies and documents that are much larger than current block-based languages support comfortably;
- might benefit from immediate feedback from the system (e.g. type inferencers or live execution) and intuitive refactoring affordances that could be integrated into the block-based GUI more naturally than in text-based IDE’s.

While we aim for a prototype that, like Blockly, can be applied to many different domains, we will be focussing on end-user programmers in the web development domain, primarily web designers who want to create working prototypes of their designs, including data-exchange with servers and other users.

2. First research activity

We have conducted a user-experience test to get a first exploratory sense of what web-designers consider useful or problematic in block-based language-interface. The participants were ten students of a 4-year multimedia-design bachelor programme (in their final year), who were tasked to create part of a moderately complex Arduino-application. On the basis of their 3,5 years of design-study, including at least a 5 month in-company internship, we consider these students to be sufficiently representative of our target audience (web designers). These students were also about 6 weeks into an entry-level programming course (covering JavaScript and Arduino), giving them enough experience to perform programming tasks that are not absolute beginner’s level, but not enough for them to be averse to block-based languages just because of a long familiarity with text based code.

The participants were observed while performing two programming tasks lasting about 70 minutes, using a Blockly variant aimed at creating Arduino software that supported programming tasks that were very similar to task they had already performed in text-based Arduino code. Afterwards, they were interviewed about their thoughts and feelings about the way the interface supported the tasks. A screen recording of the task performance, including an eye-tracker overlay, was shown to the participant in order to prompt the participant to reflect in specific moments during the task performance. This method is called a Retrospective Think Aloud test which is considered to yield data that sheds more light onto the cognitive aspects of the user experience (as opposed to Concurrent Think Aloud, where the user’s comments tend more to describe actions and sensory perceptions).

We are currently analysing the results of this test, and can not yet report definite findings. We will, however, informally discuss two things that were noted by the experimenter in the sessions with almost all participants.

3. Sources for improvements in the block-based user experience

Participants were quickly looking for UI-features they knew from professional graphical design tools such as Sketch or Adobe Illustrator, specifically keyboard-shortcuts and other features that speed up common tasks in direct manipulation interfaces like copying objects, creating selections of multiple
objects etc. This is, of course, not unexpected with participants with a background in design. But an
often-stated motivation was that the drag-and-drop interface felt cumbersome when the user was
executing a plan involving multiple steps he/she had already in mind. Rather than dismissing their
requests for features found in graphical design software as mainly a desire to work in familiar ways,
we surmise that these tools have been perfecting drag-and-drop direct manipulation interfaces (and
accommodating user’s need for speed of manipulation) for decades, and that block-based language
UI’s could improve for many more kinds of users by learning from the UI-design of these
professional design tools.

Another source for ideas to improve block-based UI design turns out to be text-based languages. Like
many Blockly-based tools, the ArduBlockly tool used in the test displays the generated textual source-
code next to the canvas where the blocks are edited. We did not remove this source-code panel,
because it provides a very quick way to our participants to learn the semantics of the (Arduino-
specific) blocks offered by the tool, given that they already understand Arduino C code. Many
participants, however, used the source code panel for a second purpose: to quickly scan the entire
code of their program (their task started with a given program of 177 blocks, equivalent to 72 lines of
C code) while looking for specific parts of the program, and getting an overall sense of the program’s
structure. During the interviews, some participants stated that they were simply more familiar with the
text-based code, so could scan and read that code more easily. Others, however, explained their
preference for reading the text-based code by pointing out several aspects of the block-based UI: the
abundance of distracting colors, the verbosity of labels in blocks, the messy feel caused by unaligned
top-level blocks, the unavailability of a linear sequence for scanning code, and the need to switch
between scrolling horizontally and vertically, or dragging the canvas (which was error-prone: one
could accidentally drag blocks apart instead of making another part of the program visible). These
complaints point to the possibility that some design-decisions that work for children editing small
programs (e.g. free-form placement of blocks on 2d-canvas) are, currently, not working well at larger
scales. We do not suggest that a block-based language UI for professionals should emulate the linear
nature of plain text files, but that it is worth finding out if designs are possible that combine the best of
both worlds.

These are not the only sources for design improvements we have in mind. The Cognitive Dimensions
of Notations framework was very much developed as a tool for designers to discuss and evaluate
designs and design goals. A systematic analysis of block-based language UI, using the CDN
framework, and with our new set of design goals in mind, is likely to yield useful and important
requirements and directions for exploring the design space. This analysis will also incorporate the
results of the Cognitive Dimensions questionnaire that all the participants of our user experience test
have filled out.

4. Conclusion
Block-based programming languages have some demonstrated benefits that can help end-user
programmers of all kinds. But being designed for children learning to program, they are likely to have
some drawbacks that might prevent professionals from adopting block-based tools as part of their
day-to-day work. In initial user study, focused on web designers is currently being analysed, and is
expected to result in requirements and design opportunities for evolving block-based language UI’s to
be useful and productive for new user groups and new use cases.