Developing computational thinking in the classroom: a framework

June 2014

Working group of authors:

Prof. Paul Curzon
Queen Mary University of London, School of Electronic Engineering and Computer Science
Teaching London Computing Project (http://www.teachinglondoncomputing.org/), funded by the Mayor of London and Department of Education through the London School's Excellence Fund

Mark Dorling
BCS, The Chartered Institute for IT and Computing At School Network of Excellence project (http://www.computingatschool.org.uk), funded by the Department for Education, industry partners and awarding bodies
Digital Schoolhouse London Project (http://www.digitalschoolhouse.org.uk), funded by the Mayor of London and Department of Education through the London School's Excellence Fund

Thomas Ng
West Berkshire Council School Improvement Adviser (ICT & Assessment)

Dr. Cynthia Selby
Bay House School and Sixth Form, Gosport, Hampshire
Southampton Education School, University of Southampton

Dr. John Woollard
Southampton Education School, University of Southampton
BCS, Chartered Institute for IT Barefoot Computing project (http://www.barefootcas.org.uk), funded by the Department for Education

Introduction

Computational thinking sits at the heart of the new statutory programme of study for Computing:

> "A high quality computing education equips pupils to use computational thinking and creativity to understand and change the world" (Department for Education, 2013, p. 188).

This document aims to support teachers to teach computational thinking. It describes a framework that helps explain what computational thinking is, describes pedagogic approaches for teaching it and gives ways to assess it.

Pupil progression with the previous ICT curriculum was often demonstrated through 'how' (for example, a software usage skill) or 'what' the pupil produced (for example, a poster). This was partly due to the needs of the business world for office skills. Such use of precious curriculum time however has several weaknesses. Firstly, the country's economy depends on technological innovation not just on effective use of technology. Secondly, the pace of technology and organisational change is fast in that the ICT skills learnt are out of date before a pupil leaves school. Thirdly, technology invades all aspects of our life and the typically taught office practice is only a small part of technology use today.

In contrast, the new Computing curriculum has an enriched computer science element. Computer science is an academic discipline with its own body of knowledge that can equip pupils to become independent learners, evaluators and potentially designers of new technologies. In studying computer science, pupils gain not only knowledge but also a unique way of thinking about and solving problems: computational thinking. It allows the pupils to understand the digital world in a deeper way: just as physics equips pupils to better understand the physical world and biology the biological world. Simon Peyton-Jones gives an account of why learning computer science and computational thinking is a core life and transferable skill in a talk filmed at TEDxExeter (Peyton-Jones, 2014).

To prepare our pupils to understand the consequences of technological change, adapt when using technologies, develop new technologies or even to work in jobs that haven't yet been invented, not only does the 'what?' and 'how?' of the subject need to be taught, pupils also need to develop techniques to ask and be able to answer the question 'why?'. Computational thinking supports doing so. Computational thinking skills are the set of mental skills that convert "complex, messy, partially defined, real world problems into a form that a mindless computer can tackle without further assistance from a human." (BCS, 2014)

Today, however, there is an interpretation, led by the popular media, implying that the new computing curriculum focuses on 'coding' (Crow, 2014; Nettleford, 2013). This gives a misleading message, especially to those new to the discipline. In contrast, our framework presented below aims to support teachers' understanding of computational thinking across the full breadth and depth of the subject of Computing and offers a way to easily and effectively integrate it into classroom practice.

The framework

There are four interconnected stages of development to our computational thinking framework:

Stage 1: Definition

Stage 2: Concepts

Stage 3: Classroom techniques

Stage 4: Assessment

We overview each in the subsequent sections.

Stage 1: Definition

To support the sharing of curriculum materials and classroom practices, an agreed definition that is suitable for the classroom is needed. We use the interpretation forwarded by Professor Jeannette Wing, who originally popularised the idea of computational thinking. She defines it as:

> "… the thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by an information-processing agent" (Cuny, Snyder, Wing, 2010, cited in Wing, 2011, p.20) ... "these solutions can be carried out by any processing agent, *whether human, computer, or a combination of both"* (Wing, 2006).

We chose this definition because it is based on Wing's original definition and has gained consensus amongst academics. Its emphasis is on pupils performing a thought process, not on the production of artefacts or evidence. It therefore fits the direction of change in the current curriculum development.

Stage 2: Concepts

The next stage is to define the core concepts involved in computational thinking. Based on a review of academic references, Selby and Woollard (2013) suggest the following are key:

- algorithmic thinking
- evaluation
- decomposition
- abstraction
- generalisation

We outline these concepts with examples below, giving linked classroom techniques in the next section.

Algorithmic thinking is a way of getting to a solution through clear definition of the steps - nothing happens by magic. Rather than coming up with a single answer, like 42, the pupils develop a set of instructions or rules that if followed precisely (whether by a person or a computer) leads to answers to that and similar problems.

For example, we all learn algorithms for doing multiplication at school. If we (or a computer) follow the rules we were taught precisely we can get the answer to any multiplication problem. Once we have the algorithm we don't have to work out how to do multiplication from scratch every time we are faced with a new problem.

Evaluation is the process of ensuring an algorithmic solution is a good one: that it is fit for purpose. Various properties of algorithms need to be evaluated including whether they are correct, are fast enough, are economic in the use of resources, are easy for people to use and promote an appropriate experience. Trade-offs need to be made as there is rarely a single ideal solution for all situations. There is a specific and often extreme focus on attention to detail in computational thinking based evaluation.

For example, if we are developing a medical device to deliver drugs to patients in hospital we need to be sure that it always delivers the amount of drug set and that it does so quickly enough once start is pressed. However, we also need to be sure that nurses will be able to set the dose quickly and easily without making mistakes and that it won't be frustrating or irritating for patients and nurses to use. There is likely to be a trade-off to be made between speed of entering numbers and helping avoid mistakes being made when doing so. The judgement about it being quick and easy has to be made systematically and rigorously.

Decomposition is a way of thinking about problems, algorithms, artefacts, processes and systems in terms of their parts. The separate parts can then be understood, solved, developed and evaluated separately. This makes complex problems easier to solve and large systems easier to design.

For example, if we are developing a game, different people can design and create the different levels independently provided key aspects are agreed in advance. Through decomposition of the original task each part can be developed and integrated later in the process. A simple arcade level might also be decomposed into several parts, such as the life-like motion of a character, scrolling the background and setting the rules about how characters interact.

Abstraction is another way to make problems or systems easier to think about. It simply involves hiding detail - removing unnecessary complexity. The skill is in choosing the right detail to hide so that the problem becomes easier without losing anything that is important. It is used as a way to make it easier to create complex algorithms, as well as whole systems. A key part of it is in choosing a good representation of a system. Different representations make different things easy to do.

For example, when we play cards, we use the word 'shuffle'. Every player understands that 'shuffle' means putting the cards in a random order. The word is an abstraction. The same type of abstraction works when programming. Implementing 'shuffle' in a computer game means giving a way to randomise the cards. We can refer to shuffling throughout the program and understand what is meant without having to think about how it is actually done by the program. All that is needed is that the program does include a description somewhere of how shuffling is to be done.

As an example illustrating the difference the representation can make, consider an art project. Pupils studying Monet could take a digital picture of a Haystack painting in a gallery. In doing so they have created a representation of it on the computer as pixels. They can then easily manipulate this digital representation in ways that would be very hard with a different representation or in the real world. For example, the colours could be changed by an algorithm. In this way a series of different but related versions of the painting could be created.

Generalisation is a way of quickly solving new problems based on previous problems we have solved. We can take an algorithm that solves some specific problem and adapt it so that it solves a whole class of similar problems. Then whenever we have to solve a new problem of that kind we just apply this general solution.

For example, a pupil uses a floor turtle to draw a series of shapes, such as a square and a triangle. The pupil writes a computer program to draw the two shapes. They then want to draw an octagon and a 10-sided shape. From the work with the square and triangle, they spot that there is a relationship between the number of sides in the shape and the angles involved. They can then write an algorithm that expresses this relationship and uses it to draw any regular polygon.

In summary, each of the above techniques fits into the well-established system design life cycle of computing projects in the business, academic and scientific communities. In practice they are used together in a rich and interdependent way to solve problems. The emphasis in these concepts is on practical techniques or thought processes, not on the production of artefacts or evidence.

Stage 3: Classroom Techniques

The descriptions of the concepts above are high-level. Although important, on their own they don't explain how computational thinking can be embedded into the classroom and integrated into pedagogy. Therefore, our next step (Table 1) is to identify learner behaviours associated with each.

Concept	Examples of Techniques
Algorithmic Thinking	Writing instructions that if followed in a given order (*sequences*) achieve a desired effect; Writing instructions that use arithmetic and logical operations to achieve a desired effect; Writing instructions that store, move and manipulate data to achieve a desired effect; (*variables* and *assignment*) Writing instructions that choose between different constituent instructions (*selection*) to achieve a desired effect; Writing instructions that repeat groups of constituent instructions (*loops/ iteration*) to achieve a desired effect; Grouping and naming a collection of instructions that do a well-defined task to make a new instruction *(subroutines, procedures, functions, methods*); Writing instructions that involve subroutines use copies of themselves to achieve a desired effect (*recursion*); Writing sets of instructions that can be followed at the same time by different agents (computers or people) to achieve a desired effect (*Parallel thinking and processing, concurrency*); Writing a set of rules to achieve a desired effect (*declarative languages*); Using a standard notation to represent each of the above; Creating algorithms to test a hypothesis; Creating algorithms that give good, though not always the best, solutions (*heuristics*); Creating algorithmic descriptions of real world processes so as to better understand them (*computational modelling*); Designing algorithmic solutions that take into account the abilities, limitations and desires of the people who will use them;

Evaluation	Assessing that an algorithm is fit for purpose; Assessing whether an algorithm does the right thing (*functional correctness*); Designing and running test plans and interpreting the results (*testing*); Assessment whether the performance of an algorithm is good enough; Comparing the performance of algorithms that do the same thing; Making trade-offs between conflicting demands; Assessment of whether a system is easy for people to use (*usability*); Assessment of whether a system gives an appropriately positive experience when used (*user experience*); Assessment of any of the above against set criteria; Stepping through algorithms/code step by step to work out what they do (*dry run / tracing*); Using rigorous argument to justify that an algorithm works (*proof*); Using rigorous argument to check the usability or performance of an algorithm *(analytical evaluation*); Using methods involving observing a system in use to assess its usability or performance (*empirical evaluation*); Judging when an algorithmic solution is good enough even if it is not perfect; Assessing whether a solution meets the specification (criteria); Assessing whether a product meets general performance criteria (heuristics)
Decomposition	Breaking down artefacts (whether objects, problems, processes, solutions, systems or abstractions) into constituent parts to make them easier to work with; Breaking down a problem into simpler but otherwise identical versions of the same problem that can be solved in the same way (*Recursive* and *Divide and conquer* strategies)

Abstraction	Reducing complexity by removing unnecessary detail; Choosing a way to represent artefacts (whether objects, problems, processes or systems) to allow it to be manipulated in useful ways; Hiding the full complexity of an artefact, whether objects, problems, processes, solutions, systems (*hiding functional complexity*); Hiding complexity in data, for example by using data structures; Identifying relationships between abstractions; Filtering information when developing solutions;
Generalisation	Identifying patterns and commonalities in problems, processes, solutions, or data. Adapting solutions or parts of solutions so they apply to a whole class of similar problems; Transferring ideas and solutions from one problem area to another

Table 1: Computational thinking concepts and associated techniques.

Examples of algorithmic thinking, evaluation, decomposition, generalisation and abstraction, are found at all stages; it is the context that determines the relevance and challenge of the activity. We have therefore tried not to attribute computational concepts and learner behaviours to particular key stages (phases of education) because doing so would imply that they are age-dependent in a way that they are not: they are capability dependent.

It is also important to emphasise that computational thinking concepts are not the content for the subject of 'Computing'. The subject content is set out in the national curriculum programme of study. Computational thinking skills enable learners to access parts of that subject content.

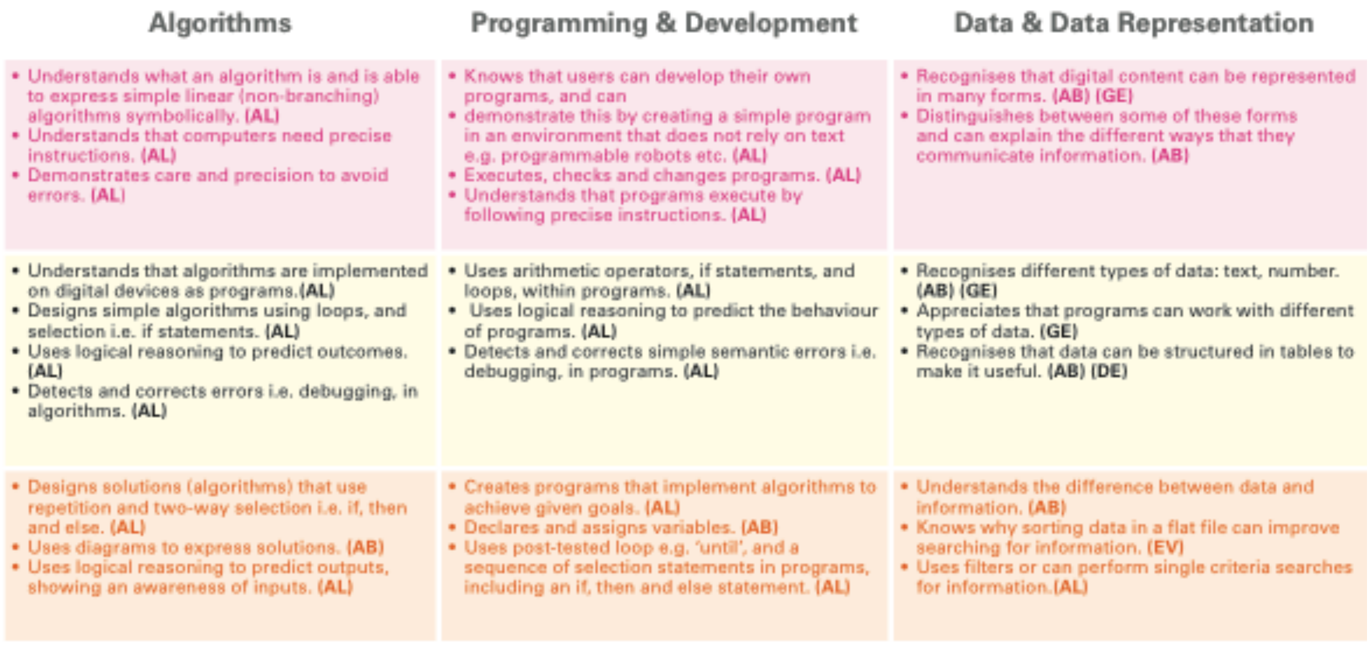
Stage 4: Assessment

The final stage needed is a way to assess the increasing competence of pupils in computational thinking. This can be done using an adapted version of the existing subject framework for the computing subject itself.

To support classroom teachers, Computing At School published an assessment framework called 'Computing Progression Pathways' (Dorling and Walker, 2014a). It sets out the major knowledge areas of computing and gives specific indicators of increasing levels of mastery of the subject in those areas. This assessment framework was produced by a small team of authors and reviewers (all teachers and academics) based on their classroom experiences. It is an interpretation of the breadth and depth of the content in the 2014 national curriculum for the computing programme of study. This breadth affords an opportunity to view the subject of computing as a whole, rather than the separate subject strands of computer science, digital literacy and information technology proposed by the Royal Society (2012). The assessment framework identifies the dependencies and interdependencies between concepts and principles as well as between the three subject strands.

Separate pathways are given for the areas of algorithms, programming & development, data and data representation, hardware & processing, communication & networks and information technology.
For example, the pathway around the subject area of algorithms at its lowest level involves understanding of what an algorithm is and an ability to express simple linear algorithms with care and precision. It then moves through levels of being able to express more complicated algorithms using selection and loops, to at the highest level being able to design algorithms that make use of recursion as well as having an understanding that not all problems can be solved computationally.

The assessment framework is also presented where the learning outcomes are organised by the separate subject strands of computer science, digital literacy and information technology (Dorling and Walker, 2014b). A further version has been developed to incorporate provision for the concepts of computational thinking (Selby, Dorling and Woollard, 2014). It now includes a description of how it can be used to acknowledge progression and reward performance in mastering both the content of the computing programme of study and the ideas of computational thinking (Dorling, Walker, 2014c). For example, algorithmic thinking is demonstrated not just in the Algorithms and Programming & Development pathways, but also in constructing appropriate search filters (Data & Data Representation) and in demonstrating understanding of the fetch-execute cycle (Hardware & Processing). See Figure 1 as an example of what you can expect to see in Computing Progression Pathways with computational thinking.



Algorithms	Programming & Development	Data & Data Representation
• Understands what an algorithm is and is able to express simple linear (non-branching) algorithms symbolically. **(AL)** • Understands that computers need precise instructions. **(AL)** • Demonstrates care and precision to avoid errors. **(AL)**	• Knows that users can develop their own programs, and can • demonstrate this by creating a simple program in an environment that does not rely on text e.g. programmable robots etc. **(AL)** • Executes, checks and changes programs. **(AL)** • Understands that programs execute by following precise instructions. **(AL)**	• Recognises that digital content can be represented in many forms. **(AB) (GE)** • Distinguishes between some of these forms and can explain the different ways that they communicate information. **(AB)**
• Understands that algorithms are implemented on digital devices as programs.**(AL)** • Designs simple algorithms using loops, and selection i.e. if statements. **(AL)** • Uses logical reasoning to predict outcomes. **(AL)** • Detects and corrects errors i.e. debugging, in algorithms. **(AL)**	• Uses arithmetic operators, if statements, and loops, within programs. **(AL)** • Uses logical reasoning to predict the behaviour of programs. **(AL)** • Detects and corrects simple semantic errors i.e. debugging, in programs. **(AL)**	• Recognises different types of data: text, number. **(AB) (GE)** • Appreciates that programs can work with different types of data. **(GE)** • Recognises that data can be structured in tables to make it useful. **(AB) (DE)**
• Designs solutions (algorithms) that use repetition and two-way selection i.e. if, then and else. **(AL)** • Uses diagrams to express solutions. **(AB)** • Uses logical reasoning to predict outputs, showing an awareness of inputs. **(AL)**	• Creates programs that implement algorithms to achieve given goals. **(AL)** • Declares and assigns variables. **(AB)** • Uses post-tested loop e.g. 'until', and a sequence of selection statements in programs, including an if, then and else statement. **(AL)**	• Understands the difference between data and information. **(AB)** • Knows why sorting data in a flat file can improve searching for information. **(EV)** • Uses filters or can perform single criteria searches for information.**(AL)**

Figure 1: Mapping the learning outcomes from Computing Progression Pathways to the concepts (from Stage 2) of computational thinking.

Using the framework to plan lessons

When planning and teaching a scheme of work in any subject, teachers refer to the planning-teaching-evaluating cycle. Computational thinking can be included in the planning stage in four steps within the planning phase of each lesson in the planning-teaching-evaluating cycle, see Figure 2.

Step 1: Determine the 'why' at the start of the unit of study (Stage 1) as well as the possible topics (the column header names from the Progression Pathways Assessment Framework) that the scheme of work will be covering.

Repeat steps 2 - 4 when planning each lesson in a unit of study

Step 2: Decide 'what' the learning outcomes are for the lesson from the Computing Progression Pathways Assessment Framework (Stage 4), which enable the pupils to move closer to completing or achieving the 'why'.

Step 3: Use the predefined mapping in the Computing Progression Pathways Assessment Framework to identify the possible associated computational thinking concepts (Stage 2).

Step 4: Use the computational thinking concepts to identify possible techniques 'how' to incorporate into and highlight as part of the chosen classroom activities (Stage 3).

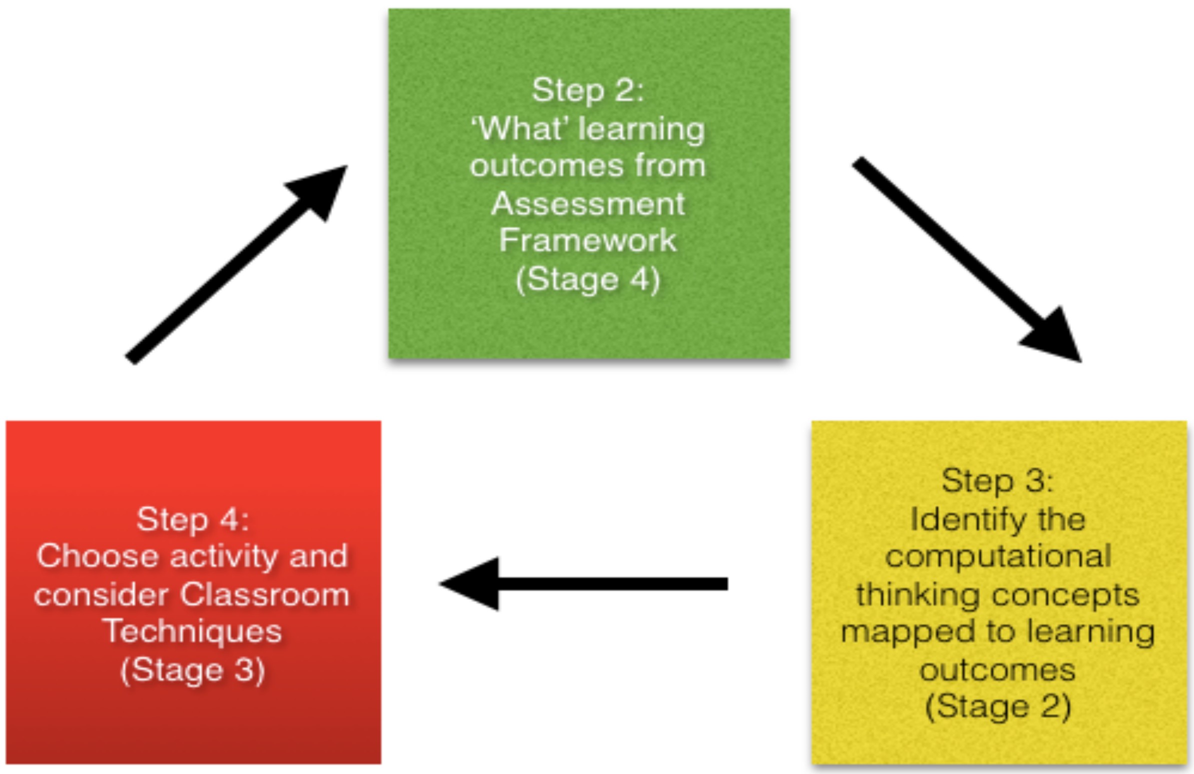

Step 2:
'What' learning outcomes from Assessment Framework (Stage 4)
Step 3:
Identify the computational thinking concepts mapped to learning outcomes (Stage 2)
Step 4:
Choose activity and consider Classroom Techniques (Stage 3)

Figure 2: Mapping the 4 stages of the framework to 'why', 'how' and 'what'.

It is important to note that the most important step in this process is the last step (step 4). Just because pupils can evidence learning in the Computing Progression Pathways Assessment Framework and that the learning outcome is mapped to computational thinking, it does not necessarily mean that the pupils will have performed computational thinking. Completion of an activity, in itself, is not evidence that computational thinking has occurred.

A Case Study

Below, we illustrate the application of the above framework with a case study, based around a lesson one of the authors (Dorling) has used in his classroom. In the sub-section of each activity, we highlight how different parts of the activity draw on the computational thinking concepts (CT). In the classroom, these concepts could be drawn out explicitly in, for example, a discussion at the end where the pupils reflect on the computational thinking skills they have used through the activity.

Topic

Networking & Communications - using a binary protocol to transfer information

Why

I first lead a group discussion aiming to draw out why networks are important. We discuss the applications pupils use on a regular basis, such as a search engine or network file shares and how these applications have completely changed the way we do things. I lead pupils to ask "what actually happens in the wire to make information go back and forth?"

How

Activity 1) Recap - I remind the pupils that they have previously studied and understood the different layers involved in computer architecture: applications, the operating system and the hardware.

- ***(CT) Abstraction*** *of functionality*
    - *As we move from **hardware** to **operating system** to **applications** we move through increasing layers of system abstraction as each hides the messy details of the level below.*

Activity 2) I introduce the pupils to the layers of network architecture: application, transport and network and point out the similarity to the computer architecture layers.

- ***(CT) Abstraction*** *of functionality*
    - *In a similar way we move up through similar layers of abstraction from the **network** to **transport layer** to **applications** as each hides the messy details of the level below.*
- ***(CT) Generalisation of solution*** *(applying the same technique to a similar problem)*

        - *We have transferred **the technique of analysis by layers** from computer architecture to network architecture.*

Activity 3) I remind pupils of their understanding of denary (decimal) numbers stored as binary numbers, that is denary numbers are an abstraction of the binary code. They hide the detail of how the numbers are actually stored. I suggest that they could use this knowledge to invent their own transportation layer protocol.

- ***(CT) Abstraction** of data*
    - ***Denary numbers** conceal the complexity of the binary representation*

Activity 4) I give the pupils a simple circuit, i.e. a battery, wires and a lamp, and ask them to transfer a decimal number across the room to a friend using the lamp.

It is up to the learners to perform the conversion into binary and transfer it across the room. I encourage them to think of the different tasks involved. The sender and receiver do different though related things. The recipient will receive the number, assemble the string of binary and convert the binary back into a denary number.

- ***(CT) Decomposition** of a problem*
    - ***Identification of the high-level steps** necessary to accomplish the whole task*
- ***(CT) Algorithmic thinking***
    - *Development of the **ordering of the high-level steps** necessary to accomplish whole task and **working out the detailed steps** for each.*

Obviously without an agreed protocol there is complete mayhem. Pupils have to work together to agree a protocol for 1 (light on) and 0 (light off). The confusion continues until the pupils realise the time or clock element that is needed so the start point is known and the light is either on or off for two seconds with a one second pause between each on or off.

- ***(CT) Evaluation** of functional correctness*
    - *Pupils **reflect on the problems** (even mayhem) of initial solutions and realise the need to improve them*
- ***(CT) Algorithmic thinking***
    - *The t**rial and feedback development loop** used between pupils is the heuristic development of an algorithm*

An alternative activity for pupils who have not yet fully grasped binary is to have them look at historical communication methods they have heard of such as Morse code or smoke signals with a view to identifying similarities between them and the current challenge.

- ***(CT) Generalising** a solution from one problem to another*
    - *Identifying that in each case one representation (a **letter**) is transformed into another (**Morse code**), recognising a pattern in the solutions.*

Activity 5) A standard protocol is agreed amongst the whole class, this was achieved through a discussion of the problems of interoperability if every pair has chosen a different way of communicating. They are then given a series of numbers the first two identifying the person (e.g. table-individual) and the next two being the message to that person (rather than an actual IP address at this stage of learning)

- *(CT)* ***Abstraction*** *of data*
    - *Understanding that an* ***IP address is a name for a machine***

Pupils again struggle with this as it can be difficult with a long string of binary, so they are likely to come up with an idea to chunk or group the binary. This is analogous to a packet.

- *(CT)* ***Abstraction*** *of data*
    - *Inventing t****he concept of a chunk or packet,*** *with chunks being sent, received and reassembled.*
- ***(CT) Algorithmic thinking***
    - *Working out t****he detailed instructions*** *to make the chunking work.*

Activity 6) Having mastered these concepts, we discuss IP addressing as analogous to the UK post code system.

- *(CT)* ***Generalising*** *a solution from one problem area to another*
    - *Recognising a pattern in the solutions to* ***network packet sending*** *and* ***sending a letter by post***

Future learning opportunities can be built on these foundations. For example, visual packet tracing tools can be used to consider the location of web servers around the world. Digital literacy questions can be posed about breaking the law when using the Internet and considering in which country a crime may have been committed.

What

From the activities discussed here, the pupils have had opportunities to use techniques associated with computational thinking concepts as indicated in order to demonstrate their understanding of the programme of study content. Depending upon the level of understanding expressed or observed in the pupil behaviours, it is possible to award progress in the subject content from the computing pathways at the following levels:

Pink Level

- Algorithms: Understands what an algorithm is and is able to express simple linear (non-branching) algorithms symbolically; Demonstrates care and precision to avoid errors.
- Information Technology: Talks about their work and makes changes to improve it.

Yellow Level

- Algorithms: Designs simple algorithms using loops and selection i.e. if statements; uses logical reasoning to predict outcomes; detects and corrects errors i.e. debugging, in algorithms.
- Information Technology: Talks about their work and makes improvements to solutions based on feedback received

Orange Level

- Algorithms: Recognises that some problems share the same characteristics and use the same algorithm to solve both.
- Data & Data Representation: Understands the difference between data and information.
- Communications & Networks: Understands the difference between the internet and internet service, for example, world wide web.
- Information Technology: Makes appropriate improvements to solutions based on feedback received and can comment on the success of the solution.

Blue Level

- Algorithms: Designs solutions by decomposing a problem and creates a sub-solution for each of these parts.

Purple Level

- Data & Data Representation: Understands how bit patterns represent numbers and images; knows that computers transfer data in binary.
- Communications & Networks: Understands data transmission between digital computers over networks, including the internet i.e. IP addresses and packet switching
- Algorithms: Can identify similarities and differences in situations and can use these to solve problems.
- Information Technology: Uses criteria to evaluate the quality of solutions, can identify improvements making some refinements to the solution and future solutions.

Summary

To engage pupils in lessons and so get the best out of them, it is important that they understand why they are learning topics. Some materials supporting the previous ICT curriculum focused on what was being taught, (perhaps a skill) and what the pupils produced (perhaps a spreadsheet model). Thinking about 'what' and 'how' the pupils were producing an artefact but 'why' they were learning a given skill were secondary considerations. The 'why' was often an assessment objective or a qualification examination instead of a real-world reason. Criticism of this approach identified a lack of focus on understanding the deeper 'how' and 'why' (problems are solved, systems are made, …) (Royal Society, 2012).

The four-step framework we have set out gives a practical way to both understand computational thinking and introduce the ideas into the classroom context. It can be used both to support the planning of activities to increase the opportunities for pupils to develop computational thinking skills and to assess their progress in doing so.

This can be achieved by considering the 'why' of the challenge they are setting for the learners at the outset. Pupils should then employ a variety of their computational thinking abilities as described in Table 1 (the 'how') to develop understanding or solve the problem in hand. The 'what' is expressed in the evidence of the actual subject learning. This could be what the pupils produce (artefact), what the pupils understand or express (write, test, verbalise), or what behaviour is observed (generalising). The 'what' matches the learning outcome statements from the Computing Progression Pathways Assessment Framework. Figure 3 maps the four stages of development described above to the notion of focusing on the 'why', 'how' and 'what'.

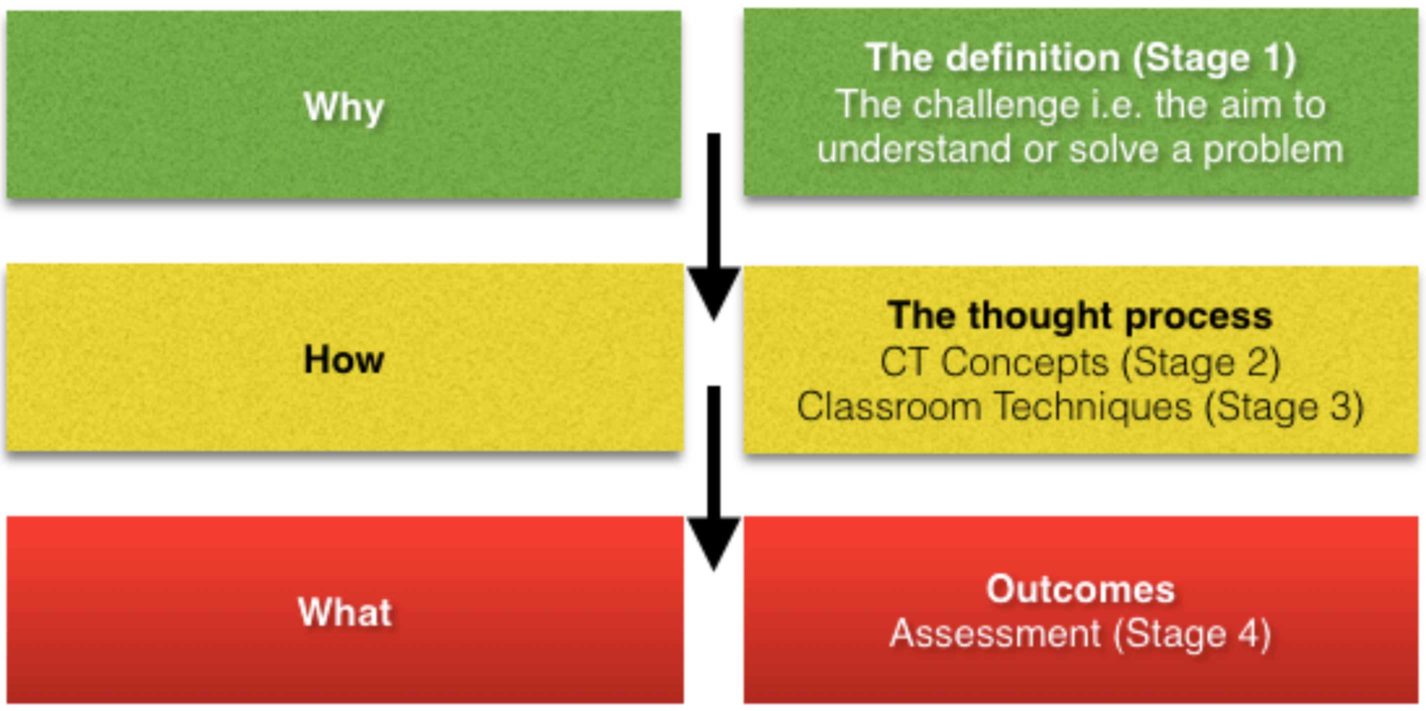


Figure 3: Mapping the 4 stages of the framework to ‘why’, ‘how’ and ‘what’.

References

BCS, The Chartered Institute for IT. 2014. *Call for evidence - UK Digital Skills Taskforce.* Available: http://policy.bcs.org/sites/policy.bcs.org/files/BCS%20response%20to%20UKDST%20call%20for%20evidence%20final.pdf [Accessed 26-06-2014].

Department for Education. 2013. *The National Curriculum in England, Framework Document.* Available: http://www.education.gov.uk/nationalcurriculum [Accessed 23-06-2014].

Dorling, M. & Walker, M. 2014a. *Computing Progression Pathways.* Available: http://community.computingatschool.org.uk/resources/1692 [Accessed 23-06-2014].

Dorling, M. & Walker, M. 2014b. *Computing Progression Pathways grouped by CS, IT and DL.* Available: http://community.computingatschool.org.uk/resources/1946 [Accessed 23-06-2014].

Dorling, M. & Walker, M. 2014c. *Computing Progression Pathways with Computational Thinking.* Available: http://community.computingatschool.org.uk/resources/2324. [Accessed 27-06-2014]

Nettleford, W. 2013. *Primary School Children Learn to Write Computer Code.* Available: http://www.bbc.co.uk/news/uk-england-london-23261504 [Accessed 23-06-2014].

Peyton-Jones, S. 2014. *Teaching Creative Computer Science.* Available: http://tedxexeter.com/2014/05/06/simon-peyton-jones-teaching-creative-computer-science [Accessed 23-06-2014].

Royal Society. 2012. *Shut down or restart? The way forwards for computing in UK schools.* Available: https://royalsociety.org/~/media/education/computing-in-schools/2012-01-12-computing-in-schools.pdf [Accessed 23-06-2014].

Selby, C., Dorling, M. & Woollard, J. 2014. *Evidence of Assessing Computational Thinking.* https://eprints.soton.ac.uk/366152 [Accessed 23-06-2014].

Selby, C. & Woollard, J. 2013. *Computational Thinking: The Developing Definition.* Available: http://eprints.soton.ac.uk/356481 [Accessed 23-06-2014].

Wing, J. 2006. *Computational Thinking.* Commun. ACM, 49, 3, 33-35. Available: http://dl.acm.org/citation.cfm?id=1118215 [Accessed 23-06-2014].

Wing, J. 2011. Research Notebook: Computational Thinking - What and Why? *The Link.* Pittsburgh, PA: Carneige Mellon. Available: http://www.cs.cmu.edu/link/research-notebook-computational-thinking-what-and-why [Accessed 23-06-2014].