Building Software is Not[1] a Craft

Antranig Basman
amb26@ponder.org.uk

Abstract

Software construction, yet to establish its role in the world, inhabits an invidious position between mathematics, engineering and craft — losing out by the comparison to each. I draw a distinction between different levels at which the term 'craft' could be applied and list four key virtues desirable for crafted items at a higher level. The current products of computer scientists or software engineers are deficient in these four virtues, but not necessarily so. I propose a shift in the value system underlying our work — recognising that the function of material is to bring communities into contact. I will argue that software is material to the extent it promotes an ecology of artefacts which cooperate harmoniously in the lives of their owners. Rather than the workmanship of certainty, we should promote the workmanship of risk, turn from correctness to truthfulness, and consider time horizons of generations rather than at most a couple of years.

1. Introduction

This paper will take a different course in assessing the nature of software as craft, materials and media than is traditional in academia. A common setup would have us note some grounds for analogy — for example, that constructing software requires some skill, is error-prone, and requires labour — just like a craft. We would then muster the resources of critical theory and subtle argument in order to extend this analogy, all the while begging the question of whether we are dealing with a craft at all. As (Wittgenstein, 1953, §414) has it,

> You think that after all you must be weaving a piece of cloth: because you are sitting at a loom — even if it is empty — and going through the motions of weaving.

This kind of error is rife — for example, (McCullough, 1996, p. 119), a seminal work on the craft status of software construction, argues

> But to say that computing is limited because it consists of no more than these few tasks is like saying painting is no good because all you can do is apply color to canvas.

Instead, I will consider the software *that we actually have* — and compare it, in terms of the value that its artefacts offer to society at large, to the highest craft practices that humanity has achieved. I will argue that a change in the values that we bring to our work can bring about better roles for software artefacts in our lives. Craft artefacts are produced within a cultural context in which they can seem natural, and can be holistically integrated into a life or a household. They are connected to human tactility, usefulness, ergonomics, beauty and durability — all virtues that I will consider as defining true craft artefacts. These artefacts take part in an ecology of similar artefacts in similar roles, as well as other cooperating artefacts in the lives of their owners. I will propose a system of values in software construction that promote the *authorial stories* which structure these ecologies — how they pass from hand to hand and take part in networks of creative accumulation and reuse. This in turn feeds back with implications on what we understand by the nature and properties of the "material" which software should be made of.

2. Craft Status of Software Artefacts

The craft status of software development remains limited, in the crucial senses which mark the phenomenon and results of craft as highly desirable to society. In a looser sense, software development

[1] Yet

shares some of the aspects of a craft — in the same way that a DIY enthusiast, journeying to their local hardware store in search of cheap, highly standardised components with poor manufacturing values can be said to be indulging in a craft as they seek to make some more or less successful home improvements. But these aren't the values we should appeal to as we position our "craft" amongst its more established cousins.

3. Virtues of Craft

As William Morris remarked as he began to establish the English Arts and Crafts movement in 1880, "Have nothing in your houses that you do not know to be useful, or believe to be beautiful" (Morris, 1880). A true craft produces artefacts that are

Beautiful - giving pleasure in use and contemplation

Durable - long-lived, stable, and if degrading, degrading gracefully

Ownable - can be the possession of their owner, living amongst them in their household

Handleable - fulfil a function which sight and touch fit into an ecology of similar artefacts

In contrast, the outputs of computer science that we see in practice are ugly, brittle, masterful and obtuse.

Let us consider our virtues in turn.

3.1. Beauty

There is an aesthetics in computer science, but it is not for the user and rarely even for the developer. The model for beauty with any currency is imported wholesale from mathematics — an algorithm can be elegant, a proof beautiful. In contrast, any actual piece of software, especially one which performs a useful function, is without fail fantastically ugly, reflecting the haphazard and throwaway values of real software culture. As Simon Peyton-Jones remarked at the PPIG keynote of 2015, of open source - "It's where people go to contribute, but it's not beautiful". A pivotal early essay on the craft of programming, (Dijkstra, 1977) gives a fundamental role to what is called "the beauty of a program", associating it with such inhuman virtues as the brevity of a proof of its correctness. However, (DeMillo, Lipton, & Perlis, 1979) have recognised that "Formal verification of programs, no matter how obtained, will not play the same role in the development of computer science and software engineering as proofs do in mathematics." A piece of software which possesses desirable, useful beauty will exhibit this beauty as it is worked on throughout its long life. I have seen no sizable or long-lived program that exhibits such a thing[1].

3.2. Durability

The lack of beauty in actual software is intimately tied up in the lack of possibility for any kind of durable expression — and still less, of any durable function. It is well-known (Programmers' Stack Exchange, 2014) that software will spontaneously "rot" as a result of a progressive violation of its environmental assumptions. Sadly, it does not rot gracefully in the manner of a forest log, but disgracefully in the form of simply failing to install or run correctly or indeed at all. The only hope one has for running an old piece of software is to create a complete replica of its original home (a "virtual machine") correct in every detail, and keep it captive there. If software were a lifeform, it would be an impossibly brittle and ill-adapted one that would mark in every case an evolutionary dead-end.

Unfortunately, our field does not so far have the courage to face up to this problem. Indeed, one of its leading lights, Alan Perlis, in (Abelson & Sussman, 1985) speculated whether software might not inherently lack durability, comparing it to a "soap bubble". I can't infer from Perlis' remark whether he believed software should lack durability or simply that we lack the understanding to make it so. I suspect the latter — and if most software didn't possess the beauty of a junkpile or shantytown rather than a soap bubble, we could be more sympathetic to its lack of durability.

[1]With the possible exception of Donald Knuth's TEX typesetting system, which being almost exclusively the product of a single mind, has not entered a traditional software maintenance process.

3.3. Ownability
Because it is not durable, but also through further causes, software can never effectively be owned by its user. Software is less ownable now than it has ever been — with the prevalence of "leasing" models for software, and increasingly aggressively pushed "updates" which will sometimes interrupt vital human activities in order to deprive them temporarily or permanently of the use of the artefact that they just believed was theirs (Hoffer, 2016). This is not the behaviour of a tool or a product, but of a tin-pot aristocrat transplanted into the user's home.

3.4. Handleability
Software is legendarily opaque in form and function. It is well-attested that the vast majority of users are unaware of or unable to exploit the majority of configuration or function which is in theory available to them (Spool, 2011). The output of a craftsman, whether it takes the form of a spoon, bowl, basket or chair, even if it is of an unfamiliar form, typically appeals to an immediately obvious use value. Part of the problem could be ascribed to the novelty of software function, but users have little incentive to invest effort in becoming familiar with their software, because of our previous point — they have no control over its form or any ability to resist change. At any moment the familiar may once again be swept away by the unfamiliar.

4. Our Lack of Materials
Some of the discourse around software "craft" centres on the property of its "material". We argue that the core of the deficiencies of software can be ascribed to the lack of anything which properly qualifies as its material. A true material is the result of a natural process, accumulated and shaped for years if not millennia. When one applies craft to a natural material, one is engaging with the natural accident of this process when meeting difficulties of resistance. By contrast, when meeting resistance in crafting software, one is primarily tangling with the obtuseness of one's colleagues and their insistence in retaining authorial power for themselves.

Before software could become a craft, we need to construct its material. The durability of a true material, as well as enjoying the sheer ability of continued survival, also has the aspect of the ability to participate in creative networks. A product made of a material can be passed from hand to hand, at each point being seen in a different role or being the starting point of a fresh creative cycle. We'll return to the nature of our materials in section 9.

5. Workmanship Leading to Craftsmanship
(Pye, 1968) makes a crucial distinction between grades of workmanship — the "workmanship of risk", where the outcome is at risk at every stage, and under the direct control of the worker, and the "workmanship of certainty" where variability has been eliminated through a industrial process. Pye is clear that only the former kind of workmanship can lead to true craft, whereas computer science exclusively values the latter — in its use of proofs, abstractions, implementation insulation, and other techniques employed to take power away from the user, by erasing variability and control from production. Pye is also clear to value the contributions of engineers, who alone are responsible for defining the nature of materials which lead to real results, as opposed to theoreticians and scientists who accept such materials as a *fait accompli* without assigning or recognising credit. (Gabriel, 2012) is also clear to point out that it is most often the case that principled engineering work long precedes formalisation.

(Lakatos, 1976) on certainty:

> "Certainty" is far from being a sign of success, it is only a symptom of lack of imagination, of conceptual poverty. It produces smug satisfaction and prevents the growth of knowledge.

This leads the way to characterising what constitutes the "risk" in our discipline. (McCullough, 1996, p. 212), himself following Pye, asks

> …must a true medium entail sufficient risk and irreversibility to demand the rigor and

> devotion that have always been necessary for great works? Can a computer with its *undo* and *save as* functions ever demand sufficient concentration on our part to enable serious, expressive works to come forth?

Two major routes by which we could or do face risk in the presence of "undo" are discussed in the following sections.

5.1. Risk through Authority

The first lies in the authoritarian structure of our current authoring idioms. In practice, only a very limited number of creative choices are delegated to the author dignified by the reductive term "end user" — and only these can effectively be undone. The greater part of the choices are reserved to the designers and programmers of the tower of language, platform, application beneath the user's feet — and these decisions, from the user's point of view are irrevocable and can never be "undone". In practice, the discourse around design typically stresses the removal of affordances from the user that the designer considers inappropriate or violate design guidelines from their point of view.

An interesting example of this kind of "dialogue against design risk" emerged during the workshop conversations supporting (Computing Community Consortium, 2015). One participant warned that user interfaces should not be made too configurable, otherwise the helpdesk tasked with supporting the user with their software might be rendered incapable of helping them — because the user had reconfigured their interface so radically that the helpdesk staff couldn't recognise it.

This is a form of "positive risk" that we are not exposed enough to. We should try to promote authorial systems and design communities which allow owners of artefacts to run more such creative risks, whilst minimising their costs.

5.2. Risk through Unmanageable Complexity

Another route to exposure to risk in the presence of "undo" lies in the potential for the complexity of artefacts to exceed our ability to manage or comprehend them. This is a risk for amateurs and professionals alike — if we can no longer comprehend the authorial affordances that we have, and reliably distinguish the virtues of one version of an artefact from another, it matters little whether all versions of the artefact that have ever been authored have been perfectly stored and can be instantly retrieved. (Eick, Graves, Karr, Marron, & Mockus, 2001) report "anecdotal evidence of systems that have reached a state from which further change is not possible".

This can be described as a form of "negative risk" that we are all already exposed to — our challenge is to structure languages and frameworks to minimise the chances of artefacts being "lost" to this risk.

6. Lutyens' Latch and Barn's Spoon

As an example of the kind of craft I am appealing to, I firstly exhibit the example of this door latch (Figure 1) designed by Edwin Lutyens, a prime proponent of the Arts and Crafts movement. Latches like these can be found throughout his restoration of Lindisfarne Castle which he undertook starting in 1902, as well as some of his other designs. It self-evidently enjoys all the virtues we have listed — it is highly beautiful (even achieving a kind of wittiness[2]), and its method of construction, based on readily available and widely understood materials, is a direct guide to its function. In the unlikely event that it requires maintenance (having endured intact for over a century), this can be achieved through widespread means.

Next, as a contemporary example, I exhibit a 'cawl' spoon (Figure 2) carved from rippled sycamore by a modern representative of the Arts and Crafts tradition, Barn the Spoon, at his shop in Hackney Road, London. Again it enjoys all of our virtues as well as participating in the *vernacular* both at the level of its design and its construction — this spoon was carved in the presence of its buyer at Barn's shop, creating a direct link between the worlds of authors and users.

[2]Note that this wit is only interpretable with respect to the *ecology of function* represented by latches in general.

A lack of durable and natural materials, as well as a suitable craft culture, represent the primary deficits in computer science which prevent it from constructing crafted items such as these.

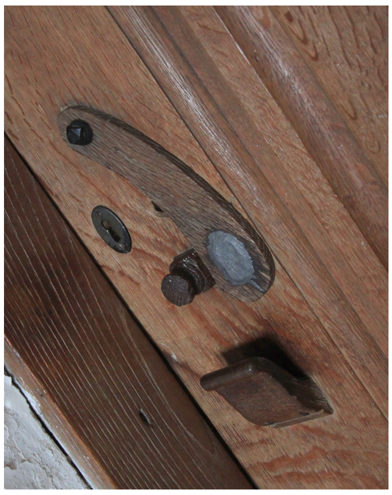
Figure 1 – Door latch designed by Edwin Lutyens

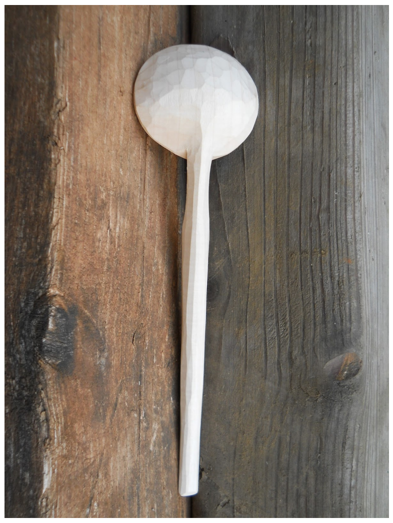
Figure 2 – Spoon designed by Barn the Spoon

7. Spectrum of Forgiveness

Whilst the software that we have is constituted of a material far more brittle and less malleable than the least forgiving of the human crafts (for example, ceramics), being a product of the mind, it has the potential to be far superior to the most forgiving (for example, knitted textiles). The following table illustrates the affordances that we can expect from different materials.

Ceramics

- Once the product's form is fixed, it cannot be further altered in a graceful way.
- It is brittle and once damaged cannot be gracefully repaired (a glued repair is always obvious, and a damaged surface can never be reformed like the original)

Wood

- With some effort, an item can be reformed or repaired, for example by sanding, varnishing, repolishing etc.
- For some items, a broken element can be removed (unglued, unscrewed) and replaced with a new part which functions and appears as well as the original

Textiles

- Most items can be repaired, sometimes indefinitely, by patching or darning
- In some cases the original raw materials can be fully recovered by unravelling the garment and a completely fresh one constructed

In contrast, we have

Computer Science of the Present:

- A product may spontaneously disintegrate without warning, suddenly becoming wholly unusable
- No modifications may be made by the owner after delivery of the software, even through amateurish affordances such as superglue
- The raw materials for software are endlessly distant from the owner, separated by a series of barriers operated by successive priesthoods (compilers, linkers, integrators, hosting services)

Whereas nothing inherent in Computer Science prevents it from being

Computer Science of the Future:

- Any product, either in form, function, or both, may be preserved indefinitely in a working condition

- Any owner of software may modify and maintain it using only the affordances to hand — or else share or receive such modified versions from peers or communities of interest
- The raw materials for software are constantly immanent — any user of software, simply through an act of perception, may choose to see it either as raw materials, finished product, or anything in between

From its current position as the world's least forgiving craft, computer science could surpass all those currently known.

8. Forces Producing the Craft We Have

We have the craft that we do, because of the power structures and mentalities embodied in our communities. As (Blackwell, 2010) notes:

> Even supposedly universal "general purpose" computer languages, although assumed to be culture-free, become imbued with metaphors of power and control that are easily recognized when inspected from outside the perspective of their designers and users.

Computer scientists, in theory charged with planning and directing the formalisms underlying the practices of software construction, have no appreciation of how software is actually made and on whose behalf. In turn, software engineers adopt methodologies that retain power for themselves at the expense of their colleagues and users, rather than appreciate that design and usability concerns should be the paramount ones governing their work[3]. These usability concerns should be those which emerge when artefacts are deployed as part of their owners' households and ecologies of living, rather than reflecting a process of "abstracting away the user"(Blackwell, Church, & Green, 2008).

8.1. The Term 'Craft' in Our Lineage

We can trace back the threads of this discourse as far back as we like. For example, (Dijkstra, 1977)'s original essay on software craft status insisted that software production should *cease* to be a craft, rather than become one - "a craft was applied, where a scientific discipline was needed". How could Dijkstra have come to make such a profound category error? It is because he already inhabited a completely self-sufficient discipline, answerable only to its own aesthetic and formal standards. Dijkstra is already capable of looking at the only "users" he has known, professional physicists and engineers, with a kind of amused contempt for their tendency to "write a three-page program in an afternoon". The thought that even less-well formally oriented citizens, the general public, could gain access to the affordances of crafting the software that they need, was unthinkable — these are people "the majority of whom it is totally unrealistic to expect that they can still acquire a scientific attitude". Those who imagine that ordinary citizens could become thus empowered are condemned as "foolish". These are the roots of the kind of disciplinary imperialism that within a few decades would bear fruit as the pernicious "computational thinking" movement of (Wing, 2008).

8.2. Mistaking the Past for the Future

But how could Dijkstra have come to believe that software production was a craft in the first place? The answer lies in the opening paragraphs of this paper. He had become confused by the simple fact that the software production process required special insight and skills as a result of its hopelessly unreliable tools and materials. In fact, in most cases the "craft" of his and our day doesn't even aspire to the level of hardware store, flat-pack DIY furniture values that we described. It is as if we bought a kit of parts for a flat-pack cupboard, and found them hopelessly ill-fitting, some pieces missing and the instructions written in an unrecognisable language with no illustrations. This is certainly the experience even of a consumer of what should be the most standard, well-understood and fully reduced to practice products of computer science — parsers. Anyone who has proposed to design a new programming language and expected to pick up a parser technology for it "off the shelf" will respond to this point.

[3]This similarity of workflow and outlook is why I fail to particularly distinguish between computer scientists and software engineers in this essay.

Rather than, as Dijkstra claims, software production needing to *cease* to be a craft, it has not even got to the starting blocks where it could qualify as one. His appeals for a "scientific discipline" and a "scientific attitude" are wholly misplaced. As is well known, computer science is not a science (Abrahams, 2013). Its subject matter is not a domain governed by natural laws[2] but instead the communities of real human beings, their needs, cognition, and psychology. Properly understood, Dijkstra's plea is simply one for reliability — but reliability does not signify the end of craft, but merely its beginning. Imagine a discipline of carpentry operated by chisels which might unpredictably bounce off perfectly ordinary-looking pieces of wood, or whose tips might spontaneously disintegrate. This is the world of tools and materials which computer science has made for itself. Acquiring reliable tools doesn't mark the beginning of a science, or Dijkstra's "production-line worker" — instead "Reliable tools are the beginnings of spiritual freedom"(Matthews, 2016).

9. What Could Materials Be?

What could the materials be like, which might be the medium for the craft we want?

9.1. Material and Immaterial — Supporting Authorial Stories

There's potential confusion and paradox in this area, since from a traditional point of view, these materials are "immaterial" since they aren't directly constituted as matter in the world that we can grasp with our hands. (Gross, Bardzell, & Bardzell, 2014) have drawn up three principal models for the materials of software — broadly speaking, these are i) the traditionally physical, as embodied in tangible user interfaces, ii) the metaphysical, embodied in the relationships between information, and iii) the traditions, relations and communication that artefacts enable between people. Of these, I am most closely in sympathy with the third model — material can be anything it needs to be, either material or immaterial, that enables the open, unbounded authorial stories that promote participation in society. Traditional materials support the growth of traditions — as artefacts are passed from hand to hand, they inspire others not only by the stories embodied in them but by the fresh stories and uses they may participate in. Today's software materials support only "dead-end" authorial stories. To adapt an artefact built with today's languages in a way other than its authors intended, one must first "fork" the source code of the artefact in order to modify it. Version control techniques only provide a paper trail by which this process could be audited, rather than a transparent means for communities downstream of the fork to remain in contact with the upstream.

How could we build software that allows us to "change things without changing them"? Just as with the apparent distinction between the 'material' and the 'immaterial', this paradox will lose its force through a change of perspective. The paradoxes of today will be the unexamined assumptions of tomorrow.

9.2. The Density of Materials

(McCullough, 1996, p. 196-197) cites a quality of materials that underpins their suitability for craft — their *density*. Each state of the material is closely surrounded by a dense collection of neighbouring states that behave similarly. As McCullough puts it, "Density supports engagement not only through continuity but also through variety". However, today's software materials, traditionally consisting of source code text in an editor buffer, could not be more sparse. The "nearest neighbouring program" to any given one is separated from it by a vast ocean of syntactically invalid or crashing variants.

A movement exploring such density at the hardware level is known as "circuit bending" (Ghazala, n.d.) — these groups of enthusiasts assemble components from low-powered electronic circuits into random configurations and expose them to random electronic signals (within a safe range). When applied to games consoles of the 1980s, for example (Belojevic, 2014), a surprising amount of interesting behaviour results — recognisable sprites or game elements will appear on screen and sometimes animate. This behaviour shows that the space of allowable system states is *dense* within the overall space of available states, and is inconceivable in a system of modern hardware or software. One approach to increasing the density of valid authorable states is simply to make them more *findable* via assistance

[2]despite the belief of some leading lights of the field, e.g. (Hoare, 2014)

from powerful IDEs or code completion techniques. This can carry us a certain distance, but more fundamental approaches to rethinking the structure of software materials can make the space of these states intrinsically dense and naturally navigable. I'll refer briefly to some such approaches in the next section.

9.3. Some concrete approaches to our new materials

(Basman, Church, Klokmose, & Clark, 2016) explores some concrete strategies for bringing true materiality to software, by expanding the repertoire of available moves for building authorial stories. A traditional distinction between artefacts is between those which are "live", that is, comprise currently executing systems, and those which are "dead", taking the form of textual documents or directly authored files on disk. The central dogma of computer science, that live systems are only derived from dead ones, and not vice versa, is one of the many inflexibilities that prevent communities trying to share software artefacts from remaining in touch with each other. A side-effect of bridging the gap between live and dead systems, by corresponding them more closely, is to increase the density of their authorable states (with respect to invalid ones) — since these states are naturally coordinatised as a result of the correspondence.

10. Resolution

Our field is making only tiny and halting steps towards craftsmanship. It is increasingly popular to show relevance to craftspeople by an appeal to our ability to work in their spaces or with their materials (see (Victor, 2014) or other "Maker" movements), but the only result will be to export our own aesthetics of frustration and power-worship into already functional communities. The experience of (Blackwell, Aaron, & Drury, 2014) exhibits the behaviour of craftspeople when provided with pieces of "techno-junk" such as a USB keyboard which doesn't work with a USB socket and other similar obstructions:

> *experiences were … "frustrating" — a word that was used consistently and repeatedly in the artists' reports*

with the result that

> *busy artists are equally likely to respond to obstacles by shifting their effort to other projects in which they are making rapid progress*

Computer scientists show little recognition of the frustration that their products cause in practice, and instead imagine that the world would be a better place if more people thought and worked like them — witness the rise of "Computational Thinking" (Wing, 2008) as the vehicle for this imperialism. Rather than the virtues we list above, the highest virtues of computer science are the tedious accountancy of efficiency and correctness. These will create a civilisation of bureaucrats rather than craftspeople.

In some quarters, there are signs of a change of heart. (Blackwell et al., 2008) turn against the tide of computational thinking towards a respect for the individual human, and (Blackwell & Collins, 2005) admit the possibility that "an end-user programmer may well prefer to accept the results of an imperfect execution" — an appeal to the "workmanship of risk".

However, there are few concrete signs of the durable yet forgiving materials that must underlie true crafts. There are some simple, house-clearing tasks that we should turn our resources to, before we are prepared to achieve the higher virtues:

- To guarantee to every user that, once given an artefact, they can always choose to use it in its current form, regardless of "improvements", updates, or environmental changes
- For every piece of software to be worked on by means of itself, rather than through the use of specialised tools only available to an elite at a different site, at a different point in its history
- Given a user's preferred way of working and seeing, for any system that they start to work with, it adapt to whatever extent it can to meet those preferences

The latter three of our virtues are something that can be worked on in our lifetimes — and if we succeed in constructing better, more forgiving materials, our descendants might possibly use them to construct something which is beautiful.

Acknowledgements

I am grateful to Colin Clark for insightful comments on previous drafts of this essay, and for many of the connections drawn within it.

11. References

Abelson, H., & Sussman, G. J. (1985). *Structure and Interpretation of Computer Programs*. MIT Press.

Abrahams, P. W. (2013). Computer science is not a science. *Communications of the ACM, Letters to the Editor*, *56*(1), 8.

Basman, A., Church, L., Klokmose, C., & Clark, C. (2016). Software and how it lives on - embedding live programs in the world around them. In *Proceedings of the Psychology of Programming Interest Group.*

Belojevic, N. (2014). *Circuit bending videogame consoles as a form of applied media studies.* Retrieved from `http://www.nanocrit.com/issues/5/circuit-bending-videogame-consoles-form-applied-media-studies`

Blackwell, A. (2010). When Systemizers Meet Empathizers: Universalism and the Prosthetic Imagination. *Interdisciplinary Science Reviews*, *35*(3-4), 387–403.

Blackwell, A., Aaron, S., & Drury, R. (2014). Exploring creative learning for the internet of things era. In *Proceedings of the Psychology of Programming Interest Group* (pp. 147–158).

Blackwell, A., Church, L., & Green, T. (2008). The abstract is 'an enemy': Alternative perspectives to computational thinking. In *Proceedings of the Psychology of Programming Interest Group* (pp. 34–43).

Blackwell, A., & Collins, N. (2005). The programming language as a musical instrument. In *Proceedings of the Psychology of Programming Interest Group* (p. 120-130).

Computing Community Consortium. (2015). Promoting strategic research on inclusive access to rich online content and services.. Retrieved from `http://cra.org/ccc/resources/workshop-reports/`

DeMillo, R., Lipton, R., & Perlis, A. (1979). Social Processes and Proofs of Theorems and Programs. *Communications of the ACM*, *22*(5), 271-280.

Dijkstra, E. (1977). Programming: From Craft to Scientific Discipline.. Retrieved from `https://www.cs.utexas.edu/users/EWD/transcriptions/EWD05xx/EWD566.html`

Eick, S. G., Graves, T. L., Karr, A. F., Marron, J. S., & Mockus, A. (2001, January). Does code decay? assessing the evidence from change management data. *IEEE Trans. Softw. Eng.*, *27*(1), 1–12.

Gabriel, R. P. (2012). The structure of a programming language revolution. In *Proceedings of the ACM Onward 2012* (pp. 195–214).

Ghazala, R. (n.d.). *Circuit-bending: a bender's guide.* Retrieved from `http://www.anti-theory.com/soundart/circuitbend/cb01.html`

Gross, S., Bardzell, J., & Bardzell, S. (2014, March). Structures, forms, and stuff: The materiality and medium of interaction. *Personal Ubiquitous Comput.*, *18*(3), 637–649.

Hoare, T. (2014). Laws of programming: The algebraic unification of theories of concurrency. In P. Baldan & D. Gorla (Eds.), *CONCUR 2014 – Concurrency Theory: 25th International Conference* (pp. 1–6). Berlin, Heidelberg: Springer Berlin Heidelberg. doi: 10.1007/978-3-662-44584-6_1

Hoffer, S. (2016). *Windows 10 Update Interrupts Weather Report.* Retrieved from `http://www.huffingtonpost.com/entry/microsoft-update-weather-report_us_572241f2e4b0f309baf0082a`

Lakatos, I. (1976). *Proofs and refutations: The logic of mathematical discovery.* Cambridge University Press.

Matthews, R. (2016). Personal communication.

McCullough, M. (1996). *Abstracting Craft: The Practiced Digital Hand.* MIT Press.

Morris, W. (1880). *The Beauty of Life.* A lecture before the Birmingham Society of Arts and School of Design (19 February 1880).

Programmers' Stack Exchange. (2014). *What is meant by "code rot"?* Retrieved from `http://programmers.stackexchange.com/questions/255866/what-is-meant-by-code-rot`

Pye, D. (1968). *The Nature and Art of Workmanship.* Cambridge University Press.

Spool, J. (2011). *Do users change their settings?* Retrieved from `https://www.uie.com/brainsparks/2011/09/14/do-users-change-their-settings/`

Victor, B. (2014). *Seeing spaces.* Retrieved from `http://worrydream.com/SeeingSpaces/`

Wing, J. M. (2008). *Computational thinking and thinking about computing.* Retrieved from `https://www.cs.cmu.edu/afs/cs/usr/wing/www/talks/ct-and-tc-long.pdf`

Wittgenstein, L. (1953). *Philosophical Investigations.* Macmillan.