Coding, Humanism, Pedagogy.

Sam Popowich
Plenary Session, Digital Pedagogy Institute, University of Toronto – Scarborough, August 19 – 21, 2015.

Over the last few years, the topic of technological or digital literacy has come to the fore in librarianship. In public library systems where this is most common, it can be understood partly as an extension of the provision of personal computer and internet training which began in the mid-1990s, and partly as an adoption by libraries (again, mostly public) of "maker" and "hacker" culture. From the public-library perspective, digital literacy support tends to take the form of training, assistance, equipment, and space – makerspaces are a growing element of library spaces and services. On the staff side, the question of digital literacy has often focused on whether librarians should learn to code. "Makerspaces" and "coding for librarians" are two major focal points of debates around social class, gender, and race in libraries, contested through the lens of privilege and diversity. On the one hand, some argue that makerspaces and coding have visible practical benefits independent of social, cultural, and political issues; learning and teaching new skills, making space and equipment available to all is seen as levelling. On the other hand, privileging making and coding can be interpreted as perpetuating race, class, and gender dominance, in fields (librarianship and technology) that have not yet solved (and in some cases not yet recognized) their race, class, and gender problem.

We can't divorce programming as a skill or a practice from its social, cultural, and political context. But at the same time, it is a valuable skill in the information and software driven society of the 21st century. The recognition of its importance, and of the ubiquity of technology in general, tends to fall into two camps: a dystopian technological determinism, and a kind of utopian humanist paradise, both of which drastically oversimplify – or even ignore - the complexity at issue here. Is it possible to adopt coding as a skill without subscribing either to determinist or utopian ideologies? Might we look at things dialectically and discover a third term, which might allow us to open up a space of resistance to the patterns of domination in which coding is embedded without closing our eyes to the limitations and structural inequalities of that space? Might we think about learning to code as part of a critical, liberation pedagogy, as outlined by Freire?

I don't consider myself qualified to speak about issues of race and gender. While I acknowledge that race and gender problems exist at the exact conjunction of the two fields I participate in professionally, librarianship and technology, there are other people better qualified to talk about the experience of gender and racial discrimination in this area, people like Andromeda Yelton, Cecily Walker, Chris Bourg, Bess Sadler, and others. But what I'm interested in speaking about today is the question of agency – that is, class - within the structures of early 21st century capitalism.

In many ways, the work of the 21st century is simply the continuation of mechanization and systematization of the work of the 19th and 20th centuries. We live in the age of the software and computer engineer, and of hyper-taylorism in all that can't be engineered away, a good example of which are the disclosures over the last year or two of the way Amazon "manages" its warehouse staff. The scientific, engineering, utopian worldview, with its assumptions of reason and meritocracy, privileging of scientific truth, and promise of salvation, lies behind many of the cultural struggles we see today. In many ways the rise of "geek culture" is a symptom of this. Engineering seems to offer a pragmatic route out of the quagmires of essentialism, argument, and subjectivity that are unavoidable and interesting (if messy) components of the human experience. We might resist the totalizing impulse of engineering, but we can't avoid it. I would like to talk a little bit about how, as humanists, we might use the quintessential engineering skill of computer programming to further humanist projects, support humanist enquiry, and contribute to humanist pedagogy.

Many people have written on the dichotomy between what I'm going to call engineering and humanism. Karl Marx spoke of the tendency of capitalism to replace human labour by machinery in the mistaken belief that inefficient human labour is a limit on profit, and Horkheimer and Adorno addressed the consequences of a privileging of the Enlightenment's "instrumental reason". David West, in his 2004 book on object-oriented design and programming, *Object Thinking,* calls the two terms of this dichotomy "formalism" and "hermeneutics" and defines the distinction as one between rationalism and determinism on the one hand, arising from the work of Descartes and Leibniz, and on the other hand, emergence, relationality, and interpretation. In West's view, the two approaches can be exemplified in computer science, in the debates around Artificial Intelligence that arose in the 1970s. Formalist AI, based on a symbolic theory of cognition, saw intelligence as deterministic, while those who advanced a hermeneutic position saw intelligence as behavioural and emergent.

Closer to home, in “The Humanist as Reader”, Anthony Grafton describes the differences between the practices of medieval and renaissance readers. Grafton compares the work of the medieval scholastics to that of builders:

> By decades of hard work with hammer and chisel, they fashioned a complex Gothic set of walls and buttresses which preceded, surrounded, and supported the texts: headings, commentaries, separate treatises. This apparatus succeeded in imposing a medieval outlook on the most disparate ancient texts.[1]

The humanists, in reaction to the over-engineering of the scholastics, rescued classic texts by stripping them of their medieval accretions, looking for an unmediated engagement with the text. The medieval texts

> were laid out in two columns and written in a spiky, formal Gothic script. They occupied a relatively small space in the centre of a large page. And they were surrounded, on that page, by a thick hedge of official commentary written in a still smaller, still less inviting script. (…) Such books naturally repelled Renaissance scholars, to whom they seemed a visual as well as an intellectual distortion of their own content. [2]

New designs for books, scripts, and libraries, replaced the old Medieval versions, and allowed reading to escape from the confines of Medieval authority, both intellectually and physically. The pocket-sized editions

[1] Grafton, p. 182.
[2] Grafton, p. 184.

of Manutius and others, printed in the new humanist hand, allowed people like Machiavelli to colonize new spaces of reading, spaces other than libraries, scriptoria, or private studies. Grafton opens with a letter from Machiavelli describing the variety of his reading practice, reading different books in his study than in the woods. Such informal reading was impossible with medieval books.

To my mind, this distinction between the over-engineered, Medieval approach to texts and the unmediated, enquiring approach of the humanists, is apparent in many different disciplines. My academic background is in cultural musicology, and one of the areas of debate in the field is around performance technique. A distinction is drawn between the formalist, engineered technique of the conservatories, and the relational, emergent technique learned in jazz clubs or on the streets of Rio. An example from "classical" guitar will suffice.

John Williams, one of the main exponents (along with Julian Bream) of the post-Segovia classical guitar, is often acclaimed for his "perfect" tone and technique, and at the same time criticized for his lack of emotion and "robotic" playing. His renditions of Bach and Barrios allow the listener to hear every note, even at high speed, and in some ways it could be argued that Williams "gets out of the way" of the music.

But criticism is levelled at the lack of humanism, of Williams' over-engineered sound. Compared to Williams, the jazz guitarist Lenny Breau, and the jazz/samba/bossa nova guitarist Baden Powell sound sloppy, with imperfect technique. Comparing a well-known piece of Bach's played by all three guitarists, exposes their different styles and approaches. On the one hand, Williams' performance is engineered to be perfect, repeatable in Walter Benjamin's sense: the outcome is pre-determined. For Breau and Baden Powell, on the other hand, the performance is extemporized, exploratory, and open.

There is nothing wrong with either of these approaches. Williams' recordings of Bach and Barrios have been an important part of my musical experience for many years. But there is something in Breau and Baden Powell that resists, in a ragged way, the instrumental logic of capitalism. Their music seems to repay repeated interrogation and investigation in a way that Williams' does not.

I've introduced these examples of engineering vs. humanism in order to raise the question of employing typically "engineering" practices in the humanist disciplines. In a recent talk, Avdi Grimm, a well-known figure in the Ruby programming community, approached this topic from the other direction. Speajing to engineers At the Tropical

Ruby conference in Brazil earlier this year (2015), Grimm argued that the Ruby community is well-placed to provide “an end-run around the formalist school”, a space where informal, hermeneutic approaches to software development can thrive[3]. “The Ruby community,” he says, “maybe more than any other community, is steeped in informalist, agile practices”. The Ruby programming language, is “an informal language”: “Ruby doesn’t try to tell you how to do things”; its community values “conversation and consensus over top-down architecture”. Grimm sums up by saying that “in a programming world that pays lip-service to OO ideas while still teaching formalist methods, Ruby is a stubborn island of informalism”.

But why is any of this important? Is this more significant than internecine squabbles within the software-development field? I believe it is. The point of Grimm’s talk is not that “Ruby is the best, most community-driven language and all other languages are bad”, but that if we understand programming languages as representing particular philosophies or world-views, then the informal, behavioural, hermeneutic way of describing, understanding, and working with the world that Ruby espouses can be a valid, new approach to problems that

[3] Grimm, Avdi, “The Soul of Software”, Tropical Ruby Keynote 2015. Youtube.

had previously been understood as purely engineering problems. Understanding the distinction between formalism and hermeneutics, in Grimm's view, opens a space within software development for informal, emergent, hermeneutic practices.

Much of Grimm's talk draws on West's *Object Thinking*, an introduction to the philosophy and culture of object-oriented design and programming, and agile software development. Both object-oriented programming and agile were developed as methods of resisting traditional, monolithic tendencies in software design and development, organizational culture, work processes, collaboration, and team composition. In a nutshell, agile methodologies attempted to create space for more of what we have been calling informal, hermeneutic practices. West's thesis, that object thinking and agile methodologies can improve both programmers and software, is founded on the idea that "software development is neither a scientific nor an engineering task. It is an act of reality construction that is political and artistic".

> Extreme programming – and to a great extent all of the agile methods – represents a new and creative idea about software development. (…) The essence of this new idea might be distilled to a single sentence: "Software development is a social activity".[4]

[4] West, 26.

For West, agile methodologies are not simply just another variety of software engineering, but represent the latest in a long line of challenges to the scientific, formalist, engineering worldview of software development:

> Sharing the goal of software improvement but rejecting the assumptions implicit in software engineering, several alternative development approaches have been championed. Creativity and art have been suggested as better metaphors than engineering. Software craftsmanship is the current [2004] incarnation of this effort.[5]

Now, in order to set object thinking and agile methods up as an alternative to traditional software engineering, West places significant theoretical weight on the dichotomy between "formalism" and "hermeneutics". That this is an over-simplified model should be apparent – as always the truth is dialectical - but I hope I've shown that there is an intuitive recognition of the at least local validity of thinking in these terms. For West, formalism has its roots in the Enlightenment and 20th century successes in science and engineering.

> The universe was considered a kind of complicated mechanism that operated according to discoverable laws. Once understood, those laws could be manipulated to bend the world to human will. Physicists, chemists, and engineers daily demonstrated the

[5] West, xvii.

> validity of this worldview with consistently more clever and powerful devices.[6]

It is useful to contrast this utopian view of rationality with the view expressed in Horkheimer and Adorno's *Dialectic of Enlightenment*:

> Interested parties explain the culture industry in technological terms. (…) A technological rationale is the rationale of domination itself. It is the coercive nature of society alienated from itself. Automobiles, bombs, and movies keep the whole thing together until their levelling element shows its strength in the very wrong which it furthered. It has made the technology of the culture industry no more than the achievement of standardisation and mass production.

This connection may seem tenuous, the theoretical freight too much for West's argument to bear. But Avdi Grimm *does* make the connection between an informal, humanist, hermeneutic conception of software development and positive social change. Just as "software development is a cultural activity"[7], so the philosophy behind object thinking and agile methodologies is grounded in the possibility of creating a positive culture, of creating consciousness that might resist central control.

Elements of this hermeneutic, humanist culture are:

[6] West, 48.
[7] West, 25.

- A commitment to *disciplined informality* rather than defined formality
- Advocacy of a local rather than a global focus
- Production of minimum rather than maximum levels of design and process documentation
- Collaborative rather than imperial management style
- Commitment to design based on coordination and cooperation rather than control
- rapid prototyping instead of structured development
- Valuing the creative over the systematic
- Driven by internal capabilities instead of conforming to external procedures[8]

I won't go into details, but suffice it to say that the tensions in this list exist also within librarianship and, I assume, within most branches of the academy.

The debate around "should librarians learn to code" came to the fore recently, and was even discussed on *Thinks and Burns with Fink and Byrne,* a local library podcast[9]. In participating in this debate, I came to realize what I think is an important, unspoken, assumption: that those who advocate for librarians to learn to program are advocating an engineering approach. The fear is that of totalization: that *all* librarians should learn to code, that coding should be a structural part of LIS education, that our work will be transformed into a dehumanized engineering job. This fear takes for granted the idea that "programming"

[8] West, 65.
[9] Thinks and Burns with Fink and Byrne, episode 3.

and “software engineering” are the same thing, and that the engineering approach is the only approach to working with machines and data through code. I believe that the opponents of “librarians should learn to code” are imputing more value, more weight to that statement than is meant. It’s true that, as Gillian Byrne points out, “librarians should learn to code” is an unhelpful shorthand, a slogan obscuring a very complex dynamic, but this does not justify an immediate leap to “*all* librarians must code” or “computer science classes should be taught in library schools”. There is an alternative, less formal, more hermeneutic approach that I would like to think through.

First of all, I’d like to think about what we mean by “professionalism” versus “amateurism”. Librarianship has a fraught relationship with “professionalism”. It is a field which tends to pride itself on being a profession, while being on the one hand definitely not a profession in the sense that lawyers, doctors, and engineers are, and on the other hand, perennially unsure and insecure about its work and its value. Because of this, as well as the challenge of managing extremely complicated information and organizational systems, there is a tendency to try to formalize all the work that we do, in the belief that it will make our professional status more convincing. This makes sense: in order to make the case for budgets, for staffing, to be taken seriously in the academy, we need to be able to point to our qualifications and achievements. But I think that this approach can sometimes prevent us from being agile or innovative, makes us slow to change, and slow to adopt new techniques and approaches, because we require so much formal and professional apparatus to be adopted at the same time. We tend to overthink, overengineer, and overprofessionalize our practices. We find it difficult to accommodate and employ "soft", "subtle", and "amateur" skills, approaches, and talents, at the same time as we recognize their value. A good example is the processes surrounding reference desk interaction: while recognizing the necessity of agility, personality, flexibility, relationships, etc, we constantly overload reference desk staff with policies, procedures, and documentation.

When I speak about librarians learning to program, I mean it in an amateur sense, in the sense that we are amateur cooks, amateur bike mechanics, amateur musicians, or what have you. I think the informal, hermeneutic approach to programming has many benefits for the kind of work librarians and other humanists do, and I don't want to see it smothered under the weight of overengineered qualification or processes.

This kind of amateur approach has only really become available over the last fifteen or so years, with the advent of lightweight, high-level languages which don't require much infrastructure to run. The explosion of web-programming, and the rapid development of Web 2.0 and semantic web/linked data technologies is due in part to this change as well. Python (1991), PHP, Javascript and Ruby (all 1995) took programming back from overengineered languages like C++ and Java and opened it up to amateurs. There had always been amateur programmers, of course, but the combination of these lightweight languages and the web made it easier to write, run, and share code. Today, we see lightweight versions of older, esoteric languages like LISP. Clojure, a LISP-dialect that runs on of the Java Virtual Machine appeared in 2007, and provides an alternative way to thinking about code and data from the more mainstream object-oriented paradigm.

One of the characteristics of languages like Ruby and Python is that they are open-source, and are both products and drivers of the explosion in open-source activity that we've seen in recent years. Open source (and openness in general) is part of the informal, hermeneutic approach. The formalist, engineering approach struggles to accommodate the different standards of work, collaboration, documentation, and process that openness represents. This tension can be seen in debates around all sort of openness: open access publishing, open educational resources, open source software, open data. There is obviously a class criticism of self-congratulation around openness; openness comes with certain educational, leisure-time, financial, and access requirements. But this criticism should not make us proponents of closed-access, closed-data, or proprietary software, which simply plays into the hands of a hermetic class of social engineers.

So what **do** I mean when I say “librarians should learn to code”; what does that slogan, in fact, obscure? Part of what I mean is that there is a certain level of technological literacy that librarians and humanists need, and that coding is a good way to achieve that. Another aspect is that I think programming is an incredibly useful tool in our increasingly data- and software-driven work. But there are many other components to this statement. Many librarians feel exploited by library software vendors, and the quality of closed-source library software is generally not high. Learning to program helps us a) to understand software and speak knowledgably with our vendors and b) gives us back a measure of control over our own machines to make us less dependent on closed-source software and software vendors in the first place. There’s also an information literacy component to this: the more we understand about software, the more we can communicate, teach, and share that understanding. Librarianship has always been a technology profession, and software is the simply current foundation of technology.

Another benefit to learning to programme is the development of computational thinking. Again, this phrase might raise the spectre of a totalizing, formal, inflexible way of thinking, but computational thinking encompasses much more than that. Our technological world runs on computation, but there are many different flavours of computation, and of ways of approaching and thinking about a computational problem. Object-oriented languages, such as Ruby, model real-world or data objects as atomic collections of attributes and behaviour, maintaining a separation between data and the code that operates on that data. Functional languages like Clojure model objects as collections of values, and make no distinction between data and code. Working in each of these languages requires thinking differently about the problem at hand.

Do I think *all* librarians need to code? No, there are plenty of positions where librarians can probably get by without coding. Do I still think it would be a useful skill for them? Absolutely. Do I think librarians need to *code*, as opposed to learning other aspects of information technology, like automation, system administration, use of high-level systems like OpenRefine, Drupal, etc? Again, the answer is a qualified no. There are many librarians active in all areas of technology who have gained the qualities and skills I've been talking about without specifically knowing how to code, who are extremely knowledgeable about software, systems, and information. Do I think learning to code would be useful for them? Sure.

What, then, are the practical elements specific to programming that I think are useful for librarians, and by extension all humanists?

In the first place, and perhaps most simply, there are the repetitive tasks that librarians perform on a regular basis, that could be automated. Parsing data files, producing reports, etc. Quite often these tasks take a non-trivial amount of time and effort, both of which remain constant no matter how many times the task is repeated. Perhaps just as often, the librarian is unable to perform the task themselves, and have to put a request in to IT, where it is assigned a priority, and the librarian waits for a response. Being able to program would allow these tasks to be automated and would not require the intervention of IT. For many librarians, this kind of work, both behind the scenes and in liaison with faculty, would be the extent of their code use.

But that's kind of a trivial example. A more interesting problem is the question of prototyping or domain exploration. A lot of our systems work involves data manipulation, conversion, cleanup, and other kinds of processing. This work cannot be done manually anymore, and so requires software tools to do it. There are two options: buy closed-source software, or deploy/customize/write your own. Many metadata and cataloguing librarians have experience using XSLT to manipulate XML data; this is one way to approach data, but other programming languages take different approaches, providing flexibility in terms of thinking about, modeling, and working with the data that we have. Being able to quickly and easily explore a data domain or problem, to quickly and easily write simple code that allows you to define and begin working with data, without having to go satisfy the (often overengineered) requirements of IT departments, is extremely valuable. This kind of coding as "thinking out loud" is known in the Agile methodology as a "spike", and is meant to be as informal as possible. It's the equivalent of whiteboarding, and just as whiteboarding sometimes leads to a formal solution, sometimes its benefit is purely in quickly and simply framing a problem. Coding allows us to “whiteboard with data”.

Knowing how to code also allows us to overcome limitations of open-source software solutions. My current project at the University of Alberta is to implement a new search interface/library catalogue. Rather than using an unsatisfactory closed-source solution, we decided to go with an open-source project. Open-source is not of course completely “free”, and there was a lot of work to be done in order to customize the software to fit our needs. But we could do this work in-house, we didn’t have to outsource or rely on outside developers to implement the requirements of our domain experts (students, faculty, and librarians).

These are immediate practical benefits to learning how to code. There are other, less tangible benefits. One of the characteristics of capitalism is alienation from the product of labour: building software gives a measure of agency and investment back to workers. This feeling is fleeting, perhaps even delusional, but it is real. Amateur learning itself, in my view, can provide a feeling of resistance to the instrumental logic of training and education that we are all embedded in. Passing along the appreciation of amateurism, through less formal training programs like Ladies Learning Code or Software Carpentry, or through very informal mechanisms like pair-programming sessions or hackfests, can also, I think, open up a space for anti-instrumental and anti-totalizing approaches to technology.

How might this space actually be opened up, in practice? In his 2011 book, *Representing Capital*, Fredric Jameson raises an important question about technology: "is it cause or effect, the creature of human agency or the latter's master, an extension of collective power or the latter's appropriation?" Two attempts at answering this question, in Jameson's view, can be found in technological determinism, and the kind of humanist paradise I began this talk by outlining. But Jameson goes on to say that

> Neither outcome is conceptually or ideologically satisfying, both are recurring and plausible interpretations of Marx, and each seems incompatible with the other. Perhaps the union of opposites offers a more productive view of what in Marx is staged as an alternation: a phenomenon like capitalism is good and bad all at once and simultaneously – the most productive as well as the most destructive force we have so far encountered in human history, as the *Manifesto* puts it.

The class, gender, and race critiques of technology, technological determinism, and a gleeful, unquestioning rush to the latest technologies, are all valid. At the same time, the possibilities for sharing, collaboration, agency, investment and commitment to work, the possibilities of overcoming alienation however briefly and however - in the end - fruitlessly, all of these are part of becoming more involved in the technological landscape of our professions. Is coding the only way of becoming more involved? No, of course not, but I feel it has the lowest barrier and a high rate of return.

What does this have to do with pedagogy? If we follow Freire and define praxis as "reflection and action directed at the structures to be transformed", then learning to code / teaching to code is important in a number of respects. In terms of the structures to be transformed, there is on the one hand, the structure of private property, challenged by participation in cultures of openness, sharing, and collaboration that comes with writing code and engaging in open-source projects; then there are the structures of gender, race, and class dominance that have traditionally been part of the culture of software engineering (even when that tradition is in fact, incorrect – as in the denial of the role women played in the first century of computing) there are structures of technologism and technological determinism that, following Marx, any critique of capitalism has to deconstruct; and there are finally structures within our professions. In librarianship, there are structural relationships between libraries and vendors, between hierarchical levels of administration, and between what I will call neoliberal and progressive ideologies. And these structures are challenged both by one's learning, and the social networks that arise through that practice (I think of communities like **#codenewbies, exercism.io, #critlib/#mashcat**, etc), but also through informal teaching – sharing information through hackfests, unconferences, and pair programming, and the formal process of teaching (which, in librarianship, tends to be concentrated in Information Literacy sessions and Library and Information Science programs). In this respect, "coding" becomes a bit of a McGuffin. The semantic content is unimportant – what is important is the activities and relationships that arise through the process of learning and acting, no matter what the object is. Coding is low barrier and has developed pedagogical tools, systems, and networks over the last few years which unix system administration, for example, has not. You could say that, in essence, I think librarians should learn code because Ladies Learning Code exists.

In conclusion, I think it's safe to say that "learning to code" can't happen in isolation. It has to be a social activity, it has to compete, in some sense, with what Marx calls the "nexus between man and man" that is “naked self-interest”. There are many other skills that humanists could adopt, there are many other arenas in which capitalist technology might be challenged, there are many other methods by which a critical pedagogy might be adopted, but learning to code seems, to me, at this historical moment, to lie at the intersection of all these considerations.