Kyle Steinfeld

public, private, protected
encapsulation and the disempowerment of the digital architect

Initial responses to the now widespread adoption of digital technologies within the architectural design studio were marked by polarization and polemics. Debates raged, and uncompromising positions were staked out by enthusiastic first adopters and reserved traditionalists alike. It is with no small amount of relief that I say that we are past all that now. As Antoine Picon observes in Digital Culture in Architecture, the question is no longer whether or not to adopt digital technology, but what "direction architecture is taking under its influence."[1] Picon goes on to document a range of influential trends and important debates in contemporary digital design culture. One of the key dialectics he identifies is the tension between those interested in the potential offered by advanced design software in unlocking the complexities of form and performance, and those in the minority who advocate moving beyond the mere use of software to directly access the "invisible computational basis"[2] of digitally produced forms. Some would claim that this movement of the majority away from an interest in the inner workings of a new technology and towards its judicious application is a natural result of more widespread acceptance—a sign of a culture reaching maturity perhaps. I disagree. Continuing to attend to the inner life of those humming boxes that sit beside our desks serves not only those designers with a particular interest in digital design techniques, but contributes to the good health of architectural practice in general.

Two Transparencies

The tension identified by Picon within digital design culture is not limited to our discipline, but is reflective of a dialectic present in many technical domains. Take, for example, the contested meaning of the word "transparency" in popular usage, which has come to simultaneously possess contradictory meanings. In her study of how scientists, engineers, and architects have integrated computation into their professional cultures at MIT over the past thirty years, Sherry Turkle touches upon how this autoantonym came to be:

> Transparency once meant being able to "open the hood" to see how things worked. Now, with the Macintosh meaning of transparency dominant in the computer culture, it means quite the opposite: being able to use a program without knowing how it works.[3]

To be understood when using this term, one must now specify: white-box transparency or black-box transparency? The former is a transparency of organizing principles, comparable perhaps to phenomenal transparency in architecture,[4] and suggestive of an ability to know completely. The latter suggests an ignorance of convenience.

White-box transparency evokes literal transparency, in that one can see through any mediating interfaces into the inner workings of the thing in question. This is what is meant by "transparency" in the institutional oversight sense of the word, or within DIY technology circles. In contrast, black-box transparency evokes ease of use. A commonly held principle within the human-computer interface community states that desirable user interfaces must anticipate and direct the cognitive actions of software users. This is known as “interface transparency”: “The transparent interface is commonly defined as one that maximizes task completion and minimizes interfering factors for the user, such as unnecessary interface complexity or performance.”[5]

As an illustration, we may return to Turkle’s discussion of computer operating systems. Compare the experience of wrestling with the relatively exposed inner workings of a Linux operating system with that of the feeling of gliding across the surface of Apple’s OSx. These two experiences could not be more different, yet they both may be correctly described as “transparent.” Merely by drawing a distinction we do not gain much. The curated and seamlessly integrated cult of Apple and the nerdy DIY world of Linux both seem to have their place. Personally, I prefer the constrained path that Apple engineers have collectively charted for me, when compared to the experience of feeling so hopelessly on my own—lost in an Ubuntu wilderness. Why should I not choose the safety of the herd? As we shall see, the implications of this choice is another matter entirely when it comes to choosing the tools of architectural production: the transparency of the black-box does not come without its costs.

Encapsulation

The experience of another discipline may prove illuminating to the choices we make regarding our software, and for this we need to look no further than software engineering itself.

One of the central tenets of object-oriented programming (OOP) is the mechanism of “encapsulation,” wherein groups of data and associated procedures are bundled within a common interface, the details of which are hidden from other processes.[6] The commonly understood advantage of this practice is not technical, but social: by packaging data structures into easily consumed capsules, a programmer can ensure that his or her work is used correctly by others. Separating the messy details of how something works (referred to as implementation) from what the rest of the system sees (referred to as interface), the programmer makes a promise of a stable basis upon which dependent methods may build. Access to the inner workings of this black-box is regulated through classifications of public, private, protected, as well as other modifiers that provide mechanisms of inclusion or exclusion.

The convention of encapsulation enables the larger community of programmers to enjoy the advantages of shared authorship. It allows program modularity and permits coders to build upon the work of those that preceded them. For example, interpreting the principles of OOP strictly, it would follow that once a sorting function is written there's no need for another—that is, until someone comes up with one that works better. Higher and higher levels of abstraction may be achieved in this way, as lower-level processes are perfected and later generations of programmers are freed to concentrate on higher-level tasks. For true believers in the OOP model, this virtuous cascade conjures visions of an endlessly compounding technological progression. Rather than reinvent low-level processes for every new program, coders build on the work of previous generations—standing not so much on the shoulders of giants, but on a pile of ten thousand dwarfs.

In the thirty or so years of programmers working under the OOP model, things have not gone as smoothly as the true believers envisioned. While, for many devotees, perfect modularity is always just around the corner, critics respond that this faith in encapsulation was a pipe dream all along, and failed to take into account a number of inconvenient realities.[7]

First, it seems that in practice, most coders prefer to write their own functions rather than accept the work of others. There is more than simple machismo in this impulse, as basic libraries often set the tone and direction of higher-level work, making an intimate understanding of the details of implementation essential. More troublesome for OOP devotees is the fact that as implementation contexts change, so do fundamental assumptions and the appropriateness of underlying structures. This implies a trade-off between adapting existing code for purposes it was not intended (referred to as a kludge), or tearing down the whole system and starting from scratch. We see this trade-off at work in the phenomenon of "feature bloat," wherein new versions of software become overburdened when adapting to previous structures.

More troublesome for OOP devotees is the fact that as implementation contexts change, so do fundamental assumptions and the appropriateness of underlying structures. This implies a trade-off between adapting existing code for purposes it was not intended for (referred to as a kludge), or tearing down the whole system and starting from scratch. We see this trade-off at work in the phenomenon of "feature bloat," wherein new versions of software become overburdened by adapting to previous structures. Complexity compounds this problem, as more advanced systems require more investment in supporting infrastructure, thereby creating more incentive to sustain existing patterns.

When HTML standards were developed in 1989, structuring text and linking one document to another were among the central concerns, and graphic layout of text was not on the radar. Even after the introduction of the graphical browser in 1993, it was a safe bet that bold fonts were near the highest level of graphic control that a user of this technology could expect. Despite having been developed for text-based context, HTML still forms the basis of most of the visual infrastructure on the web—from animations formatted for viewing on a desktop computer to the tight graphic constraints of mobile devices. Most web programmers agree that many aspects of the HTML standard are inappropriate for today's context, and that a much better set of standards could be developed if starting from scratch. The costs of reprogramming every page on the world-wide-web hold sway, of course, and few such proposals gain traction.

Lastly, and least visible from a technical perspective, the foundations laid by previous generations are anything but neutral, and instead reflect the values and assumptions of the context in which they were laid. This idea was first put forward by Melvin Conway in 1968. The axiom that became known as Conway's Law states that "organizations which design systems ...are constrained to produce designs which are copies of the communication structures of these organizations."[8] The work of developing a piece of CAD software might be divided amongst three teams of programmers: a file input/output team, a user interface team, and a geometric modeling team, for example. Conway's Law correctly predicts that such an organization would tend to produce a piece of software consisting of three major subsystems (I/O, GUI, and a geometry kernel), and that the interfaces between these modules and failure-proneness of this software would correlate with the communication structures linking the three teams of programmers that produced it.[9]

Even from the fairly pragmatic viewpoint of software engineering, we may see that black-box approaches are bundled up in the messiness of contested authorship and the uncertainty of dynamic contexts. Given the opportunity, most experienced coders would choose to start from scratch using only a lightweight and narrowly-focused framework. Are the lessons of this technical culture transferrable to the choices we make as designers? Certainly not on their own, but the paradoxical empowerment of the black-box "transparency" is not limited to software engineering, but is inherent in the very concept.

"Statements Too Costly to Modify"

While we have seen the technical problems and limitations of black-box thinking and touched on the issues of authorship and power implied by it, to go further, we must examine the issue of encapsulated knowledge on a broader level. In his essay "Visualization and Cognition: Thinking with Eyes and Hands," Bruno Latour demonstrates the socio-cultural implications of an idea very similar to encapsulation at work in his observations of the mechanisms of scientific production. He asserts that the unique power of modern scientific culture may be attributed to a set of mundane and practical skills in producing, reading, and writing about images, which he terms "inscriptions." To be effective, these inscriptions must be immutable (durable and generalizable to a variety of situations), mobile (allowing one to gather up and encapsulate many facts from many locations), and combinable (compatible with existing inscriptions such that they may build upon one another). This form of "knowing," unique to western scientific practice, is not a disinterested cognitive act, but is instead tied up with what Latour terms the "agonistic situation": in a conflict between two agents, the "winner" is inevitably the one that can bring forward the greatest number of compelling facts. Seen from this point of view, the power of scientific progress may be explained as the cascading of ever more concentrated inscriptions resulting from a progression of agonistic situations which build upon one another—a condition which both progressively empowers the authors of this material to make discoveries, and serially raises higher barriers to challenges made of established claims.

On the one hand, concentrated inscriptions empower those that wield them: "the great man is a little man looking at a good map."[10] On the other hand, this position requires the raising of a high wall, as ever-higher barriers of entry raised by ever-more concentrated inscriptions empower only some, and only at a cost. Naturally, those outside the system must surmount a barrier in order to participate. But more than that, as inscriptions become denser, the culture as a whole becomes more invested in the structure of the existing body of knowledge. Through this mechanism, paradigm-shifting innovations tend to be stifled and conservative positions tend to become entrenched.[11] Latour concisely summarizes this trade-off with his proposed definition of "reality" as "the set of statements too costly to modify."[12]

While Latour's essay discusses the construction of scientific knowledge, it is a short leap from there to a discussion of architectural software. Take building information modeling (BIM), for example. BIM brings the tenets of object-oriented programming to 3D modeling, and provides a platform for many agents to come together in an "agonistic situation" through what Latour would surely regard as a very highly concentrated set of inscriptions. But does it then follow that the great architect is a small architect looking at a good BIM model? Let us hope not. The user of a BIM model is in a similarly empowered position as Latour's map-reader, and accepts similar costs, both for himself and for the culture at large. Consumers of platforms like Autodesk's Revit are beholden to the use-case assumptions of the software engineers that created it. This includes assumptions regarding workflow, architectural part-definition, even sequences of construction that are often integrated into the information model of the software in a way that is difficult for an end-user to alter.[13] Here, the BIM user must trade his domain over these decisions for the position of power offered by the BIM system.

Latour shows us that authority, whether in the cultures of scientists, software engineers, or architects, can be explained by looking at both the makeup of the authoritative inscriptions these cultures employ, as well as the sociocultural uses of the material. A closer look at "advanced" design software such as BIM reveals a trade-off that many designers might not accept.

Public, Private, or Protected?

The tension observed by Picon in digital design culture—between those interested in applying advanced software in order to unlock the complexities of form and performance, and those who seek to move beyond software in order to access a more fundamental computational approach to digitally-produced form—may be concisely summarized by Turkle's competing definitions of transparency: The black-box transparency of Apple's user-friendly interfaces or the white-box transparency of Linux's open hood. This choice has real consequences, as our institutions must decide between investing limited resources in powerful software (often requiring expensive licenses) and investing in training more empowered users.

First, when it comes to applied design technology, black-boxes and white-boxes are mutually exclusive. One cannot have both. As our brief survey of the promises and pitfalls of object-oriented programming would suggest, powerful software can come at the cost of disempowered users, and truly empowered users prefer application frameworks to software platforms. The black-box of more "advanced" architectural software platforms simply have more assumptions built into them when compared to more lightweight application frameworks. This allows the designer to stand atop taller shoulders, but at the cost of fewer decisions within her control.

Next, I would argue that the culture of our discipline would be healthier if we chose empowered users over powerful software. Latour reminds us that social-cultural values are carried within systems of knowledge production. As one such system, architectural design software carries the values and the authority of the culture in which it arose. While seemingly an innocent act, the choice to accept embedded high-level functionality in software is a choice to defer authority and authorship to software engineers and user-interface designers. While the disempowered position of architects may remain benign for now, over time it will negatively impact digital design culture. Design roles will tend to crystallize, as architects, engineers, and builders collaborate through pre-defined portals. Conventional design methods will tend to be reinforced, as descriptions integrated into information models are more likely to reflect well-known and well-documented procedures. Barriers to entry will grow more formidable as software grows in complexity. Only a strong understanding of software fundamentals will ensure that this complexity actually leads to architectural advances.

Finally, it is my assertion that in order to be effective, computational literacy must extend beyond the mere understanding and judicious use of software, and into a comprehension of more foundational material. While I have not discussed the current state of architectural pedagogy here, take my word that this will require a thoughtful and thorough revamping of the way we integrate critical computation in architectural design education.

I do not advocate that architecture abandon high-level software, but rather suggest the following: if upon careful consideration you find that a given piece of software matches well with your needs and those of a given situation, if you find that the world presented by its interfaces and representations is one in which your design thinking might take root and flourish, then by all means put your money down. At the same time, I remind you that the very facility to intelligently ask these questions requires foundational knowledge and critical thinking. As does the ability to discern the difference between effective and ineffective software, and the capability to maneuver beyond the latter when required. Nurturing this critical faculty is the charge of design education, and foundational knowledge is the key to the empowerment of the digital architect.

[Endnotes]

1. Antoine Picon, Digital Culture in Architecture: An Introduction for the Design Profession, Boston, MA: Birkhaeuser, 2010, 8.
2. Ibid., 10.
3. Sherry Turkle, Simulation and Its Discontents, Cambridge, MA: The MIT Press, 2009, 44.
4. Colin Rowe and Robert Slutzky, "Transparency: Literal and Phenomenal," Perspecta. Vol. 8, January 1, 1963: 45–54
5. Rick Oppedisano, "Common Principles: A Usable Interface Design Primer," The Usability Professionals Association Voice, September 2002, accessed April 30, 2013, http://www.usabilityprofessionals.org/upa_publications/upa_voice/volumes/4/issue_3/common_principles.htm.
6. Erich Gamma et al., Design Patterns: Elements of Reusable Object-Oriented Software, Reading, MA: Addison-Wesley Professional, 1994.
7. See Joe Armstrong Interview in Coders at Work: Reflections on the Craft of Programming, Peter Seibel, New York, APress, 2009: "The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle."
8. Melvin Conway, "How Do Committees Invent?," Datamation, April 1968, accessed April 30, 2013, http://www.melconway.com/research/committees.html.
9. Nagappan Nachiappan, Brendan Murphy, and Victor Basili, "The Influence of Organizational Structure on Software Quality: An Empirical Case Study," in Proceedings of the 30th International Conference on Software Engineering, 521–530. ICSE '08, New York, NY: ACM, 2008.
10. Michael Lynch, ed., Representation in Scientific Practice, 1st ed., Cambridge, MA: The MIT Press, 1990, 26.
11. See Thomas Kuhn's, The Structure of Scientific Revolutions, Chicago, IL, The University of Chicago Press, 1962.
12. Bruno Latour and Steve Woolgar, Laboratory Life: The Social Construction of Scientific Facts, Beverly Hills, CA: Sage Publications, 1979, 243.
13. Early versions of Revit enforced rules regarding architectural geometry, such as walls being perpendicular to the ground, and design workflow, such as analyses that may only be performed on a geometrically complete design iteration.