Programming as Architecture, Design, and Urban Planning

Tomas Petricek
School of Computing, University of Kent, UK
tomas@tomasp.net

ABSTRACT

Our thinking about software is shaped by basic assumptions and metaphors that we rarely question. Computer science has the term science in its very name; we think of programming languages as formal mathematical objects and we hope to make better software by treating it as an engineering discipline. Those perspectives enabled a wide range of useful developments, but I believe they have outlived their usefulness. We need new ways of thinking about software that are able to cope with ill-defined problems and the increasing complexity of software. In this essay, I draw a parallel between the world of software and the world of architecture, design and urban planning. I hope to convince the reader that this is a well-justified parallel and I point to a number of discussions in architecture, design and urban planning from which the software world could learn. What kind of software may we be able to build if we think of programming as a design problem and aim to create navigable and habitable software for all its users?

CCS CONCEPTS

• Software and its engineering → Software creation and management → Designing software.

KEYWORDS

Software design, Architecture, Design, Urban Planning

ACM Reference format:

Tomas Petricek. 2021. Programming as Architecture, Design and Urban Planning. In *Proceeding of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2021). ACM, New York, NY, USA, 12 pages.* https://doi.org/10.1145/3486607.3486770

Onward! '21, October 20–22, 2021, Chicago, IL, USA

ACM 978-1-4503-9110-8/21/10…$15.00
https://doi.org/10.1145/3486607.3486770

Formal order (…) is always and to some considerable degree parasitic on informal processes, which the formal scheme does not recognize, without which it could not exist, and which it alone cannot create or maintain.

James C. Scott, Seeing Like a State (1998)

1 History of Software Metaphors

Our thinking is shaped by basic assumptions that we rarely question. In the context of science, *research paradigms*[1] determine what are the legitimate research questions and what are the scientific methods that should be employed for studying them. In the context of software development, a similar role is played by metaphors that we use for talking about programming and software. Metaphors are apparent in the language we use. The term *language* in programming language is an allusion to both natural language and formal languages in logic[2]. When we talk about software *engineering* or software *maintenance*, we are using terminology deliberately adopted to treat software development as an engineering discipline. But are those metaphors inevitable? How could programming look if it were shaped by different metaphors?

1.1 Programming Languages and Software Engineering

There are at least two metaphors that shape our thinking about programming and software development that are so ubiquitous that we often forget they exist. The first views programming languages as formal languages in a sense derived from formal logic. The second views software development as an engineering discipline. Both of these are a result of historical developments that could have gone differently.

At the beginning of 1950s, computers were programmed in idiosyncratic machine codes. Early compilers were able to translate higher-level pseudo-code to low-level machine code, but the pseudo-code and the interpretative routine doing the translation were still machine-specific[3]. We can see the linguistic metaphor slowly appearing in the language through the use of words such as "translation", but this was initially inspired by cybernetics, which saw computers as human-like agents and compilers as "translating code to a language a machine can understand."[4] The term language started to be used in late 1950s and the 1957 FORTRAN manual talks about the "FORTRAN language",

[1] Kuhn (1962)
[2] Lakoff and Johnson (2008)
[3] For example, see Hopper (1955)
[4] For a detailed historical account, see Nofre et al. (2014)

even though the language is still not a stand-alone object, independent from the machine. The adoption of the language metaphor was completed with the 1958 report on the International Algebraic Language (IAL). The report presented IAL (later ALGOL 58) as a stand-alone mathematical object, independent from any specific machine, using a formal language of the Backus normal form (BNF) notation inspired by work on formal languages in logic. The birth of the idea of a programming language served the academic community by giving them an object that could be analyzed using rigorous mathematical methods[5]. It was also motivated by commercial needs, because it detached programs from specific machines and allowed (at least some form of) program portability. The metaphor provided computer science with a powerful perspective and it made it possible to think about questions such as language syntax and semantics. It made possible the very field of theoretical programming language research.

The second prominent metaphor for thinking about software development is often associated with the 1968 NATO Software Engineering Conference. The conference was motivated by growing concerns about managing large software projects and the new metaphor for thinking about software development was embedded in its very name. It aimed to turn the "black art of programming" of 1950s and 1960s into a new, soon to be developed, "science of software engineering". The follow-up conference held in 1969 showed that there was much disagreement about what exactly the science of software engineering should be, but the new metaphor stuck. Management found itself a new domain to transform and applied its preexistent process models to it. Software development started to be treated as a structured activity with independent phases such as software design, development, testing and maintenance. It also shifted focus to issues such as production, reliability, requirements gathering or management.[6]

1.2 Software as Architecture, Design and Urban Planning

Despite the many virtuous developments enabled by the language and engineering metaphors, they force us to accept simplifying assumptions that are not (always) true about programming. This is inevitable and any research paradigm or a metaphor has to do so. At the same time, it is valuable to be aware of those assumptions and investigate whether another metaphor would be more suitable, especially as the nature and practice of software development keeps evolving and changing.

The language metaphor lets us treat programs as entities of formal logic, but terms that are typically analyzed in formal logic are smaller than even the simplest computer programs and much smaller than the code behind large software systems. Consequently, the nature of proofs about programs is different than the nature of proofs in formal logic, which has been a source of a program verification controversies in the 1970s and 1980s. The quantitative difference in size gives rise to a qualitative difference and so the metaphor may not be suitable for thinking about large and complex software systems. The engineering metaphor lets us focus on the right process for building software, but it is largely inspired by the kind of engineering used when solving well-defined problems. Unfortunately, many problems that software developers face are ill-defined, change during the development and are such that their framing often determines the solution[7].

What metaphor should we use if we want to talk about software development that involves large complex systems that address ill-defined problems? In this essay, I suggest framing the problems of programming and software development in terms of ideas borrowed from architecture, design and urban planning. I hope to convince the reader that this perspective is at least as productive as using ideas from logic and engineering.

I am, of course, not the first to suggest linking software development with architecture, design and urban planning. The idea of design patterns, popular in object-oriented programming, was inspired by the work of Christopher Alexander,[8] although a deeper look at the work of Christopher Alexander suggests that the standard software design patterns are a trivialized and not very useful version of the idea[9]. A number of people also acknowledged the role of design in software engineering. Just like actual engineers sometimes solve ill-defined design problems,[10] software engineers also do so, at least in the early phase of the software engineering process and researchers studied how the design aspects of software development are done in practice,[11] reflect on the design work[12] and propose new approaches[13].

My goal in this essay is to convince the reader that there are good grounds for using architecture, design and urban planning as a source of ideas for software development. I will look at a number of aspects where there is a striking similarity between issues discussed in the world of architecture, design and urban planning and the world of software. The responses to those issues in the context of architecture, design and urban planning may well prove a useful inspiration for work that we need to do in the context of software.

> *We may wish for easier, all-purpose analyses, and for simpler, magical all-purpose cures, but wishing cannot change these problems into simpler matters (...) no matter how much we try to evade the realities and to handle them as something different.*
>
> *Jane Jacobs, The Death and Life of Great American Cities* (1961)

[5] For a detailed historical account, see Priestley (2012)

[6] Apparent in the NATO 1968 Conference Proceedings headings (Naur and Randell, 1969)

[7] This kind of problems is also known as "Wicked problems" (Rittel and Webber, 1973)

[8] Reference to Alexander's work (Notes on the Synthesis of Form) appears as early as 1968 in the NATO 1968 Conference Proceedings; initial work adapting the concept of design patterns is by Beck and Cunningham (1987), which is quite different than the widely known later work by Gamma et al. (1995).

[9] For a more critical perspective, see Gabriel (1996).

[10] My understanding of actual engineering work follows Vincenti (1990)

[11] Baker (2010)

[12] Petre and Van Der Hoek (2019)

[13] Jackson (2015) and Kaijanaho (2017)

2 The Nature of Software

Software has been likened to a wide range of other human activities and research fields including cooking, writing, gardening and the study of biological eco-systems.[14] The reader can surely imagine other possible analogies. What makes architecture, design and urban planning a better perspective? I believe the answer is that those disciplines, like software development, deal with complex systems and ill-defined problems.

2.1 Designerly Ways of Knowing Software

Design is a very broad area and many software practitioners already think of *software design* as being a design activity. Making the connection more explicitly is still useful. Doing so reveals the assumptions that we are making about software if we treat it as a design problem.

In his analysis of "designerly ways of knowing,"[15] Nigel Cross views design as a third culture of human knowledge, complementing those of science and humanities. He contrasts the three by looking at what each study, what their methods are and what each culture values:

- *Sciences* study the natural world; using controlled experiments, classification and analysis; aiming for neutrality and the 'truth'.
- *Humanities* study the human experience; using analogies and metaphors; aiming for subjectivity, imagination and a 'justice'.
- *Design* studies the artificial world; using modelling and synthesis; aiming for practicality, ingenuity and 'appropriateness'.

The domain in which computer programs belong in this classification is clear. Programs are a prime example of the artificial,[16] modelling is a major concern of many software development methodologies,[17] and software is often built from smaller pieces through the method of synthesis. Like designers, software developers rarely worry about truth or justice, but rather focus on finding developing appropriate solutions.

As discussed later, viewing software as a design problem makes us realize that we are often dealing with ill-defined problems that have very diverse but still appropriate solutions. It also suggests design-inspired approaches to solving those problems. The classification by Cross is also interesting in that the design category would likely encompass both traditional engineering and also mathematics. This suggests that the shift from thinking about programming as engineering or as mathematics to thinking about programming as design may not be as dramatic as it may at first seem.

2.2 What Kind of Problem Software Is

Drawing an analogy between software and urban planning is certainly a less obvious move than likening software to design, but it will prove revealing. Like programmers, urban planners often deal with very complex problems that are difficult to reduce and simplify.

The point has been made very clearly by Jane Jacobs, who studies the complex and subtle ways in which large American cities work.[18] To answer the question "what kind of problem city is", Jacobs refers to an essay on science and complexity,[19] which looks at three stages of development in the history of scientific thought, based on the kinds of problems that science was able to tackle:

- *Problems of simplicity* are problems with small number of variables that admit a precise analytical solution, such as how a gas pressure depends on the volume of the gas.
- *Problems of disorganized complexity* have a large number of variables, such as laws of thermodynamics derived from statistical analysis of the motion of atoms. The behavior is sufficiently random, making it possible to get useful insights using statistics and probability.
- *Problems of organized complexity* are the most challenging ones. They involve complex structures that cannot be abstracted away using statistics. For example, how is the genetic code reflected in the characteristics of a developed organism?

According to Jacobs, urban planning is a problem of organized complexity and many issues with earlier theories follow from the fact that urban planners treat them as problems of simplicity or disorganized complexity. Urban planning involves many different problems, interconnected through a large number of variables. For example, how a park is used depends on how it is designed, but also who lives around it and what businesses exist in the neighborhood, which, in turn, depends on the size of blocks and the age of buildings in the surroundings. To learn anything useful about a park, you have to study this highly sophisticated network of factors in its full complexity.

Like cities, software systems are problems of organized complexity. They involve a large number of variables and processes that influence each other in subtle ways. David Parnas captured this well in his 1985 critique of the Strategic Software Defense Initiative[20]. To explain the difficulties with producing reliable software, he presents a categorization of computer systems that is remarkably similar to that of Weiner:

- *Analog systems* can be modelled as continuous functions. This means that they can contain no hidden surprises. A small change in the input will cause a correspondingly small change in the output.
- *Repetitive digital systems* such as a CPU have a very large number of states, but they consist of many copies of small subsystems that can be analyzed and tested exhaustively.
- *Non-repetitive digital systems* such as software systems cannot be modelled as continuous functions and have a very large number of states that cannot be exhaustively analyzed and tested.

The two categorizations are remarkably similar. Problems of simplicity and analog systems can be understood in full. Problems of disorganized complexity and repetitive digital systems can be reduced, either using

[14] For cooking, see e.g., a quote by Grace Hopper in April 1967 issue of the Cosmopolitan magazine (Mandel, 1967). For writing, gardening and biological systems, see e.g., Hermans and Aldewereld (2017), Papapetrou (2015) and Northrop et al. (2006)

[15] Cross (2007)

[16] This view is advocated, for example, by Simon (1969)

[17] For example, see work on Domain-driven design such as Evans and Evans (2004)

[18] Jacobs (1961)

[19] Weaver (1958)

[20] Parnas (1985)

statistical methods or using logic. But problems of organized complexity and non-repetitive digital systems are both too large to be analyzed in full and cannot be reduced to smaller problems. As Jacobs puts it, "*the large number of interrelated variables form an organic whole*".

Programming language and systems researchers have done a fair amount of work towards being able to analyze complex systems with increasing number of states, mostly by building tools that exploit various heuristics to reduce the number of states.[21] However, the number of states in non-repetitive digital systems grows faster than the size of the system and our tools are bound to be limited. Urban planners never attempted to model every little detail about how cities work, yet, they have learned valuable knowledge about cities. Perhaps there is something to be learned from them.

2.3 Structures Obtained by Gradual Development

In engineering, the right way to construct a structure is to correctly design it and then build it according to the plan. This is the case for most architecture designed by architects too[22], but not all architects agree that this leads to the best results. A notable dissenting voice is Christopher Alexander. In an interview with Stewart Brand, he argues that[23]:

> *Things that are good have a certain kind of structure. You can't get that structure except dynamically. Period. In nature you've got continuous very-small-feedback-loop adaptation going on, which is why things get to be harmonious. That's why they have the qualities that we value.*

Brand develops the idea further and suggests that many great buildings achieved their greatness by gradual stepwise evolution over time. New buildings need to be designed so that they can evolve when they outlive their initial use or when the needs of their users change. This should be done, for example, by making sure that changing the space layout in the building is possible without changing its structure.

In the world of software, agile methodologies treat software development as an evolutionary process,[24] but the key point has already been made in 1969 by Joseph Weizenbaum when discussing the feasibility of designing software for an anti-ballistic missile (ABM) system. Weizenbaum argues that "*large computing systems are products of evolutionary development*" and that they only become reliable through a process of slow testing and gradual adaptation to an operational environment.[25] This means that building an ABM system is infeasible, because the environment with which the software interfaces evolve at a faster rate than the rate at which the software can be adapted.

The idea that buildings achieve greatness by evolution may be controversial, but evolutionary metaphors are commonplace in urban planning.[26] Perhaps we could use vernacular architecture and evolutionary urban planning to learn how to build adaptable software.

> *The pseudoscience of city planning and its companion, the art of city design, have not yet broken with the specious comfort of wishes, familiar superstitions, oversimplifications, and symbols, and have not yet embarked upon the adventure of probing the real world.*
>
> *Jane Jacobs, The Death and Life of Great American Cities* (1961)

3 Beautiful Theories

Jane Jacobs' work on urban planning is a critique of utopian planning theories of the early 20th century including Le Corbusier's *Ville Radieuse* and Ebenezer Howard's *Garden City* movement. The theories were based on simple and rational principles, supported by reasonable arguments. Cities should be legible, spacious with many green areas, organized by function and support effective transportation. The only issue is, as Jacobs and others document, that such theories do not work. They treat cities as a problem of simplicity or disorganized complexity and, consequently produce cities that lack the vital complex processes that make a city a lively and attractive place for living.

Software development and computer science abound with beautiful rational theories. We uncritically praise abstraction even when it leads to failures,[27] we religiously follow principles such as information hiding even if that hinders long-term maintainability.[28] If we want to design a programming language with a well-defined behavior, we define a small formal model that captures the essential properties of the language. Except that it often turns out that the omitted non-essential aspects were equally important for the usability of the language.[29]

There are a number of methods and ideas, developed by designers and urban planners, that the world of software could explore.

3.1 What Works Despite Theory

Jane Jacobs' critique of utopian theories of urban planning is so powerful, because it gives a detailed account of specific city districts that are safe, attractive and lively, yet should not work at all according to the utopian theories and were, in fact, candidates for being torn down.

Her two examples are Greenwich Village in New York and North End in Boston in 1950s. Both have little green space, are dense and feature a messy mix of housing, shops and entertainment venues. To understand how the district works, Jacobs looks at the details. Thanks to its mixed-use buildings, there is an active sidewalk life, which provides informal safety; a combination of older and newer buildings allows people who start earning more money stay in the neighborhood and uplift it. The details add up to a complex, but functioning whole.

---

[21] A prime example of this line of work is the ongoing research done on the Z3 SMT solver (De Moura and Bjørner, 2008)

[22] A counter-example has been the MoMA 1964-67 exhibition "Architecture without Architects" and the accompanying book Rudofsky (1987)

[23] Brand (1994)

[24] This has been observed by Christian (2003)

[25] Quoted in Slayton (2013)

[26] For example, see a review by Mehmood (2010)

[27] The case of abstraction is studied in a recent Onward! Essay by Steimann (2018)

[28] A point made by Clark and Basman (2017)

[29] As pointed out in private communication by Jeremy Gibbons, there is a difference here in that utopian models in urban planning were intended to be used for actual building whereas computer scientists use them merely for analysis.

The world of programming is full of systems that work remarkably well, despite being completely wrong according to our utopian theories. Popular programming environments like PHP, JavaScript and R are the most obvious examples. There are some attempts to explain why[30], but we mostly just disregard those saying that they got popular by accident[31]. For a community that prides itself in being thorough and scientific, this is a very shallow argument. To quote Peter Naur:[32]

> *It is curious to observe how the authors in this field, who in the formal aspects of their work require painstaking demonstration and proof, in the informal aspects are satisfied with subjective claims that have not the slightest support, neither in argument nor in verifiable evidence. Surely common sense will indicate that such a manner is scientifically unacceptable.*

Following Jacobs, we should study existing programming environments that work despite theory. This work will need to be partly technical and partly sociological, because technical characteristics often enable certain ways of using a system. For example, during the 1990s era of JavaScript, many learned how to program by copying other people's rollover or mouse-follow effects. This would not be possible if the language was compiled or if the relevant code was not easy to isolate and copy.

This kind of research is being done in human-computer interaction, for example, looking at the specifics of how an eco-farming community in Aarhus used an evolving range of computing systems and tools over a number of years to manage their work.[33] Looking at programming systems through this perspective is rarer, but Colin Clark and Antranig Basman make the first step by documenting how the MIDI interface achieved longevity precisely because it implemented no form of information hiding.[34]

3.2 Seeking for Unaverage Clues

Jane Jacobs criticizes utopian theories, because they treat cities as problems of unorganized complexity and assume they can be reduced and understood using statistical methods:[35]

> *In the form of statistics, [citizens] could be dealt with intellectually like grains of sand, or electrons, or billiard balls. (…) It became possible to map out master plans for the statistical city, and people take these more seriously, for we are all accustomed to believe that maps and reality are necessarily related, or if they are not, we can make them so by altering reality.*

In the world of software, logic has been a powerful tool for reducing accidental complexity, i.e., that which exists primarily inside the computer and is relatively isolated from the outside world. However, most complexity in software is caused by the outside world, as pointed out by Fred Brooks in his famous "No Silver Bullet" essay:[36]

> *Much of the complexity a [software engineer] must master is arbitrary complexity, forced with-out rhyme or reason by the many human institutions and systems to which [their] interfaces must conform. These differ from interface to interface, and from time to time, not because of necessity but only because they were designed by different people (…).*

I suspect that computer scientists often think about accidental complexity in software and, based on this experience, come to believe that all software complexity is reducible. But this is not the case with the organized (essential) complexity imposed by the outer world.

If software systems are largely systems of unorganized complexity, what can we do to understand them? Let's see what Jacobs recommends for understanding other such systems:[37]

> *In the case of understanding cities, I think the most important habits of thought are these: (1) to think about processes; (2) to work inductively; (3) to seek for 'unaverage' clues involving very small quantities, which reveal the way larger and more 'average' quantities are operating.*

I believe the interesting methodological point is the call to look for 'unaverage' clues. As an example, Jacobs describes a chain of five bookshops. Four stay open until 10pm or midnight, but the one in Brooklyn downtown closes at 8pm. The management clearly keeps its stores open if there is any business to be had, which gives us a clear sign that downtown Brooklyn is deserted by 8pm, a valuable insight for an urban planner.

When studying programming languages, we often attempt to reduce them to their essence, such as a simple formal calculus. We then end up looking at a simplified version of the problem that eliminates interesting unaverage properties. Instead of trying to prove universal properties about such simple models, we should be looking for unique cases that illustrate something interesting about the system. On the formal side, a counter-example program showing that a type system is unsound is exactly this,[38] but we should take a more non-reductionist point of view and document interesting examples, applications or use-cases that show, for example, how and why a particular programming language works (or could work). The 10 PRINT CHR$(205.5+RND(1)); : GOTO 10[39] book does this by taking a well-known BASIC program as a starting point, but using it to discuss broader range of technical and cultural issues.

3.3 Treating All Problems as Ill-defined

Jane Jacobs helps us understand how we should study systems. Her work also offers some points about better city design, but those are largely specific to the problem of cities. If we abandon appealing, but wrong, utopian theories, how should we approach software design?

This is where we can learn from professional designers. First of all, most problems that designers face are ill-defined or wicked.[40] The

[30] For example, Meyerovich and Rabkin (2012) study sociology of language adoption
[31] For example, Ghica (2016).
[32] Naur (1992)
[33] Bødker et al. (2016)
[34] Clark and Basman (2017)
[35] Jacobs (1961)
[36] Brooks (1995)
[37] Jacobs (1961)
[38] This has been shown for Java and Scala by Amin and Tate (2016)
[39] Montfort et al. (2014)
[40] Rittel and Webber (1973)

designer does not (and cannot) have access to all relevant information, there is no clear success metric and exhaustive analysis of such problems is not possible. Many problems in software are equally ill-defined, especially if we do not see them from a purely technical perspective, but consider them in the actual context in which it will be used.[41]

According to Cross, designers follow a solution-oriented approach to problem solving. Rather than starting with a detailed analysis, they quickly iterate on a number of possible solutions and explore the problem through the perspectives offered by those solutions. Seeing a problem in this way lets designers reframe (and even change) it. Good designers are distinguished by their ability to come up with a strong framing, utilizing their past experience and or a range of theoretical principles. However, perhaps the most interesting observation by Cross is that designers approach all problems they face as ill-defined. Even when solving a problem that can be treated as well-defined, designers still proceed by changing the problem goals and constraints.

The engineering-inspired approach to software development starts with a specification. It implicitly assumes that the problem it faces is well-defined and can have an exhaustive description. If we start treating problems as ill-defined, we instead need a broader project brief that explains the problem together with its context, but does not limit the space of possible solutions.

Agile development methodologies[42] already eliminated some of the up-front planning in software development, but a more fundamental shift in our thinking is needed before we start treating all problems as ill-defined and, hopefully, use software design not just as a way of building software, but as rethinking problems we are solving. For this, we will also need the equivalent of designer's sketches, i.e., a quick way to see what the solution would look like in order to be able to explore its consequences.[43]

> *The fact that the layout of the city [like Bruges], having developed without any overall design, lacks a consistent geometric logic does not mean that it was at all confusing to its inhabitants.*
>
> *James C. Scott, Seeing Like a State* (1998)

4 Conceptual Coherence

Let's now shift our attention from the outside perspective of planning a city or software to the inside perspective of navigating through or making sense of a city or software. Navigability and understandability of a city allows its inhabitants to use it well. Similarly, a codebase needs to be understandable and navigable if it is to be modified, extended or used by other programmers.

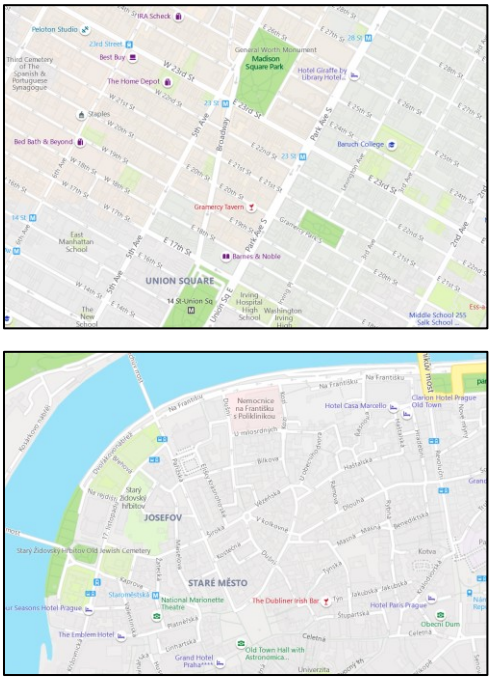

Figure 1. Two cities – Manhattan, New York (above) and Prague (below) (source: Bing Maps: https://www.bing.com/maps)

Note that navigability of a city is equally important for its designers and users whereas navigability of a codebase is only a concern for the programmers. This may seem like a flaw in the analogy, but perhaps it also points to new possibilities. What if we built software so that it is open and navigable to its users, as well as to its programmers?

The issue of developing understandable software has been discussed at length by Fred Brooks in his essays.[44] Brooks argued that the most important characteristic that determines understandability of a software system is conceptual coherence. In a conceptually coherent system, every part reflects the same design philosophies and the same balancing of forces. Brooks suggests that conceptual coherence is best achieved if a system is designed by a single person, but this does not scale to very large systems:

> *Conceptual integrity (..) dictates that the design must proceed from one mind, or from a very small number of agreeing resonant minds.*
>
> *Any product that is sufficiently big (...) must be conceptually coherent to the single mind of the user and at the same time designed by many minds.*

Brooks suggests a number of methods for achieving conceptual coherence, such as separating architecture from implementation or a team structure where a small group of individuals is responsible for the conceptual design. But are navigable and understandable cities conceptually coherent and produced by a single mind with clear vision?

[41] This is a perspective adopted by the analysis of Ultra-large-scale systems by Northrop et al. (2006)

[42] The perspective has been outlined by Floyd (1987), but has later been popularized by the Agile Manifesto published by Beck et al. (2001)

[43] For user interface design, this is achieved by wireframing tools, but what I propose here is more focused on functionality and interaction than just (static) user interfaces.

[44] Brooks (1995)

4.1 Two Kinds of Cities

Cities exist in a variety of forms. As Figure 1 shows, some are visibly more conceptually coherent than others. Manhattan got much of its structure from an early 20th century plan and is a product of a small number of minds. Prague developed organically since its founding in 7th century and although some of its parts have been rebuilt with structure in mind, it follows no overall plan. But as Scott observes,[45] the fact that a city developed without any overall design does not mean that it has to be confusing to its inhabitants. A city that lacks conceptual integrity can still be understandable to its inhabitants, but it will be illegible to outsiders. In words of Scott, "it *privileges local knowledge over outside knowledge*". The two kinds of knowledge play a very different role. While local knowledge is used by the inhabitants living in the city to move around, the outside knowledge is used by the outside authorities, strangers and the military. It lets the government effectively plan public services such as transportation, garbage collection and, in the past, tax collection. But it also allows the military to move more smoothly through the city. To quote Scott again, "*the relative illegibility of some urban neighborhoods has provided a vital margin of political safety from control by outside elites*".

In the case of cities, conceptual coherence only serves certain users and purposes. The same seems to be the case for software. Moreover, software systems such as large open-source ecosystems are very unlikely to achieve conceptual coherence, because they are simply a product of too many minds. Perhaps cities can teach us how to make such systems understandable, despite the lack of conceptual coherence.

4.2 The Image of the City

A perfect starting point for looking at how inhabitants understand the cities where they live is a study by Kevin Lynch.[46] A good city needs to be legible to its inhabitants, but this does not need to be achieved through a conceptually coherent master plan. Legibility is the result of interactions between a number of aspects of a city. A legible city is one *whose districts or landmarks or pathways are easily identifiable and are easily grouped into an overall pattern*. Software developers should strive to produce software that is legible, both to its users and its developers and future contributors. My primary focus here is on the legibility of a codebase, but I believe many ideas will be equally relevant to legible design for users of a system.

To study legibility, *we must consider not just the city as a thing in itself, but the city being perceived by its inhabitants*. Lynch looks at how people navigate around three cities, Boston, New Jersey and Los Angeles, and identifies a number of aspects that are important for its legibility, such as paths, districts and landmarks. The interactions between those allow inhabitants to navigate around a city. For example, the inhabitant may be able to identify a district from the characteristic features of its buildings (e.g., red bricks) and find their way from the district by following a path towards a visible landmark (such as an easily identifiable church tower).

Figure 2. Prague metro map – an example of a structure that makes a city legible (source: https://pid.cz/ke-stazeni/?type=mapy)

Although concepts such as paths, districts and landmarks are notions from urban planning, we can easily imagine similar ideas in the context of software systems. There are multiple types of paths that one might follow through software. One path may be the execution order while another may be based on data dependencies. Some paths may be disconnected, such as when looking for all references to a given definition. Software may also have different districts if it consists of multiple components. Contrary to established wisdom, inconsistent coding styles may be, in fact, useful as one indicator that can be used for identifying code district in which a programmer finds themselves. The programmer may then understand that they are in a "bad neighborhood" where any code change is likely to break the system.

There are two points in Lynch' analysis that may be particularly relevant to making sense of software. The first is that more knowledgeable people typically rely on *paths* whereas visitors tend to rely more on districts. Even if the low-level structure is illegible to outsiders, the high-level structure of districts can provide a basic guide for navigation. This may provide a valuable alternative to utopian ideas about building of software. We may not need to enforce strict separation of concerns or structure our software into independent layers as long as our programming system provides enough hints about the districts in which code lives (to let newcomers find their place in a codebase) and makes it easy to follow the paths that exist in the software (to allow experts to efficiently move around).

The second point is that there are often multiple overlapping images of a city. One may be provided by the layout of districts, while another may be derived from a particular path. Figure 2 shows one such example – a map of the Prague metro. A coherent city or software makes it easier to keep a full image of the system in mind, because it can be efficiently abstracted and compressed. But you do not need to keep a full image of the system in mind, as long as there is a basic structure, such as the path defined by the public transport, from which you can start when you need to make sense of a particular new part.

[45] Scott (1999)

[46] The Image of the City by Lynch (1960)

A range of observers of architecture are now suggesting that the field may be bankrupt (...) the methods inapplicable to contemporary design tasks [and that] collectively they are incapable of producing pleasant, liveable and humane environments, except perhaps occasionally and then only by chance.

C. Thomas Mitchell, Redefining Designing (1997)

5 Adaptable Software

Is there an alternative to software development based on utopian theories or unattainable ideals of conceptual coherence? I believe my essay shares this question with some of the work of Richard P. Gabriel,[47] who was himself inspired by the architect Christopher Alexander. Gabriel argues that software should be habitable. In architecture, *habitability makes a place livable, like home*. In the case of software:

> *Habitability is the characteristic of source code that enables programmers, coders, bug-fixers, and people coming to the code later in its life to understand its construction and intentions and to change it comfortably and confidently.*

The ideas on how we navigate around cities and around software that I discussed in the previous section are closely related to habitability. A habitable city or a software does not need to be conceptually coherent, but it must be navigable and understandable.

However, there is more to habitability than just understanding. Habitability is also about being able to change the system. This idea is equally important to urban planners, enlightened architects and software developers. In particular, Jane Jacobs, who wrote about cities that work despite not being designed according to utopian theories also writes about the conditions that enable inhabitants uplift their city districts, while Stewart Brand writes about how inhabitants adapt their buildings.

As Gabriel points out, *a New England farmhouse is habitable and [a] new owner feels just as comfortable changing or adapting that farmhouse as the first farmer was*. But how do we achieve this for software?[48]

> *[H]ow do you enable a programmer to feel responsible for software developed earlier? Here is where habitability comes in. Just as with a house, you don't have to have built or designed something to feel at home in it. Most people buy houses that have been built and designed by someone else.*

If we want to build long-lasting software that adapts to changing requirements and contexts and improves over time, we need to make sure it is habitable and adaptable. I will first look at the case of buildings, before exploring what ideas can be relevant in the context of software.

5.1 How Buildings Learn

Christopher Alexander[49] distinguishes between *unself-conscious design* that achieves a good fit between context and form through gradual

Figure 3. Barn window with wood siding. An example of a material that "looks bad before it goes bad" (source: https://pxhere.com/en/photo/569328)

adaptation and *self-conscious* design that aims to achieve theoretical understanding of the complexity of the system and design a solution.

The invention of architecture, as a self-conscious design method, destroyed the old process of building, but it has not always been for the better. Adaptability is one of the aspects that self-conscious architecture often gets wrong. Stewart Brand offers a damning summary:[50]

> *Almost no buildings adapt well. They're designed not to adapt; also budgeted and financed not to, constructed not to, administered not to, maintained not to, regulated and taxed not to, even remodelled not to. But all buildings (...) adapt anyway, however poorly, because the usages in and around them are changing constantly.*

The same could be said about software. Software is often designed with the implicit assumption that it won't need to be modified once it is complete, yet, the context evolves and software needs to evolve too.

Brand does not give simple advice on how to design an adaptable building, but he starts from an interesting analysis. He documents how different types of buildings evolved over time, distinguishing two kinds of buildings: *Low Road* buildings, such as a former warehouse, are flexible, cheap to modify and can easily adapt to a very different purpose. *High Road* buildings, such as an English manor house, adapt slowly with more respect to their history, are more expensive to maintain, but they develop a unique character.

A similar distinction might exist in the world of software. On the one hand, some software provides a minimal robust core structure and can be easily adapted and modified within this structure. On the other hand, there are software systems that evolved more slowly, have longer history that they have to respect and are more expensive to maintain, but can reliably provide services that are complex and cannot be easily replaced.

More generally, Brand's work suggests that we have much to learn about software by undertaking a detailed analysis of past software systems. When talking about utopian urban planning theories, I pointed out that we should follow the example of Jane Jacobs and document software systems that work well despite theory. Similarly, we should

[47] Especially essays collected in Patterns of Software (Gabriel 1996)
[48] Gabriel (1996)
[49] In Notes on the Synthesis of Form (Alexander, 1964)
[50] Brand (1995)

follow the example of Stewart Brand and document how software systems evolve over time. Only then we can meaningfully start looking for various patterns in such evolutions and use these to design more adaptable software systems.

5.2 Maintenance and Materials

Until a thorough analysis of the evolution of software is completed, the best we can do is to see whether there are any specific ideas in the world of architecture that could be equally relevant to the design of adaptable software. I believe there are two such ideas in Stewart Brand's writing. The two ideas focus on *maintenance* practices and building *materials*.

Just like software systems, any building requires maintenance over time. And just like with software systems, building owners are often bad at performing the necessary maintenance:[51]

> *Too often a new building is a teacher of bad maintenance habits. After the initial shakedown period, everything pretty much works, and the owner and inhabitants gratefully stop paying attention to the place. Once attention is deferred, deferring of maintenance comes naturally.*

Brand's answer is to design buildings so that they teach good maintenance habits. One way to do this is to make some parts of the initial design intentionally ephemeral. If there are parts that will require maintenance within a year, the owners will get into a good habit that will be necessary once the building is older. The same seems to be a very good suggestion for building software systems. If we build our systems in a way that intentionally makes some parts degrade more quickly, we will establish the right methods and processes for maintenance that will be valuable in the long run. One step in this direction is *chaos engineering*,[52] which improves system resilience by intentionally disabling random services, in order to ensure that the rest of the system recovers from such failures. This focuses on operational aspects of distributed systems, but perhaps we can imagine a form of chaos engineering for other aspects of the software development process?

The maintainability of a building (and software) also crucially depends on the material from which it is built. Brand mentions the cautionary tale of vinyl siding, which is often used to cover wooden walls with peeling paint. The problem is that vinyl siding blocks moisture and the humidity behind it can cause structural damage to the building. The peeling paint on a wooden wall, illustrated in Figure 3, is a desirable property of the material:[53]

> *The question is this: do you want material that looks bad before it acts bad, like shingles or clapboard, or one that acts bad long before it looks bad, like vinyl siding?*

The case poses an interesting question about the materials we use to build software. What is the software equivalent of a material that looks bad before it acts bad? How can we build software such that it gracefully degrades rather than abruptly stops working?

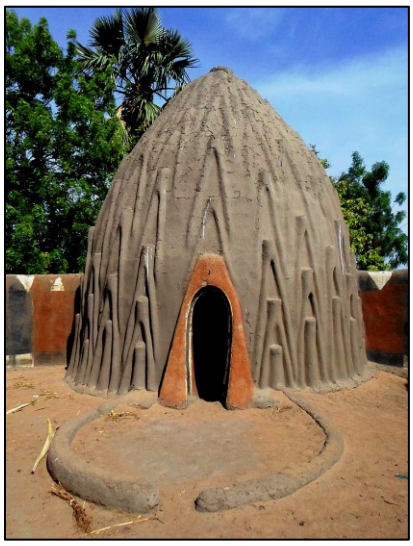

Figure 4. Musgum mud hut. An example of well-adapted vernacular architecture (source: https://en.wikipedia.org/wiki/Musgum_mud_huts)

F# type providers for accessing external data[54] may be one step in this direction. A type provider may generate types based on an external schema, making type safety conditional on the external world. When the external data source changes (the environment degrades), the programmer will get a compile-time error (material looks bad) before a runtime error occurs (material goes bad).

5.3 Vernacular Architecture

Concerns like materials and maintenance habits are important for self-conscious architecture where the architect first analyzes the problem to understand its requirements, such as adaptability, and then proposes a design that achieves those objectives.

A very different process for obtaining buildings with desirable properties takes place in the context of unself-conscious or *vernacular* architecture, which broadly refers to buildings built by non-architects. Vernacular architecture achieves desirable properties without the same theoretical understanding of the problem. Instead, it works by gradual step-wise adaptation of a design over longer period of time. Crucially, vernacular architecture works without reinventing the architectural form of a building.

A New England farmhouse, used earlier as an example of a habitable building by Gabriel is one case of vernacular architecture. When building a new farmhouse, the farmers do not invent a new form from scratch. Instead, they mostly follow the structure of other farmhouses, but make small adaptations based on recent experience. The use of existing structure ensures that the new building will work; small adaptations ensure that the design improves over time. Another

[51] dtto.
[52] First developed at Netflix by Basiri et al. (2016)
[53] Quoted from Brand (1995)
[54] Petricek et al. (2016)

example, mentioned by Christopher Alexander[55] are traditional mud huts built by Musgum people. Despite developing without theoretical foundations, the huts use a mathematically ideal catenary arch and are extremely good at keeping houses cool inside on hot summer days.

Vernacular design restricts the scope of the problem by limiting architectural ideas to what is typically used in the local context. This reduces the design task and allows the builder to focus on skillful solutions to specific problems rather than at reinventing forms. Such architecture might appear homogeneous and unified at first, but is rich and diversified in details. As Alexander acknowledges, unself-conscious architecture never faces the problem of complexity that modern architects face, but it is still worth studying because it has a very efficient way of solving problems in a narrower context.

In the world of software, we typically start by reinventing the form and, consequently, we have to face a very wide range of design problems. Are there cases of software construction that are more akin to the vernacular or unselfconscious design? When do we create software by taking an existing solution and gradually adapting it?

I believe there is a number of areas in software development where the basic architecture is fixed and no reinvention of form takes place. One example is large enterprise software such as SAP. The basic structure is fixed, but the system is adapted to local context through configuration and extensions. Another example might be spreadsheet systems like Excel. Spreadsheets define a relatively fixed form and allow the user to focus on skillful solutions to specific problems. Thanks to the fixed form, such specific problem solutions often transfer well between different applications.

We can also find the vernacular approach to software construction also in creative applications that were enabled by programming systems which made it easy to copy and adapt existing code. Two such examples include HyperCard (where stacks were frequently copied and modified) and the 1990s web which made it easy to remix ideas from existing web pages. Indeed, some of my own first programming experience involved copying JavaScript code for "roll-over" image effects and "cursor trailer" animations.

Vernacular software development may only be achievable with sufficiently open software that can, in fact, be copied and adapted. But it could lead to software systems that very efficiently solve recurrent problems and allow us to focus on the specific of the problem, rather than on the challenging and wasteful process of reinventing the form.

6 Conclusions

The kind of software that we need to build is increasingly complex and it addresses ill-defined problems. Yet, our thinking about software and programming is rooted in small mathematical models of programming languages and methods for solving well-defined engineering problems. In this essay, I argue that we should instead model our way of thinking about software after architecture, design and urban planning.

If you were hoping for a plan of a new research paradigm, then I'm sorry to disappoint you. This essay identified a number of methods, characteristics and specific ideas that would be a part of such paradigm, but I openly admit that the paradigm in which those ideas all come together remains elusive. As much as I find this new imagined way of thinking about software relevant for the software we are building today, I expect that we may need to wait for a broader change in the socio-technological context first.

To clarify, let's look at two past changes in how software is built.[56] The first change is the emergence of software engineering and the focus on rigorous development methodologies after the 1968 NATO Software Engineering Conference. The change was not triggered by the conference itself, but instead by a broader software crisis. The increasing computer power and availability meant that programmers started solving more complex problems and an increasing number of companies wanted to build software. The kind of software that needed to be built changed first, the change in thinking about software followed.

The second change is the appearance of lightweight software development methods like Scrum and Extreme Programming in the 1990s. Again, the change was triggered by a broader change in the kind of software that was required at the time. With the growing popularity of affordable personal computers, companies started envisioning new ways of using computers and wanted to build those before their competition would. Again, the kind of software that needed to be built changed first, the change in thinking about the development process followed. The history thus suggests that a new way of thinking about software comes hand in hand with the shift in what kind of programs need to be built. But what kind of programs would be best built using the methods inspired by architecture, design and urban planning?

First, my discussion often blurred the distinction between a user and a programmer. This was not an accident. Livable cities are shaped not just by urban planners, but also by their inhabitants. Adaptable buildings are modified by their non-architect occupiers. The future software requiring new thinking will allow gradual progression from user to a programmer. Habitability, i.e., the ability to understand and modify the system, will apply equally to users and programmers.

Second, much of what I have written requires software that is more open than most systems today. The navigability of software should make it understandable to both programmers and users. However, this cannot happen if the user remains outside of the city walls. We need to let the user in and do not hide the structure of the software from them.

Third, many of my parallels suggest that good software takes time and is built by less organized development methods than we are used to. Buildings arising from vernacular architecture are built and refined by generations of their users. A planning change in Greenwich Village happens when a local community organizes itself and stages a protest. Just like inhabitants own their buildings and cities, inhabitants of software need to own and be able to adapt their software.

[55] The example is taken from Alexander (1964)

[56] The 1960s crisis has been documented by Ensmenger (2012); my account of the history of lightweight development methods is based on that of Varhol (2019)

It remains to be seen if the next software crisis results in software that does not strictly separate users from programmers, allows users and programmers to become joint owners of systems and does not artificially leave anyone outside of the city walls. If this happens, and I sincerely hope it will, then software engineers will need to become (actual) software architects, software designers and software urban planners.

Acknowledgements

The author is grateful to the members of the Temporary Comp Collective and the associated reading group that brought my attention to a number of the references used in this essay. A number of ideas discussed here took their final form during discussion at Salon Littéraire 2021 at the virtual ‹Programming› 2021 conference and during a virtual visit to the MIT Software Design Group. The essay is based on an earlier blog post[57] and the follow-up online discussions also contributed to this revised version. Last but not least, the reviewers provided invaluable feedback and numerous additional corrections and references, including the intriguing reference to Christopher Alexander's work in the proceedings of the NATO 1968 conference.

REFERENCES

Alexander, C. (1964). Notes on the Synthesis of Form (Vol. 5). Harvard University Press.

Amin, N., Tate, R. (2016). Java and Scala's type systems are unsound: The existential crisis of null pointers. In Proceedings of OOPSLA, ACM.

Baker, A. T. (2010). Theoretical and Empirical Studies of Software Development's Role as a Design Discipline. University of California, Irvine.

Basiri, A., Behnam, N., De Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J., Rosenthal, C. (2016). Chaos engineering. IEEE Software, 33(3), 35-41.

Beck, K. (1987). Using pattern languages for object-oriented programs. Technical report CR-87-43. Available at: http://c2.com/doc/oopsla87.html.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A.,... Sutherland, J. (2001). Agile Manifesto. Available at: https://agilemanifesto.org/

Brand, S. (1995). How Buildings Learn: What Happens After They're Built. Viking Press

Brooks Jr, F. P. (1995). The mythical man-month: essays on software engineering. Pearson Education.

Bødker, S., Korsgaard, H., & Saad-Sulonen, J. (2016). 'A Farmer, a Place and at least 20 Members' The Development of Artifact Ecologies in Volunteer-based Communities. In Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing (pp. 1142-1156).

Christian, E. F. (2003). Stewart Brand showed "How Buildings Learn", So Why Can't Software? Available at: http://www.manasclerk.com/blog /2003/07/23/brands-how-buildings-learn-and-why-cant-software/

Clark, C., Basman, A. (2017). Tracing a paradigm for externalization: Avatars and the GPII Nexus. In Companion to the first International Conference on the Art, Science and Engineering of Programming (pp. 1-5).

Cross, N. (2007). Designerly ways of knowing. Birkhäuser.

De Moura, L., & Bjørner, N. (2008, March). Z3: An efficient SMT solver. In International conference on Tools and Algorithms for the Construction and Analysis of Systems (pp. 337-340). Springer, Berlin, Heidelberg.

Evans, E., & Evans, E. J. (2004). Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional.

Ensmenger, N. L. (2012). The computer boys take over: Computers, programmers, and the politics of technical expertise. Mit Press.

Floyd, C. (1987). Outline of a paradigm change in software engineering. In Computers and Democracy: A Scandanavian Challenge, Brookfield, VT: Gower Publishing Company, pp. 191–210.

Gabriel, R. P. (1996). Patterns of software. Oxford University Press.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing.

Ghica, D. (2016). What else are we getting wrong? Available at: http://danghica.blogspot.com/2016/09/what-else-are-we-getting-wrong.html

Hermans, F., Aldewereld, M. (2017). Programming is writing is programming. In Companion to the first International Conference on the Art, Science and Engineering of Programming (pp. 1-8).

Hopper, M. (1955). Automatic coding for digital computers. Remington Rand Incorporated. Available online at: http://www.bitsavers.org/pdf/univac/HopperAutoCodingPaper_1955.pdf

Jackson, D. (2015). Towards a theory of conceptual design for software. In 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!) (pp. 282-296).

Jacobs, J. (1961). The Death and Life of Great American Cities. Random House

Kaijanaho, A. J. (2017). Concept analysis in programming language research: done well it is all right. In Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (pp. 246-259).

Kuhn, T. S. (1962), The Structure of Scientific Revolutions. University of Chicago Press.

Lakoff, G., Johnson, M. (2008). Metaphors we live by. University of Chicago Press.

Lynch, K. (1960). The image of the city (Vol. 11). MIT press.

Mandel, L. (1967). The Computer Girls. Cosmopolitan. April 1967 issue.

Mehmood, A. (2010). On the history and potentials of evolutionary metaphors in urban planning. Planning Theory, 9(1), 63-87.

Meyerovich, L. A., Rabkin, A. S. (2012). Socio-PLT: Principles for programming language adoption. In Proceedings of the ACM symposium on New ideas, new paradigms, and reflections on programming and software (pp. 39-54).

Montfort, N., Baudoin, P., Bell, J., Douglass, J., Bogost, I. (2014). 10 PRINT CHR$(205. 5+RND(1));: GOTO 10. MIT Press.

Naur, P., Randell, B. (1969). Software engineering: Report of a conference sponsored by the nato science committee, Garmisch, 7th-11th October 1968.

[57] Available at http://tomasp.net/blog/2020/cities-and-programming/

Naur, P. (1992). The place of strictly defined notation in human insight. In Computing: A Human Activity, ACM Press

Nofre, D., Priestley, M., Alberts, G. (2014). When technology became language: The origins of the linguistic conception of computer programming, 1950–1960. Technology and culture, 40-75.

Northrop, L., Feiler, P., Gabriel, R. P., Goodenough, J., Linger, R., Longstaff, T., ... Wallnau, K. (2006). Ultra-large-scale systems: The software challenge of the future. Carnegie-Mellon Univ, Software Engineering Institute.

Papapetrou, P. (2015). Software Gardening: Yet Another Crappy Analogy or a Reality? Available online at: https://www.methodsandtools.com/archive/softwaregardening.php

Parnas, D. L. (1985). Software aspects of strategic defense systems. Communications of the ACM, 28(12), 1326-1335.

Petre, M., Van Der Hoek, A. (Eds.). (2019). Software Designers in Action: A Human-Centric Look at Design Work. Chapman and Hall/CRC.

Petricek, T., Guerra, G., & Syme, D. (2016). *Types from data: making structured data first-class citizens in F#*. In Proceedings of PLDI 2016, ACM.

Priestley, M. (2011). A science of operations: machines, logic and the invention of programming. Springer Science & Business Media.

Rittel, H. W., Webber, M. M. (1973). Dilemmas in a general theory of planning. Policy sciences, 4(2), 155-169.

Rudofsky, B. (1987). Architecture without architects: a short introduction to non-pedigreed architecture. UNM Press.

Scott, J. C. (1999). Seeing Like a State: How Certain Schemes to Improve the Human Condition Have Failed. Yale University Press.

Simon, H. A. (1969). The sciences of the artificial. MIT Press.

Slayton, R. (2013). Arguments that Count: Physics, Computing, and Missile Defense, 1949-2012. MIT Press.

Steimann, F. (2018). Fatal abstraction. In Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (pp. 125-130).

Varhol, P. (2019). To agility and beyond: The history—and legacy—of agile development. Available at: https://techbeacon.com/app-dev-testing/agility-beyond-history-legacy-agile-development

Vincenti, W. G. (1990). What Engineers Know and How They Know It: Analytical Studies from Aeronautical History. John Hopkins University Press

Weaver, W. (1958). A Quarter Century in the Natural Sciences. In The Rockefeller Foundation Annual Report, 1958. Available online at: https://rockefellerfoundation.org/wp-content/uploads/Annual-Report-1958-1.pdf