Computational Thinking: Magical Book Magic

Paul Curzon and Peter McOwan
Queen Mary University of London

You take a book that involves Witches or Wizards, Macbeth for example, and demonstrate how magic has seeped into the words of such books over the ages. The volunteer picks a word from the start of the book and then, letting the book itself direct them, they end up with the word that no one could have possibly known, but that you predicted at the outset having hidden the prediction in an envelope that they have held all along.

In exploring how the magic works, you learn about computational thinking: especially the importance of evaluation to algorithmic thinking. You explore both testing and hazard analysis. The magic trick shows how computer scientists, engineers (and magicians) have to check their algorithms thoroughly. They must think carefully about how things might go wrong as well as checking they will go right.

The Magic

You give a volunteer from the audience a clipboard and pen. On the clipboard are written the opening lines to Shakespeare's Macbeth, the famous scene where the three witches meet. It is a version without any stage direction, just the words spoken. A dotted line is drawn after the first sentence. The last part of the quotaation is coloured red.

When shall we three meet again

In thunder, lightning, or in rain?

- - - - - - - - - - - - - - - - - - -

When the hurlyburly's done,

When the battle's lost and won.

That will be ere the set of sun.

Where the place?

Upon the heath.

There to meet with Macbeth.

I come, Graymalkin!

Paddock calls.

Anon.

Fair is foul, and foul is fair:

Hover through the fog and filthy air.

Later in the play the witches meet Macbeth and make a prophesy that he will be King. But words said aloud over and over come to have a magic of their own. The opening lines of Macbeth have been said so often, that, because they are words about magic, those who know how can use them to make their own predictions.

Get three more volunteers to speak aloud the words of the three witches from a book of the play. As they read the words have the first volunteer with the clip board check that the words on their clipboard are exactly the same words that the witches speak. Here are the words as you should find them in a book with the directions of who speaks what:

First Witch
When shall we three meet again
In thunder, lightning, or in rain?

Second Witch
When the hurlyburly's done,
When the battle's lost and won.

Third Witch
That will be ere the set of sun.

First Witch
Where the place?

Second Witch
Upon the heath.

Third Witch
There to meet with Macbeth.

First Witch
I come, Graymalkin!

Second Witch
Paddock calls.

Third Witch
Anon.

ALL
Fair is foul, and foul is fair:
Hover through the fog and filthy air.

Now, words spoken, magic swirling through the air, the time is ripe for prediction.

Get the volunteer to put their hand on the book of the play and "draw on the magical power that has infused throughout it over the centuries". Have them then pick a word from the first sentence of the quotation (before the dotted line), as spoken by the first witch:

When shall we three meet again
In thunder, lightning, or in rain?

For example, they might choose 'three'. They should circle their chosen word, telling everyone what it is. Now explain that they are to let the book control them. They will jump around the text, guided by the words themselves, allowing the magic of the book to take control. Explain that they should stop the moment they land on a red word. The red word will be their ultimate chosen word.

Explain that to do this they should count the number of letters in the chosen word, and count on that many words to land on a new word and circle it. For example, if they chose the word, 'three', then it has five letters, so they should count on five words and land on ‘lightning’ which they should circle. Let them do it with their word, double checking that they haven’t made a mistake.

They should do this repeatedly bouncing from word to word, each time counting the number of letters in the ***new*** word and moving on that far. They should ignore punctuation including apostrophes. They should also ignore all the stage direction in the book version, which is why you’ve given them a version with only the words spoken.

Get them to keep going until they land on a red word for the first time. They should tell everyone what that word is.

Once they have landed on a red word, have them agree that it was their own free choice of which word to choose, and that they had no idea what word they would ultimately land on - no one knew.

Now explain that using the magic that is in all books about those who practice magic, you made a prediction. Point out that there is an envelope on the clipboard under the paper that the volunteer has been writing on. You placed it there before the trick started. Have them open it. Inside is a piece of paper with a single word written on it: “HEATH”: which remarkably is also the red word that they ended on. The magic-infused words unlock the power of prediction.

Deeper Magic?

Now you know the trick, you can do it yourself, even if you don’t understand how it works (yet). If you start with the words from Macbeth, it will always work. In fact it is even more magical than that. It will work with just about any book about witches, wizards or other magic. Try it with the words from the start of a completely different book: ‘The Wizard of Oz’, for example. They are given below.

Dorothy lived in the midst of the great Kansas prairies, with Uncle Henry, who was a farmer, and Aunt Em, who was the farmer's wife.

Their house was small, for the lumber to build it had to be carried by wagon many miles.

There were four walls, a floor and a roof, which made one room; and this room contained a rusty-looking cooking stove, a cupboard for the dishes, a table, three or four chairs, and the beds.

Choose a word from the first sentence, follow the same steps again stopping when you first land on a red word in the third sentence. I predict you will have landed on the word ‘FOUR’.

Want more evidence. Try yet another completely different magical book: ‘The Cat in the Hat’ by Dr Seuss. It’s about a magical cat. Don’t trust me to give you the words. Go find a copy yourself. Choose a word from the first sentence. I predict the first word you land on when you get to the second page will be “SAT”.

The magic is even deeper than that. It will work with just about any writing, though sometimes the sentence to colour red is further on than in others.

An algorithmic conspiracy?

So is this some powerful, deep magic? Or has their been a conspiracy amongst authors, all sending some kind of coded message? Perhaps it’s something to do with the Knight’s Templar, Da Vinci Code like?

Actually no. Its just a simple, if at first surprising, computational property of words that has been turned in to a magic trick. Magicians call this kind of trick a ‘self-working trick’ because it always works if you follow the steps. You don’t have to know how or why it works to make it work. Just follow the steps and the magical prediction will be right. Computer Scientists call instructions like this an **algorithm**. Algorithms are what computational thinking is all about. The skill of inventing algorithms is called **algorithmic thinking**. Algorithms allow us to solve problems without having to think too much. We can just follow the steps of the algorithm and the right thing happens even if we don’t know what we are doing or why. That, it turns out, is exactly what we need for computers. A computer doesn’t know what it is doing or why. They just follow programs. Programs are just instructions for a computer to follow, written in a special language - a programming language suited to the task. They just describe algorithms telling a computer how to do useful things. Descriptions of magic tricks and programs are essentially the same thing - algorithms. Magic tricks describe algorithms for magicians to follow that result in magical effects.

Programs describe algorithms for computers to follow that result in…well whatever the programmer wanted the computer to do.

Testing Times

I've claimed the trick always works for that passage from Macbeth. Do you believe me enough to try the trick for yourself on a live audience? How can you be sure? Well, you could try it a couple of times. Would that convince you that it will always work? Does that give you convincing enough evidence? What if they pick a word you didn't try. You have no idea whether it works for that one. You will look a bit silly if your prediction turns out to be wrong.

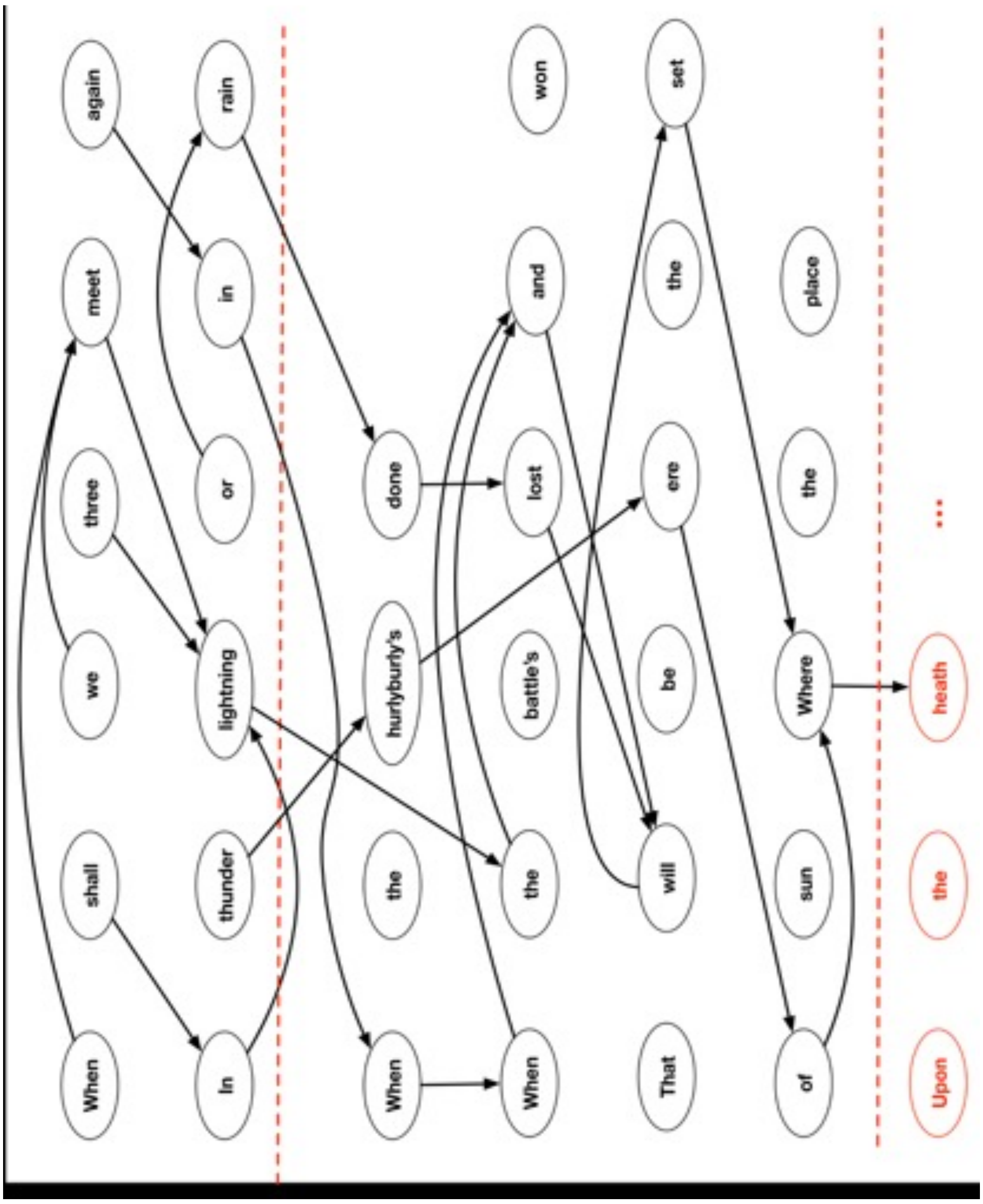
There is only one way to be really sure, and it sounds a bit tedious. You have to check every alternative. You have to be very patient if you are to be a good magician or a good computer scientist. Careful checking and attention to detail really, really matters. Programmers call checking programs like this, **testing**, and more time is normally spent testing new programs than writing the code in the first place.

More generally checking programs are fit for purpose is called **evaluation** and it is an important part of computational thinking. It is only by the careful and detailed checking of algorithms that you can be sure they do always work.

Doing things carefully and methodically matters. We want to be sure we really have checked all the possibilities. The easiest way to do this for our trick is to start at the first word, check it works, then move to the next, check it and so on, until we have done them all. We are doing algorithmic thinking again: what I just described is an algorithm for checking our magic trick works. We can use algorithmic thinking even when evaluating algorithms! Since we now have an algorithm to do it, we could even write a program based on it to check our trick rather than do it by hand. If you can write programs, you might want to try it. If not, check the magic trick really does always work for Macbeth and the Wizard of Oz by hand.

We can actually come up with a slightly better algorithm to do the checking, that involves less work. Here is how. If we actually circle each word as the volunteer did in the actual trick as we check the paths then we can see which ones we have checked already. If we add an arrow showing where we jump to, we can see the paths we take too. Most of the words bounce us to a new word in the first sentence that could have been chosen as a starting word. When it is their turn to be checked we won't have to do anything more as we will already have checked them as part of checking the earlier word.

The diagram below shows the result of checking every word from the first sentence in Macbeth. As part of checking the word 'when', for example, we bounce to 'meet' and then to 'lightning'. When it is the turn of those words to be checked we will have nothing more to do as having checked the trick works for 'When' we have already checked those words too as they follow the same path. With a bit of simple logical thinking we have reduced the number of paths to check from 12 to only 7, almost halving the work to be done.


When shall we three meet again
In thunder lightning or in rain
When the hurlyburly's done
When the battle's lost and won
That will be ere the set
of sun Where the place
Upon the heath
...

We can reduce the work needed further still. If at any point a path takes us to a circled word we have already checked then we need go no further in checking that path.

In doing the checking, you have to focus on the detail, being very careful and precise about what you do. Its easy to make a mistake counting. If you do, then you might find one day your trick doesn't work after all. That means its best, having done it once, to double check your work. By drawing the arrows you've made that slightly easier. You just need to check every word has an arrow out, and every arrow jumps the right amount.

Try it yourself this way for ‘The Wizard of Oz’ and ‘The Cat in the Hat’ too. Now I claimed it works for lots of books. Perhaps it works for all books. How could we check that? We would have to work through every book in turn, and see if the paths for each did join together as in Macbeth. If ever we found one that didn’t work we would have proof in that book that it doesn't work whatever an author writes. However, each time we check a new book that it does work for, we’ve proved nothing conclusive. We would have to keep testing to be sure it works for the next book and the next… To prove that it worked for every book (if it does) we would need to check every single book ever written. That would take far too long. We need some logical thinking instead.

So how does it work?

First though, why does it, surprisingly, work for so many books (at least three so far)? Did I pick those books specially just because they do work? Not really. It does work for lots and lots of books. Think again about what happens in that first sentence as the paths of some words join up like ‘when’, ‘meet’ and ‘lightning’. Once paths have joined they don’t split up again. Any short word in that first sentence is just going to join with another word that could have been chosen. They will end up on the same red word. Only a few words, mainly ones at the end of the sentence or very long ones, will jump straight to the second sentence. In fact only four words jump out of that first sentence: ‘thunder’, lightning’, ‘in’ and ‘rain’. What appeared to be 15 possibilities for the volunteer to choose from is actually only four different choices of onward paths: only four paths cross the dotted line onwards. A similar thing will happen with any book. Long first sentences may look like they are offering lots more choices but actually only a few words will still be long enough to escape as separate paths. Once we are beyond the dotted line, of course, no new paths are introduced. We don’t have to consider ever word from that point on, only those we can land on from one of the existing paths.

So what happens to those four paths for Macbeth? Does that still mean there are four possible red words we could end up on. Well no. As their paths bounce around, if ever they happen to hit the same word, then the two paths will merge and stay together from then on. You can see that that is what happens in the diagram for Macbeth. The further we go the more likely it is by chance that the different paths that are left will land on some shared word. Once they do there is one less possibility for what the final word could be. For Macbeth they all converge just 20 words later.

The amazing thing is actually not that so many writers have written books so that we can predict a word, but that any might have managed to write a book where we can’t.

All books?

Does it work for all writing? We can answer that now with some **logical thinking**. If it doesn’t we should be able to think through a way that a book might be written so that at least two paths don’t join up? Well, if two words are the same length, then they will jump to positions the same distance apart. If that keeps happening the paths will never meet. So if we have a book that is something like the following, for example, then the trick won't work.

> The dog saw the rat,
> but the cat ate him.
> The dog was sad for the rat.
> …

Here all the words are three letters long, so we end up with only three paths straight away. However, those paths just bounce along together, never joining up. If every word in a book is the same length like this, the trick will fail to converge on a single word how ever far you go. Once we can see that simple case it is easy to see that other variations could lead to the trick not working too. So without checking any real book at all we have worked out that it is possible for the trick not to work on some books. There certainly could be writing that the trick doesn't work on, though most are likely to work if you go far enough.

So the trick isn't fool proof for all books. With similar reasoning we can also see what will help make it work quickly. The paths will generally join quickly, for example, if the book uses lots of short words but that have different lengths: say lengths 2, 3 and 4 letters.

Avoiding disasters

Just as we tested the trick for all possibilities to prove it works, programmers test their programs over and over again to make sure they always work whatever happens. When a computer crashes, it means you have just found a situation that the programmers didn't test. It isn't your fault, it is theirs, though their job is difficult. It is virtually impossible to write complex software, without it containing mistakes somewhere. Why? Because modern software may be millions of instructions long, and it takes perfect attention to detail to get it right. It is just too easy for the programmer to make mistakes in the instructions and not notice. Testers don't find the problems because there are just too many possibilities to check them all as we did for the trick.

Testing software is actually more like checking the trick works for all the books in the world than checking that Macbeth or the Wizard of Oz work. There might be millions, billions or more possibilities to check, and it just can't be done in any reasonable length of time. Instead programmers test what they hope is a representative sample. They then cross their fingers and hope. That is why modern software like apps are constantly being updated with 'bug fixes'. It is only when millions of people start using the software that it starts to get properly tested. Often software companies release what they call 'Beta' versions of a new program that they let people use to try and find the problems before they release the final version people pay for. They are using their customers as their testing team.

For most apps bugs are perhaps just an inconvenience, but there are many situations where they can cause really big problems. For example, when you pick up a phone to dial the emergency services you don't want to find the phone is dead because of a software bug. In early 1990, it happened across the whole of the United States. Telephone company AT&T's engineers had upgraded the software of their 114 US switching centres. They are the computers that make the connections so your phone links to the one you are calling. On January 15th 1990, they stopped working properly: 70 million calls failed. The problem was in a few lines of code out of millions, and the programmers who had changed them hadn't tested for all eventualities. AT&T lost $1 billion as customers fled to their competitors.

In 1996 an Ariane 5 rocket exploded forty seconds after lift-off. The project had taken 10 years, and cost $500 million. This spectacular software failure was due to squeezing a large number into the computer memory reserved for a small one. As it went faster, there wasn't enough space to hold the rocket's speed when it was passed to another smaller memory store. This caused the rocket to veer off course, break up and explode. Again the testers had missed testing this critical situation so no one realised the mistake until it was too late, though 40 seconds in to a real flight was long enough to find it!

NASA's Mariner 1 spacecraft was supposed to fly by Venus. Instead it made it as far as the Atlantic Ocean. Unfortunately, a mistake was made writing the program. A hyphen was missed out that should have been there: a simple grammatical error, but lethal in a computer language. Testing the software didn't find the problem. The flight software miscalculated the rocket's trajectory and the rocket lost control. It had to self-destruct before it caused bigger problems.

In 1992 the London Ambulance service introduced a new dispatch system that was used by those taking emergency calls to send out ambulances. Things started to go wrong within a few hours and more than 20 people may have died as a result of ambulances not arriving in time to save them. The new system had both software and hardware problems. It turned out that some aspects of the system hadn't been tested thoroughly enough, and had not been tested under the full load it would have in a real situation.

In 2000 an Osprey hybrid aeroplane / helicopter crashed after a mechanical failure - a hydraulic line broke. That shouldn't have been a disaster though. The crew correctly pressed the button to reset the flight system. Instead of helping as it should, a bug in the system caused the plane to stall and ultimately crash.

In 2013 a CareFusion infusion pump - a device that pumps drugs in to hospital patients at a controlled rate - was recalled. There was a software bug that meant the control panel buttons could stop working while it was in the middle of delivering drugs.

In 2014 software from Baxter used by pharmacists for eight years to calculate doses of nutritional fluids for patients who could't eat or drink was also recalled. Four different software bugs had been discovered, one of which meant it sometimes got the calculations wrong, giving double the correct amount.

These kind of problems keep happening and occur in many more systems than the few above. Testing is an important part of any evaluation, and the testers have to test as completely as possible. But testing alone just isn't good enough to find the problems in complex software. There are just too many possibilities to check. There are also problems testing won't find.

Especially, in critical situations, programmers and hardware designers use other evaluation methods too. For example, they use logical thinking to work out what to test, trying to ensure they cover, if not every situation, a full range of representative situations. They also use logical thinking to prove mathematically that crucial part of programs do always do the right thing. That's a bit like the reasoning we used but to prove that a system does always work. Yet another way involves focussing on what might go wrong …

Avoiding the Hazards

To ensure software is as reliable as possible, you need to think about what might go wrong as well as just checking if things are right. Part of this process is called **Hazard Analysis**. The engineers try and work out what bad things could happen. They then work out the ways they might actually happen. Finally they work out how they might prevent them happening or make sure the consequences aren't too bad if they do.

We did a hazard analysis for our magical book trick and made changes from the very original version to mitigate the hazards. We had to think what might ruin the trick. The most obvious thing for that trick is that the prediction we make is wrong. That's a hazard we have to avoid. How could that happen? Well the most likely way is that the words don't all actually land on that same particular word. We are mitigating against that by testing every possibility … then double checking in case we make a mistake checking the first time.

There are other ways the same hazard could occur though. For example, the volunteer could make a mistake, either miscounting the letters in a word or miscounting how far to jump. We mitigate against that by checking each step as they do it. Perhaps that's not enough as we have a lot to think about. We could do even better by enlisting the audience to check with us. We could put the words up on the screen and circle them as we go, getting the audience to count with the volunteer. They will point out any mistake.

Another way the prediction could be wrong is if the envelope has the wrong thing in it. We might have forgotten after the last time we did the trick to check the volunteer put the right thing back, or perhaps we might confuse envelopes with the prediction for another trick. Ways to mitigate against that hazard include always keeping the props for different tricks separate and always double checking everything when setting up before a show. Also part of the point of having the envelope clipped on the clipboard is to mitigate against it going wrong in that way. Everything is clipped together so less likely to get mixed up. We also write on the outside of the envelope what is supposed to be inside it, and use different coloured envelopes for different tricks.

Notice that testing of the algorithm - checking that every starting word leads to the predicted word won't help at all with these other ways the hazard could happen. Testing is only a small part of the evaluation of algorithms. The system as a whole that the trick works within has to be checked too.

Another hazard is that the trick works in that the prediction is correct, but that the audience don't see the trick as particularly magical. Ultimately you will need to try out a trick on real people to evaluate it in this way, but you can still use logical thinking to think through ways it might go wrong.

Our trick might not seem magical, for example, because the first sentence is quite short with only a few words to choose from. That could be solved simply by choosing a book with a long first sentence or using the first couple of sentences rather than just the first. It might also occur because the audience assume you have just found a book that it happens to work for. One way round that might be to start with three different books about magic each sitting on their own clipboard. You could then get an audience member to choose which to use. That would emphasise it is about the magic in the books. On the other hand that would increase the likelihood of another hazard: that you get envelopes mixed up, leading to the prediction being wrong. That's one of the problems with hazards. If you aren't careful, sorting one out can just lead to another biting you even harder. It could even introduce completely new hazards you haven't thought about.

The software equivalent to a trick being magical or not is called **user experience**, and you need to evaluate software to make sure it does give its users an appropriate experience as well as that the algorithm works. For a game that might be that it is fun, for business software perhaps that it is not frustrating to use, for a shopping website that it leaves shoppers with a feeling of delight. Experts in user experience are employed to evaluate software in this way. They have to understand people as much as machines.

A thorough hazard analysis would look for even more ways things could go wrong to spoil the trick. What if no one will volunteer, for example? That's a new hazard again. How would you avoid that? Ultimately you weigh up how likely the hazards are, and how bad it will be if they do happen. What matters is whether the ways you have found to mitigate the problems mean the chances of them happening are small enough to be acceptable (there will always be some small chance that things will go wrong).

The hazard analysis process for software is very similar just with different hazards. A hazard for a pharmacy system like one described earlier is that it leads to the wrong amount of drug being administered to a patient. A hazard for a plane is that it stalls, for a rocket's guidance system that it veers off its intended trajectory. For any program there are many hazards, many ways for the hazards to occur and many ways to mitigate them. As we have seen these hazards can all be caused by software but also could be caused by other problems too. To reduce the risk of a hazard you need to find ways to mitigate all the ways it could happen.

Computational Thinking

Both magic tricks and programs are created using **algorithmic thinking**. In either case it is important to check that the algorithm you have created is correct. That involves **evaluation**. One kind of evaluation is testing where you check the trick or program works over and over again. This requires a lot of patience, care and attention to detail. The best kind of testing is **exhaustive testing** where you check every possibility - like checking every starting word in the magic trick really does work. For real software exhaustive testing isn't possible so other kinds of evaluation need to be done too. This is especially so when the software is to be used in critical situations. Many of the complementary ways used by computer scientists involve logical thinking - thinking through why a program (or trick) always works but also thinking through ways it might not work too. Hazard analysis is just one important technique for doing that. There are many aspects of both tricks and software that are important; from whether it works, to whether all the hazards have been mitigated and whether it gives a good experience. All need evaluating thoroughly if a program or trick is to be a success.

Use of this booklet

This booklet was created by Paul Curzon and Peter McOwan of Queen Mary University of London, cs4fn (Computer Science for Fun www.cs4fn.org) and Teaching London Computing (teachinglondoncomputing.org).

We are grateful for support provided by Google and the EPSRC funded CHI +MED project involving QMUL, UCL, Swansea and City Universities. This booklet was distributed in London with support from the Greater London Assembly.

See the Teaching London Computing activity sheets in the Resources for Teachers Section of our website (http://teachinglondoncomputing.org/resources/inspiring-unplugged-classroom-activities/) for linked activities and resources based on this booklet.

Department for Education

SUPPORTED BY MAYOR OF LONDON

COMPUTING AT SCHOOL EDUCATE · ENGAGE · ENCOURAGE

Version 1.0, 15 August 2015