Concrete Thoughts on Abstraction

Keiron Nicholson, Judith Good, Katy Howland
IDEAs Lab, Department of Informatics, University of Sussex
{K.Nicholson, J.Good, K.L.Howland}@sussex.ac.uk

Keywords: computational thinking, abstraction

Abstract

This paper considers the notion of abstraction and its relevance to the computational thinking agenda, including potential difficulties with its use, and possible solutions. We first discuss a paper by Blackwell, Church and Green (2008), "The Abstract is the Enemy", in which the authors use four examples of software design projects to outline the potential dangers of abstraction. The example projects all show how unhelpful abstractions can cause problems for users, and the authors argue that these problems result from attempts to mediate the human and technological worlds. Blackwell et al. (2008) then argue that the computational thinking drive may invite people to adopt a particular style of thinking and problem-solving; one which sees abstraction as 'a friend'. We argue that abstraction is neither friend nor enemy but simply a tool which can be helpful, or not, depending on the manner and context of its use. We propose that whilst the computational thinking agenda does in fact recognise this, it has not yet offered suggestions as to how people can be taught to use this tool effectively. We consider abstraction in terms of five key skills: considering the context in which the abstraction operates, defining and choosing the right abstraction, working with multiple layers of abstraction simultaneously, considering abstractions critically, and knowing when to use abstraction. Within these five skill groups, we consider fourteen sub-skills which we see as important to the successful use and teaching of abstraction. As such, they allow us to begin to design a 'curriculum for abstraction'.

1. Introduction

The computational thinking manifesto (Wing, 2006; 2008), has attracted a lot of interest as of late in the field of computer science. Broadly speaking, it suggests that teaching people how to "think like a computer scientist" is of benefit to all, and that this type of thinking has to date led to significant advances in almost every field of endeavour. Recently, Wing (2008) has defined computational thinking more specifically in terms of abstraction and automation:, namely, the ability to define, choose and use abstractions, to consider multiple levels of abstraction and their relationships, and to automate these abstractions.

In "The Abstract is 'an Enemy'", Blackwell, Church and Green examine a group of software projects in which the use of abstraction by the designers appears to have had a detrimental impact on the usability of the software. Abstraction being a fundamental aspect of computational thinking, they go on to question whether the computational thinking agenda might similarly have some negative consequences that have not been fully examined. The concerns of Blackwell et al. about the over-enthusiastic application of computational thinking

are understandable, but we argue that there is space within the concept of computational thinking for the understanding that abstraction is not always the answer. The agenda has the power to introduce the issue of when and how abstractions can be used to best effect. Abstraction can be problematic, as Blackwell et al. demonstrate very effectively, but the computational thinking drive does not have to exacerbate this problem. In fact, it has the potential to teach people to understand and avoid these pitfalls by explicitly highlighting the difficulty of using abstraction well.

To address the challenge of how to teach the effective use of abstraction, we have conducted a detailed analysis of the abstraction process. We begin with Wing's description of the essence of abstraction in computational thinking (Wing, 2008) and go on to consider the problematic examples given by Blackwell et al. Each of these examples is examined in some detail, and situations which are presented as failures of abstraction are largely re-characterised as failures to use abstraction well. From this analysis we draw a set of key skills and understandings necessary for the successful use of abstraction. Finally we outline a new approach to teaching the appropriate and effective use of these skills.

2. Abstraction as a universal tool

Abstraction is the process of representing a subject in such a way that only information relevant to one's purposes is retained. An abstraction is the result of this process: a proxy or summary which characterises, classifies and captures the essence of the subject with respect to a relevant aspect of interest.[1] Within computing in particular, abstraction is often characterised in terms of two key types; abstraction by parameterization and abstraction by specification (Liskov and Guttag, 2000; Abbott and Sun, 2008). The former entails software abstraction, where computational elements are represented in an abstract form and parameters are only entered on instantiation. This type of abstraction is important in creating elegant and generalisable programming solutions. Abstraction by specification takes place at the earlier stages of requirements gathering and design and involves simplifying real world complexity so it can be represented in a computer system.

Abstraction is a crucial tool for advancing human thought and communication, and has been so throughout our entire history. Abstraction through representation has allowed humans to develop cognitively to be the creatures they are today (Donald, 1991). It is unavoidable, and utterly necessary. To abstract is to be human. Amusingly, the abstract for the Blackwell et al. paper simply reads "An enemy." This is not merely a play on words. A paper summary is known as an abstract for a very good reason; summarisation is in essence abstraction. Similarly, classification and categorisation have abstraction at their heart. Among the examples listed by Blackwell et al. there is one is taken from Bowker and Star's work "Sorting things out"(Bowker and Star, 1999), the introductory chapter of which is entitled "To classify is human". In this work the authors argue that classification is a very human activity, which can have negative consequences but is inescapable. Ultimately, all methods of representation are abstraction by definition, as they seek to represent something in place of the thing itself. Abstraction is universal: there is no question of 'When should we use abstraction?', only, 'How can we use

[1] The authors would like to thank the benevolent anonymous reviewer who suggested a version of this definition.

abstraction well?' However, while abstraction may be universal, the process of examining it critically is not. The computational thinking drive has the power to change this for the better.

In a paper expanding on the idea of computational thinking, Wing identifies abstraction as "[t]he essence of computational thinking". Yet she is not unaware of its dangers, stating also that "[i]n working with rich abstractions, defining the 'right' abstraction is critical. The abstraction process—deciding what details we need to highlight and what details we can ignore—underlies computational thinking" (Wing, 2008, p.3718). To fail to give the process of abstraction the consideration it deserves can have negative consequences, as artfully demonstrated by Blackwell et al. who provoke debate by highlighting examples of thoughtless abstraction. Wing herself points out the dangers of "not thinking very hard about defining the right abstraction" (Wing, 2008, p.3719), saying that this is all too easy, but that developing computational thinking skills involves taking this process seriously.

In each of the examples given by Blackwell et al. the problem is not with abstraction per se, but with unhelpful abstractions that hide information which is relevant, or classify things incorrectly. In each case the unhelpful abstractions are made by humans, either individually or collectively as a department in an institution. The examples read more like an indictment of the institution which commissioned the projects than of the computer systems themselves. Blackwell et al. point out that a piece of software necessarily reflects the organisational structure in which it was produced, but argue that these abstractions can be made more rigid by virtue of being expressed in computational form. In each of these examples, the unhelpful abstractions could equally have been enacted by a non-computational system such as a paper form, though it is true that altering an abstraction is likely to be more difficult when it is fully integrated into a software system. Nevertheless, a rigid abstraction is not such a problem if it was correctly defined to begin with. Learning how to apply abstraction effectively is an advanced skill. Blackwell recounts how he almost certainly used generic names for functions and methods as a young programmer, and that he later (in the course of his career as a computer scientist) learnt to avoid this.

We would argue that humans always have and always will make use of abstraction, but that they can be taught to use it more effectively. Blackwell et al. present convincing examples of how humans thinking computationally have badly misused the skill of abstraction, and it is reasonable to assume that such mistakes could occur more frequently with the spread of computational thinking. What we should take from these examples is that even trained computer scientists are liable to choose the wrong abstraction, to abstract away the wrong information, to get confused about which abstraction they are working with, and in general, to get it wrong. This does not mean we should not teach abstraction, only that we should teach it well. Its failures must be made explicit so that they can be recognised and avoided. In the following section we examine abstraction skills in more detail, with reference to computational thinking and the examples of misuse of abstraction presented by Blackwell et al.

3. Key Abstraction Skills

In examining abstraction, and looking at the ways in which it can be taught, we have defined the process of abstraction in terms of five key skills: 1) identifying the contexts of use, 2) defining the abstraction based on those contexts, 3) working with multiple layers of abstraction, 4) examining abstractions critically, and 5) knowing when to use abstraction at all. In turn, for each

of the skills, we have defined a number of subskills which provide a further level of refinement and detail to the high level definition of the skills. Taken together, these skills and subskills could form the basis of a 'curriculum of abstraction'.

3.1 Consider an abstraction in relation to the contexts in which it operates

It is crucial that when creating an abstraction, the contexts in which it will operate are first thoroughly considered. Any abstraction strips away much of the information available about its subject, and this can only be successful if we know that no key information has been lost. It is therefore imperative to know exactly what the key information is, and to know this we must know exactly how the abstraction is being used. Many of the examples provided by Blackwell et al. can be blamed on either a failure to identify the contexts in which an abstraction will operate, or a failure to ensure that the abstraction is assessed for suitability against each one of these contexts.

The 'task at hand', for which the abstraction has been defined, is the most obvious context to consider, though it is not always considered well. The example of CamRIS being designed to incorporate costing and accounting processes for the management of research, whilst failing to consider the realities of the environment in which the research is actually conducted, shows a clear case of a key context of use not being fully considered. There were existing processes in place for carrying out these tasks, yet the abstractions chosen caused important functionalities relating to these processes to be scattered throughout the system. The abstraction which was imposed upon the users reflected the institutional context, and was based on the needs of one politically important group whose perspective became adopted as the one 'correct' way of conducting the research process, but was a 'misfit' with user abstractions which were already in use. It is important that information about existing abstractions be gathered to avoid such cases where these abstractions are ignored to the detriment of the usability of the system. This gathering and evaluation of existing abstractions must be carried out at an early stage in the design process to avoid situations such as the case recounted where user defined abstractions could not be adopted as they conflicted with abstractions already defined by system developers. There may also be cases where existing abstractions are deliberately replaced by new ones, such as when a company undertakes business process reengineering. The same risk exists of alienating those people who continue to think in terms of the existing abstractions.

Problems also arise when contexts beyond the 'task at hand' are not considered. For example, if an abstraction is exposed to the end-user, it must be considered to have a new and separate set of requirements to meet, that of acceptability within social and political contexts. A computational representation of a user which is also accessible to the user has to represent the user in a way which both meets the needs of the software and is acceptable to the user themselves. Ignoring the social ramifications of asking office workers to lock their workstations (and potentially cause offence to their office-mates), or the possible offence caused to users by asking them to select their ethnic origin from one of three broad groupings are examples of this given by Blackwell et al.

An abstraction may acquire new contexts if it is re-used for another task (for example, if developers add new features to their software but use the same abstract representations of relevant data). It is important to be conscious of an abstraction crossing over into a new context, and judge its suitability afresh. It may be appropriate for a particular task, but not for another – key information may have been abstracted away. Or, it may have become public for the first time, and therefore the question of how the user reacts to it becomes relevant.

More generally, Blackwell et al. are concerned that "designers of the abstractions think that they somehow capture the "essence" of the problem"(Blackwell et al., 2008, p. 6) while stripping

away crucial subtleties. If we accept that we must sometimes create abstractions, in which most of the information about a subject is necessarily lost, we must attempt to ensure that we are not careless about the contexts into which we deploy them. By identifying the contexts in which an abstraction will operate, and carefully considering the information that will be relevant to each of those contexts, we can do our best to ensure that such subtleties will be considered when they need to be.

A series of subskills which relate specifically to contexts of use can now be defined:

(a) Clearly identify each and every context in which the abstraction will be used or considered. Any task that the abstraction will help to accomplish is one context of its use. In particular, understand that if an abstraction is exposed to the end-user then it must be considered to be operating within social and political contexts and judged for its suitability within them.

(b) When choosing an abstraction, be aware of abstractions which are already used within the relevant context, particularly in terms of the way people who operate within that context tend to work or think. While there may be benefits to imposing a new abstraction, a 'misfit' between an abstraction already in use and one that is artificially imposed can have negative consequences. Consider whether an existing abstraction may be the correct one to use.

(c) When choosing to re-use an existing abstraction, carefully evaluate whether or not it is still suitable for the new context. Remember that it has been defined for a different purpose and that important aspects of the subject may have been abstracted away or simplified in a way that is not appropriate for the new context. Be particularly careful not to allow abstractions to cross into new contexts by accident or default.

3.2 Define and choose the right abstraction

When defining an abstraction it is particularly important to choose the right level of description. This is a key skill in software design, and one which is widely recognised as important for novice programmers in developing an understanding of the process. Ensuring that the abstraction chosen is neither too specific nor too generic is something which is highlighted as important in a wide range of computer science text books (see for example (Wirfs-Brock and McKean 2002) and (Hoberman 2002)).

A common source of failure is the use of abstractions which are too generic to be useful for the task at hand, as exemplified by Blackwell with reference to the module name "ProcessData". Again, context is crucial. There are valid reasons to consider a module as something which simply 'processes data', just as an object-oriented programmer will sometimes find it useful to treat a variety of complex software components as simply 'objects with states and behaviours'. The key is whether that level of abstraction is useful for the task at hand. We name modules to uniquely identify their purpose, and "ProcessData" is not a good way to uniquely identify a module. By the same token, novice programmers often make the opposite mistake, defining software components which are unnecessarily specific and denying themselves the opportunity to re-use them.

Overly generic abstractions can be dangerous if information that has been abstracted away later proves to be important. Rather than rejecting the abstraction as unsuitable, the developer may sense the need for the missing information and 'fill in the blanks' with their own assumptions, consciously or otherwise. Blackwell et al. note that it is "far easier to assume that a generic user thinks about the world the way a developer does, than it is to say that 'Fred from upstairs' does"(Blackwell et al., 2008, p. 4), and they reference a situation in which the generic user was inappropriately assumed to have great technical prowess. Making such an assumption is to take

an abstraction of the user which does not represent technical ability, and apply it to a context which requires technical ability to be considered, leading to costly mistakes.

A further risk is that by abstracting away aspects of a subject you may provoke unintended consequences, as demonstrated by the example in which ceasing to classify diseases on a geographical basis led to the loss of implicit travel information from patients' medical records. The fact that this information was derived from the abstraction, rather than actually captured in the abstraction itself, understandably led to its eventual loss; if the information was important, it should have been explicitly represented in the abstraction. This is a somewhat ambiguous example, as the presence of implicit information could as easily be damaging as helpful. Nevertheless, the abstraction was changed without full consideration of the consequences.

Given the generic skill of defining and choosing the appropriate abstraction, we can further refine it into the following subskills.

(a) Avoid choosing an abstraction which is too specific for the task at hand, and unnecessarily excludes subjects which could have benefited from inclusion.

(b) Avoid choosing an abstraction which is too generic for the task at hand, and as a result is too inclusive to be useful. In particular, be aware of a harmful tendency to substitute your own assumptions for information missing from the abstraction but relevant to the task.

(c) Understand that simplifying or excluding aspects of a subject may have consequences that are not immediately clear.

3.3 Work with multiple layers of abstraction simultaneously

Computer scientists will typically deal with representations of the same subject at multiple layers of abstraction simultaneously - for example, working with a class hierarchy in which an object is an Eagle, a Bird and an Animal all at once. Wing (2008) has noted that this process of "working with multiple layers of abstraction and understanding the relationships among the different layers" is key to the abstraction process. Several of the failures of abstraction Blackwell et al. present can be attributed to confusion about which layer of abstraction is currently being dealt with, as well as confusion between the abstraction and the subject itself.

The case of CamSIS requiring undergraduates who were also PhD applicants to use separate logins is illustrative. While PhD applicants and undergraduates may have very distinct sets of needs, the need to log into CamSIS is one that they share. As such, it is useful to define two layers of abstraction, dealing with them as a single abstract 'user' when they need to log in, and dealing with them as distinct PhD applicant and undergraduate users when providing functionality specific to those roles. Dealing with a PhD applicant when the task at hand relates to a 'user' is a failure to work at the right layer of abstraction, as is treating someone as a 'user' when the task at hand relates to a PhD student, an example within the same application being the use of a menu system which has the same structure for all users despite all the functionality available to PhD students being buried several layers deep.

From the skills of simultaneously using multiple layers of abstraction, we can derive the following subskills:

(a) When operating in multiple contexts you may have to deal with representations of the same subject at multiple layers of abstraction. Again, clearly identify the contexts and the aspects of the subject that are relevant to them, and from this derive an abstraction for each context.

(b) When working with multiple layers of abstraction, understand how the different layers of abstraction relate to each other, and always be clear as to which of the layers you are currently dealing with.

(c) Always be clear as to whether you are dealing with the subject itself, or an abstraction of that subject, particularly when they use the same name.

3.4 Consider abstractions critically

No matter how carefully an abstraction has been defined, it must always be considered critically. It is important to be aware that any representation which is encountered has always been abstracted in some way, and that abstractions are never the same as the thing they represent; something is always hidden or lost as a result of the abstraction process. Abstractions must not be treated as though they are the things they represent. They are simplifications, and a failure to recognise this can cause problems. Abstraction can influence the way in which the subject is reasoned about in ways that may not be obvious, and so when working with a given abstraction it is important to consider who has created that abstract representation and for what purpose.

Additionally, there can be no guarantee when working with an abstraction, that the details of the subject which have been abstracted away will not re-emerge in some form during your attempts to use that abstraction. One example is that of paper currency: although paper bank notes are taken to represent monetary value, the abstraction breaks down if you burn one of your bank notes, as it isn't possible to actually destroy monetary value itself.

Whilst Blackwell et al. admirably demonstrate the principle of considering abstractions critically, they themselves have created abstractions to aid their argument which must be considered critically. Their argument is based on a representation of abstraction which is negative, with carefully chosen examples which are themselves abstracted descriptions of real-world occurrences. The examples represent the projects under consideration in a specific light, whilst there will be extensive complexity and detail which has been left out because it did not help advance the argument presented.

From generic skill of considering abstractions critically, we can derive the following subskills:

(a) Understand that any representation you encounter may have been abstracted in a particular way, and that this can have consequences for how the subject of the abstraction is understood. Try to identify what information has been simplified, removed or made prominent, and consider the abstractor's intention in making these choices.

(b) Understand that although an abstraction may have simplified or hidden a particular aspect of the subject, it cannot be guaranteed that these abstracted details will not re-emerge in a way that requires that they be considered.

3.5 Know when to use abstraction

We have generally characterised the examples given as being due to incorrect usage of abstraction, but in some cases it is clear that it was not appropriate to use abstraction at all, or rather, that abstraction was used for the wrong reasons. Equally, there is at least one case presented in which a chance to usefully abstract was missed.

To choose an abstraction which is too generic for the task at hand is a failure we have already discussed, but far more insidious is the failure to actually recognise it as a mistake. Naming a module "ProcessData" is not useful, but it is accurate – the unreflective programmer can feel

satisfied that they have described it correctly, even while being aware, on some level, that the problem has simply been deferred. By giving vague instructions we can avoid responsibility when the process goes wrong; by making generic statements we can appear to be correct simply by covering all bases. Learners should be taught to recognise when they are using abstraction as a crutch. Similarly, Blackwell et al. note that developers are often guilty of 'goal conflation', in which an abstraction used to build a solution to a problem is ultimately used to judge its success, inappropriately equating 'the proposed solution was implemented' with 'the problem was solved'. (Note, however, that when judging success in the real world, whether by logging the time taken to complete activities, gathering feedback from users or tracking sales of the software, we are still using abstractions to hone in on some metric of success.)

"The Abstract is 'an Enemy'" includes a single example in which it is clear that an opportunity to use abstraction was missed: the failure of CamSIS developers to recognise that complaints from individual students could be generalised, revealing a clash between the abstractions that students used to reason about the interface and those that the developers had used to implement it.

In terms of the broader skill of knowing when abstractions are appropriate, the following subskills can be derived:

(a) Avoid using abstraction simply in order to abdicate responsibility and avoid making a decision. Learn to both recognise and resist this tendency.

(b) Understand that a problem existing in the real world must ultimately have the success of its solution judged in the real world, not on the abstract level used to define the solution.

(c) Recognise that when several different subjects share some similarity, it can be an opportunity to create a generalised abstraction, drawing out that which is the same about them in order to aid your thinking.

4. Conclusion

From the arguments made above, it seems clear that some of the problems associated with abstractions have less to do with the use of abstractions than with their misuse. Similarly, one could argue that, in some cases, the problems are less to do with the abstractions themselves than with the way in which they collide rather infelicitously with a broader, real-world context. Taken together, this in turn begs the question of how to teach students to use abstractions appropriately, and understand the contextual implications of a given abstraction. Calls for a curriculum of abstraction have been increasing and whilst there has been some progress towards measures for assessing abstraction skills (Kramer, 2007; Hill, Houle, Merritt and Stix, 2008), there has been little development of methods for teaching these skills.

In some sense, a delineation of the subskills involved in working with abstractions goes some way toward addressing these issues, as it provides tractable points which can be addressed more concisely through teaching, in contrast to a generic goal of "teaching abstraction". At the same time, simply being aware of the subskills to be taught does not in itself suggest the most effective learning activity for helping students to master them.

We are currently considering an approach to abstraction which focuses on three issues:

- Immediacy of feedback

- Clarity of feedback
- The bidirectional relationship between abstraction and consequences

Immediacy of feedback relates to being able to see the consequences of abstraction choice. In many of the situations described by Blackwell et al., the consequences of the misuse of abstractions did not occur until well after the event of defining the abstractions. By attempting to shorten the feedback loop, and make the consequences of a choice of abstraction more immediately obvious, we hope foster students' awareness of the need to consider consequences from the outset, rather than wait and see what happens when the abstraction is deployed.

Additionally, the abstractions discussed by Blackwell et al. were often embedded in a complex system which may have made it difficult to disentangle the faulty use of the abstraction from other behaviours of the system as a whole. We plan to create simple scenarios in which students can experiment with abstractions and see the relationship between abstraction and consequences more directly, rather than mediated by other features of the software.

Finally, a useful skill in working with abstraction involves not only the ability to choose an appropriate abstraction and use it in a way which is concordant with its broader context, but to be able to look at existing software solutions, determine whether they are problematic, and understand how abstractions may have played a role. Again, we plan to use simple scenarios which involve both creating abstractions and forward reasoning about their consequences, and investigating existing situations in which abstractions have been used as a way of reasoning backwards about the faulty consequences of these abstractions. In so doing, we hope to foster the comprehension and manipulation of abstractions as a bi-directional skill.

In conclusion, we have argued that, although using abstractions well is a difficult task, and all software projects are likely to harbour examples of its misuse, it is at the same time an integral part of human activity, and it is therefore in our interest to learn the skills associated with the use of abstraction, and to be aware of the broader contextual factors that relate to putting them into practice. By putting the focus on abstraction, the computational thinking agenda has the opportunity to provide support for this activity and, hopefully, to support a more holistic view of the teaching of abstraction from an early stage.

References

Abbott, R. and C. Sun (2008). Abstraction abstracted. *Proceedings of the 2nd International Workshop on The role of Abstraction in Software Engineering*, Leipzig, Germany, ACM New York, NY, USA.

Blackwell, A., L. Church, et al. (2008). The abstract is 'an enemy': Alternative perspectives to computational thinking. *20th Annual Workshop of the Psychology of Programming Interest Group (PPIG 08)*.

Bowker, G. and S. Star (1999). *Sorting things out*, MIT Press Cambridge, Mass.

Donald, M. (1991). *Origins of the modern mind: Three stages in the evolution of culture and cognition*, Harvard University Press.

Hill, J., B. Houle, et al. (2008). Applying abstraction to master complexity. *Proceedings of the 2nd international workshop on The role of abstraction in software engineering*, Leipzig, Germany ACM New York, NY, USA.

Kramer, J. (2007). "Is abstraction the key to computing?" *Communications of the ACM* 50(4): 36-42.

Liskov, B. and J. Guttag (2000). *Program development in JAVA: abstraction, specification, and object-oriented design*, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

Wing, J. (2008). "Computational thinking and thinking about computing." *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 366(1881): 3717-3725.