How Teaching Programming Across Disciplines Can Instil Systems Thinking

Leo Riviera

Voxa
leo.riviera@voxa.dev

Maeve Murphy Quinlan

Research Computing, University of Leeds
m.murphyquinlan@leeds.ac.uk

Rebecca L. Colquhoun

Department of Earth Science and Engineering, Imperial College London

Devanjan Bhattacharya

WARSHARE, School of Social and Political Science, University of Edinburgh; MagIC, Nova Information Management School, Universidade Nova de Lisboa
d.bhattacharya@ed.ac.uk

Introduction

The purpose of university has been the subject of much discussion (Gupta 2021). Some see it as a focal point for the humanist tradition in the Renaissance fashion, where one goes to engage with and anchor themselves in the great work of man. Others see it as a site to acquire knowledge, skills, values, beliefs, and commitments; young people develop their mental faculties and define their world views, becoming more ready to integrate into society (McArthur 2011).

Some see it as a government-mandated institution crucial to the smooth function of certain specialist fields. A university ensures, through rigorous academic examination, that the doctor treating your illness, the solicitor defending your case, and the architect designing your home are all sufficiently competent (Ryan, Toohey, and Hughes 1996).

Universities function as hotbeds of cutting-edge research, funded by subsidies from the public purse and investment from industry. Given the market-oriented university system of the UK and its ruthless focus on “value for money” (Wilkinson and Wilkinson 2020; McArthur 2011), some posit the core focus of higher education institutions to be to produce findings with some practical application. We, of course, cannot forget about the university as a modifier of social status.

However, it is impossible to deny that one of the core functions of a university is to educate. Universities provide their students with the knowledge necessary to pursue their chosen career paths, and provide a space for individuals to develop and hone their chosen skill. When viewed in this light, the value of teaching programming across disciplines becomes apparent.

Teaching Computational Thinking Through Programming

We have long foreseen a world where computers are ubiquitous; today, they are undeniably present in almost every aspect of modern life (Weiser 1999; Stewart 2012). Over the past few decades, they have become fundamental to the way we communicate, work, analyze data, and solve complex problems. Understanding how to use a computer is no longer a skill reserved for computer scientists, but serves as an invaluable tool for problem-solving across all disciplines. In fact, the skill is not only crucial to being a good worker, but also a good citizen who can affect change. With some of the opinion that computer literacy in decline among young people (Stewart 2012), teaching programming across disciplines can provide students with the base knowledge they need to understand the computers they come across during their careers.

Learning to code is associated with improved problem solving skills in some domains (Scherer, Siddiq, and Sánchez-Scherer 2021). When we teach students to program, we often start by introducing them to a language’s syntax. However, we also introduce them to resources for self-study, like online tutorials. We teach how to find useful resources and evaluate their quality. In their self-study, students discover reference documentation and language specifications. That, in turn, supports their discovery of capabilities of the programming language they were never formally taught. Students eventually learn not only how to write code but also how to teach themselves.

In learning to code, students encounter computational thinking (Jeannette M. Wing 2006). Computational thinking involves taking a problem and developing solutions we can use a computer to execute. Although different models of computational thinking consist of different steps, Shute, Sun,

and Asbell-Clarke (2017) describes it as a process which includes

- Problem reformulation, where we reframe a problem into a solvable and familiar one
- Recursion, where we incrementally construct a solution based on preceding information
- Decomposition, where we break a large problem down into more manageable sub-problems
- Abstraction, where we model the core aspects of a complex problem or system
- Systematic testing, where we take purposeful steps to derive solutions

Computational thinking has applications beyond computing. It is seen as fundamental for analysing and solving all sorts of problems, because we can transfer techniques we use to make our solutions efficient for computers to other disciplines[47] (Jeannette M. Wing 2010, 2012). Programming is a tool to teach computational thinking, but computational thinking can be applied to other problems. Even if one disagrees with the premise that computational thinking provides skills which are directly transferrable to other disciplines, learning to code is a good form of mental exercise and teaches us to solve problems (Love 2019; Scherer, Siddiq, and Sánchez-Scherer 2021).

However, teaching programming outside of computer science courses may have another, less obvious benefit: instilling systems thinking skills in students. To understand why this could have value, we need to take a brief foray into the world of systems.

A Leap Into Systems Thinking

A system is a collection of entities which are connected to each other and work together for some purpose. As a result of interactions between entities, the system produces its own pattern of behaviour over time. The pattern of behaviour that emerges from a system can be quite different from the behaviour of its constituent parts (Kim 1999, 2–5). The process of identifying, understanding, decomposing and rebuilding systems is called systems thinking. More formally, systems thinking can be defined as "…a set of synergistic analytic skills used to improve the capability of identifying and understanding systems, predicting their behaviors, and devising modifications to them in order to produce desired effects"[48] (Arnold and Wade 2015). Systems thinking is, at its core, a tool for allowing us to see the world differently, ask different questions and get different (often unexpected, sometimes better) answers.

What we colloquially refer to as a computer today was at one point called a computer system. The computer was the hardware used to perform calculations, while the computer system was the combination of hardware and software that could be programmed to perform a specific task ("An Introduction to Computers and Computer Systems: Session 2: 6" 2021). As devices have become more capable and more compact, this differentiation has become a point of pedantry. However, for the purposes of our discussion, this distinction is helpful. The physical components of a computer (its CPU, memory, storage drives, peripherals and the like), are unable to achieve anything by themselves. Similarly, software is useless without a device to run it on. When both are combined, we have a useful system which can take in data, process it into useful information, and output it in some usable form. We have a computer system!

By our definition of a system, computer programs are also systems. They are a collection of elements (most obviously input, source code, and stored data) connected together (in the body of a program and in a machine's memory)[49] for the purpose of producing outputs over time. Let us explore this further.

Computer Programs as Systems

The obvious elements of a simple computer program include, among other things,

- its inputs,
- its outputs,
- its application code,
- the internal state it maintains and any stored data.

Computer programs also feature many intangible elements, including

- accessibility for users of assistive technologies and users with disabilities,
- its availability and uptime,
- its maintainability, and the ease with which changes can be made,
- its correctness, where a given input produces an expected output. This can be assured with manual and

[47] The point we're making here is quite nuanced. Analysis by Scherer, Siddiq, and Sánchez-Scherer (2021) establishes that learning to code may be associated with *some* broader cognitive benefit, with any possible transfer of learning coming with an explicit cost in the form of sufficient teaching time, supportive educational environments, and teacher training. However, this chapter posits that computational thinking, often taught through computer programming, provides a cognitive framework that makes solving problems in other disciplines easier.

[48] Interestingly, Arnold and Wade (2015) take a systems thinking approach to defining systems thinking. How meta!

[49] The theoretical Turing machine springs to mind, here. Despite its simplicity, it forms the basis for how many computer scientists think about computation today.

automated system testing, and in some cases, can be formally verified,

- its stability, in its ability to handle large volumes of input and remain stable if those limits are exceeded,
- the aesthetics of its source code, often enforced by code review and code linters.

These elements are inherently interconnected and dependent on each other. The application code transforms the input, considering and possibly updating its internal state in the process. Data can be stored for retrieval at another time, and can form a state which affects future outputs of the same program. The application code directly affects the correctness and stability of the program. A computer program produces output to fulfil its purpose, and that output can even be the input of another computer program. The general purpose of computer programs is to compute, to process some form of input into some form of output. The purpose of a particular piece of software can be deduced by how it processes data, looking at its inputs, outputs, and side effects. This makes a computer program a system which, in turn, runs on a computer system. It's a system within a system!

Not all elements matter equally when designing a program. It depends on the purpose. Scripts for calculating the answers to mathematical problems on Project Euler[50][51] would emphasise correctness. However, a Government website designed to provide information to the public would emphasise accessibility and availability (Government Digital Service 2012). An online multiplayer video game would likely focus on maintainability to roll out changes quickly, and stability to handle spikes in traffic from players. A computer program written to control a nuclear power plant looks very different from one used to track library books!

Like any other system, a software program's response to input is characteristic of itself[52]. It depends on the way we have encoded instructions to process input and produce output. This is even more true for a program which holds state or uses randomness, given the same inputs may produce drastically different results based on some internal state. More complex programs can even change and adapt their behaviour in response to their environment and their input. For example, a rate-limiting system can restrict requests from a user to prevent overload. Programs can also be fault-tolerant, using circuit breakers to avoid repeatedly trying to run an operation which fails. These can be considered balancing feedback loops, intended to prevent failure and keep performance metrics within some desired threshold.

In the real world, it is more likely that a computer program that is actually useful is doing more than one thing. Like many complex systems, computer programs can be composed of subsystems. Although most of a program's behaviours are explicitly designed and defined in application code, many arise as a logical consequence of different elements interacting with one another. This emergent behaviour is another key characteristic of systems (Meadows, n.d., 11–34).

Thinking one level up from a program, computer systems are made up of several programs that are constantly interacting with each other; a program itself forms a subsystem within a computer system. The Unix kernel schedules time for programs, background processes and system operations, each of which affects its environment and is affected by its environment (Alomari, n.d.). A program can affect its execution environment (for example, by storing data on disk or holding it in memory), and the environment can affect the program (by interrupting its normal execution). Feedback loops like these are common in systems found in the real world.

As people interact with the world, they intuitively build up an understanding of and appreciation for complex systems (Forrester 1971). Teaching students programming merely provides them with the language to describe and interact with complex systems. This knowledge can then be used to understand other systems which make up the world. Of course, there are several concepts in systems thinking which have no obvious analogies when discussing computer programs. However, the way we build, use and debug computer programs provides a useful introduction to systems thinking.

Much like any other system, a computer program's purpose is what it does, regardless of the creator's intent (Meadows, n.d.). Depending on how a program behaves within and interacts with its execution environment, interesting (and sometimes dangerous) side effects may occur. To find and fix unexpected behaviours in our computer programs, we carefully examine the individual components and how they work together, in the context of the desired function. Similarly, complex systems have structures which contain latent behaviours. In fact, a big challenge of systems thinking is to arrange the structures and conditions of a system to encourage and amplify beneficial behaviours, and limit and reduce destructive behaviours (Meadows, n.d., 72).

With a computer program, we may have access to logging, tracing, and observability tools which help us diagnose deviant behaviours. Eventually, we identify the element or interconnect which has caused our unexpected behaviours (colloquially known as a "bug"), and we make an adjustment to remediate it. The methodical troubleshooting approach can be applied to the systems which

[50] https://projecteuler.net/

[51] Project Euler is a collection of 981 (at time of writing) mathematical problems designed to be solved with computer programs. It allows users to discover and discuss solutions to these problems, and is quite a popular resource for learning programming.

[52] Deterministic programs always produce the same output for a given input. Non-deterministic programs can produce different outputs for the same input, depending on factors such as internal state, randomness, or concurrency. Regardless of whether a program is deterministic or non-deterministic, its behaviour can be understood and predicted based on its design and implementation.

govern the behaviour of friend groups, companies, charities, universities, market economies, and other groups of interconnected entities.

Decomposing Conceptual Systems With Computational Thinking

We can decompose the systems which exist in our world using the same process we use to construct computer programs: computational thinking. As mentioned previously, the end result of the computational thinking process is algorithms that can solve a problem. Along with other components, the algorithms we define and the relationships between them form a system we call a computer program. Perhaps performing the steps of computational thinking in reverse could be useful for decomposing a system into its constituent parts?

In his *Introduction to Systems Thinking*, Kim (1999, 17–18) details the different modes we can act in when dealing with a system. Thinking about these modes, particularly within the context of computational thinking, allows us to transcend from being acted on by a system to acting on a system.

Our interactions with systems are normally restricted to reacting to what it does. After observing enough of its behaviour, we can notice patterns in those events we can adapt to. A system's architecture, also referred to as a "systemic structure", is the way a system's components are organised. It is responsible for the events and patterns we observe (Kim 1999).

When examining a system's architecture, we can start with systematic testing from computational thinking. We can ask ourselves: what steps could someone have taken to determine the elements that make up this system, and how they are connected to each other? Where could they have searched? Who could they have asked for a review?

Next, considering its stated purpose, we can de-abstract the system. If we were designing a system to achieve a similar goal, what detail would I consider irrelevant? Putting myself in the shoes of the people who have built and maintained this system, what are their incentives? What are this system's intended goals, versus its real impact? Is there any detail about the real world that, when abstracted, would cause the system to act in the way it does? Is there a way to improve this?

Considering the system as a whole, we can recursively recompose it. Looking at its purpose and emergent behaviour, what individual components make up the system? How could those be linked to one another? Based on the model I've constructed, what about this system could be broken? What about it could be inefficient? What about its approach to problem solving is inelegant? Does it have any reason to be like that?

Finally, we can ask ourselves: what is this system trying to do, and what was the intent? What output does it produce? What side effects can be directly attributed to it? How does its output affect other systems? What adjacent problems exist that this could be tweaked to solve? What is the real purpose of this system?

A system is a useful abstraction for thinking about behaviour which emerges from the interactions between different components. However, as it is an abstraction, the systems we concoct using this method will never be complete. There will always be context we lack and so we will omit components and connections between them. However, interrogating a system in this way could prove useful depending on the problem we're trying to solve.

When picking apart a system in this way, we look at things it does and look at trends in the things it does. Eventually, we can consider the way the system and its structures are organised to produce patterns which then produce outcomes. We can then consider what components should make up a system, and how they ought to be organised. This is similar to how we're able to make changes to a program's implementation to improve it or change its behaviour once we understand what it does and how it does it.

Computer programs differ greatly from systems found in the real world, because we can change and understand computer programs comparatively easily. They have defined execution environments and can be disabled if a feedback loop causes undesirable consequences. However, despite the fact computer programs are relatively simple systems, they are systems nevertheless. This means computational thinking is useful to identify systems, and develop our capacity to be more creative, reflective and generative when dealing with systems.

Rebuilding the World in Our Image

During the process of designing systems, we can consider our mental model of the world. Reflecting on how systems are designed, as well as how we would design them, exposes the assumptions we harbour about how the world works. We can then imagine new ways of organising a system's components. Understanding and changing the way we model the world in our heads allows us to usher into existence entirely new systems. Systems thinking allows us to constantly question how the world works, and what is actually important. From there, it is not a large leap to think about the world we want[53].

Our thinking is bounded by our ability to describe the world. If we lack the

[53] There are interesting parallels to be drawn with Derrida's deconstruction, but this is left as an exercise to the reader.

language to discuss change, we will struggle to conceptualise it. Put more succinctly by Ludwig Wittgenstein, the limits of one's language are the limits of their world (Wittgenstein, n.d.). Teaching programming instils computational thinking, as a stepping stone to systems thinking. With systems thinking, we have a language to describe the way the world works and how we can change it. In fact, the two work in tandem. Computational thinking provides us the ability to solve complex problems by breaking them down into smaller parts; systems thinking gives us a way to model how those smaller parts fit together and interact with one another.

What we teach speaks to the kind of people we want in the world. The type of work we do influences the type of people we are. We grow to fit the jobs we work based on the skills we use every day (Woods et al. 2020). We should encourage our students to develop systems thinking skills through programming, and to use those skills as part of their problem-solving toolkit in the workplace. Equipping people with the ability to identify, decompose and eventually restructure the systems which govern our world will make them more effective, more independent, and more capable agents of change. They will finally see the world for what it is: a collection of elements to be examined, interrogated, and reconstructed in the image of something more equitable and more sustainable.

Acknowledgements

Thank you to everyone who has offered feedback on this chapter, including Dr. Mark Stonehouse, Dr. Heather Allansdottir, Ole Thal, and other entities who wish to remain anonymous. Your input has been invaluable.

References

Alomari, Ahmed. n.d. *Oracle 8i and UNIX Performance Tuning*. The Prentice Hall PTR Oracle Series. Prentice Hall.

"An Introduction to Computers and Computer Systems: Session 2: 6." 2021. The Open University. https://www.open.edu/openlearn/mod/oucontent/view.php?id=109019&section=6.

Arnold, Ross D., and Jon P. Wade. 2015. "A Definition of Systems Thinking: A Systems Approach." *Procedia Computer Science* 44: 669–78. https://doi.org/10.1016/j.procs.2015.03.050.

Forrester, Jay W. 1971. "Counterintuitive Behavior of Social Systems." *Technological Forecasting and Social Change* 3: 1–22. https://doi.org/10.1016/s0040-1625(71)80001-x.

Government Digital Service. 2012. "Government Design Principles." GOV.UK; https://www.gov.uk/guidance/government-design-principles#this-is-for-everyone.

Gupta, Achala. 2021. "What's the Purpose of University? Your Answer May Depend on How Much It Costs You - LSE Impact." London School of Economics; Political Science. https://blogs.lse.ac.uk/impactofsocialsciences/2021/01/21/whats-the-purpose-of-university-your-answer-may-depend-on-how-much-it-costs-you/.

Kim, Daniel H. 1999. *Introduction to Systems Thinking*. Pegasus Communications.

Love, David. 2019. "Computational Thinking – a New Modality of Thought or Just What Coders Do?" https://digiteacher.wordpress.com/2019/09/20/computational-thinking-a-new-modality-of-thought-or-just-what-coders-do/.

McArthur, Jan. 2011. "Reconsidering the Social and Economic Purposes of Higher Education." *Higher Education*

Research & Development 30 (6): 737–49. https://doi.org/10.1080/07294360.2010.539596.

Meadows, Donella H. n.d. *Thinking in Systems*. Chelsea Green Publishing.

Ryan, Greg, Susan Toohey, and Chris Hughes. 1996. “The Purpose, Value and Structure of the Practicum in Higher Education: A Literature Review.” *Higher Education* 31 (3): 355–77. https://doi.org/10.1007/bf00128437.

Scherer, Ronny, Fazilat Siddiq, and Bárbara Sánchez-Scherer. 2021. “Some Evidence on the Cognitive Benefits of Learning to Code.” *Frontiers in Psychology* 12. https://doi.org/10.3389/fpsyg.2021.559424.

Shute, Valerie J., Chen Sun, and Jodi Asbell-Clarke. 2017. “Demystifying Computational Thinking.” *Educational Research Review* 22: 142–58. https://doi.org/10.1016/j.edurev.2017.09.003.

Stewart, Tom. 2012. “Computers Are Everywhere.” *Behaviour & Information Technology* 31 (4). https://doi.org/10.1080/0144929x.2012.675162.

Weiser, Mark. 1999. “The Computer for the 21st Century.” *ACM SIGMOBILE Mobile Computing and Communications Review* 3 (3): 3–11. https://doi.org/10.1145/329124.329126.

Wilkinson, L. C., and M. D. Wilkinson. 2020. “Value for Money and the Commodification of Higher Education: Front-Line Narratives.” *Teaching in Higher Education* 28 (2). https://doi.org/10.1080/13562517.2020.1819226.

Wing, Jeannette M. 2006. “Computational Thinking.” *Communications of the ACM* 49 (3): 33–35.

Wing, Jeannette M. 2010. “Computational Thinking: What and Why?” https://www.cs.cmu.edu/~CompThink/resources/TheLinkWing.pdf.

———. 2012. “Computational Thinking.” https://www.microsoft.com/en-us/research/wp-content/uploads/2012/08/Jeannette_Wing.pdf.

Wittgenstein, Ludwig. n.d. *Tractatus Logico-Philosophicus*. 2nd ed. Routledge Classics. Routledge.

Woods, Stephen A., Grant W. Edmonds, Sarah E. Hampson, and Filip Lievens. 2020. “How Our Work Influences Who We Are: Testing a Theory of Vocational and Personality Development over Fifty Years.” *Journal of Research in Personality* 85: 103930. https://doi.org/10.1016/j.jrp.2020.103930.