Generative Design in the Context of Process

The design world is increasingly integrating generative approaches to their workflows. As Lev Manovich notes, the program Processing is gaining an increasing degree of marketshare among designers—in fact, Manovich singles out Processing as "coming close to [Alan] Kay's vision" of an easy-to-use programming environment that allows users to "develop complex media programs and also to quickly test out ideas" (Manovich 2008: 79–80).

The Programmer and the Designer

Alan Kay's Vision for Personal Computing

In his text *Software Takes Command*, it is Manovich's inclination to focus on the work of Alan Kay at Xerox PARC when discussing the development of *cultural software*. Cultural software—and the *cultural computing* which it facilitates—is defined by Manovich as software that is "directly used by hundreds of millions of people" and that "carries 'atoms' of culture (media and information, as well as human interactions around these media and information)" (2008: 3). Alan Kay is a pioneer figure in the computing world, an individual who not only theoritically formulated a vision of the computer as a 'metamedium' but also did a great deal of practical work in order to achieve this vision. Unfortunately, as is all too common in the lives of visionaries, key elements of Kay's ideal never breached into the mainstream even as other aspects where appropriated and commodified wholesale by Apple and Microsoft.

The first, and probably most important, element of Kay's original platform that failed to transfer from his ideal "Alto" personal computing platform to the commercial GUI-drive operating systems that now define the landscape of cultural software[^foss_exception] is the concept of *easy programmability*. To this end Kay founded the basis of all the Xerox PARC work in personal computing on a programming language called Smalltalk. In his text "The Early History of Smalltalk," he explains that the computer science work of the 1960s began to look like "almost a new thing" as the funding of the United States' Advanced Research Projects Agency (ARPA) led to the development of techniques of "human-computer symbiosis" that include screens and pointing devices (Kay 1993). Kay's focus became investigating what, from this standpoint of "almost a new thing," what the "actual 'new things' might be." Any shift from room-size, mainframe computers to something suitable for personal use presented new requirements of "large scope, reduction in complexity, and end-user literacy" that "would require that data and control structures be

done away with in favor of a more biological scheme of protected universal cells interacting only through messages that could mimic any desired behavior" (Kay 1993)[1].

To this end, Kay and other members of the PARC team developed this language Smalltalk to facilitate a "qualitative shift in belief structures—a new Kuhnian paradigm in the same spirit as the invention of the printing press" (Kay 1993). The technical details of Smalltalk are indeed impressive to those who can read and understand them: Smalltalk is *object-oriented*, *dynamically typed*, and *reflective* (*Smalltalk* 2010). All three of these approaches have found increasing adoption in computer programming languges. For instance, Yukhiro Matzumoto's popular language Ruby offers all three of these characteristics and the founder readily admits that his language is greatly influenced by Smalltalk (!CITE!). For the sake of reference, the idea of dynamic typing remains controversial among programming language designers from the implementation of Smalltalk in the 1970s until this very day.

One of dynamic typing's greatest attribute—and perhaps at least a part of why some choose to object to it—is that it makes the programmer's job easy. Whereas *statically typed* languages such as Java or C require the declaration of the kind ('type') of data the programmer expects to store in a given variable, a dynamically typed language has no such limiting expectations. Instead anything can be stored within any variable, something which allows for a great deal of flexibility in the hands of a programmer.

Object-oriented programming, though, represents an even larger shift in the potentials presented to programmers. Object-orientation metaphorically maps the concept of 'objects' onto the practice of programming, providing a means to encapsulate sections of code in a way that encourages reusability and thus increases programmer productivity. It is also said to be easier for the human mind to learn to think in the metaphor of objects over, for instance, the functional programming paradigm that is heavily influenced by the lambda calculus and which represents relatively steep initial barriers to those for whom discrete mathematics is not a second language. Since objects can contain other objects as well as maintain inherited relationships with yet other objects (using the abstract form called a 'class,' which is a pre-instantiated description of what an object will contain once it is called into existence), the potential for constructing programs out of the pieces of other programs was greatly increased.[^I will return to this topic when we reach discussions of 'process hybridity.']

The last attribute of Smalltalk on this list, its *reflectivity*, simply means that every object can explain its own properties, i.e. which "methods" (blocks of code oriented towards performing a specific task) it provides, what variables it is keeping track of, etc. The result is that "objects can easily be inspected, copied, (de)serialized and so on with generic code that applies to any object in the system" (*Smalltalk* 2010). This capacity

[1] http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html

for writing "generic" code once again has a positive impact on programmer productivity, as demonstrated by Kay and Goldberg's description of a hospital simulation designed to be able to compute "any hospital configuration with any number of patients" in their ground-breaking essay presentation of their work, "Personal Dynamic Media" (Kay and Goldberg 1977[2]: 400).

This digression into the specific mechanisms by which Smalltalk eases the tasks of programming highlights the focus of Kay and his group (the Learning Research Group) at Xerox PARC. The intentions of their research—as Kay says in a passage quoted earlier—was a paradigm shift on the level of the printing press. Given the degree to which the mechanics of contemporary computing received their "start" within the Learning Research Group as well as the pervasive impact of that contemporary ("cultural") computing, it can reasonaly be said that they achieved this goal. Further testament to the success of their system was the fact that "children really can write programs that do serious things" (Kay and Goldberg 1977: 394). Indeed, a "young girl, who had never programmed before, decided that a pointing device *ought* to let her draw on the screen. She then built a sketching tool without ever seeing ours" (399).

There is, however, one major element of their implementation—central to the capacities for children programming—that did not find manage to find its way through the commercialization of their ideas via Apple and Microsoft: that of unlimited access to the building blocks of the system itself. While Kay clearly saw not only the potentials but also the real, invigorating effects of providing "exceptional freedom of access so kids can spend a lot of time proving for details, searching for a personal key to understanding the processes they use daily" (404), unfortunately for the un-folding of cultural computing neither the Apple Macintosh nor Microsoft Windows shipped with the capacity for unfettered tinkering with the platform itself. Even worse, they shipped without the programming tools required to even make a program on one's own. Considering the enthusiasm with which children interacted with Kay's system, it is hard not to question where computer literacy—the ability to read *and* write within the computer—might be should either of those seminally-positioned products have shipped with the kind of development environment to be found in Kay's largely-forgotten prototype.

Contemporary Substrates of Design

This line of inquiry into lost historical possibilities serves only to highlight the serious degree to which history affects the unfolding of the metamedium that we call a computer.

[2] The page numbers refer to the article's position within *The New Media Reader*, published in (!CITE!). The original date of publication has been maintained in order to respect the publication's true chronological position in computer history.

Historical developments have also lead to further opportunities, as well. The manifestation of the metamedium as a space operating nearly exclusively on proprietary software in the 1990s drove many freedom-minded individuals to begin, and contribute to, free and libre software. What at first approach looked like an impossible task (Richard Stallman's pledge to implement a completely free implementation of Unix called Gnu's Not Unix (GNU) OS) has resulted in a snow-ball effect that provides not only the backdrop for much of this thesis' discussion but indeed the GNU project provides the compiler with which Apple builds its invisible, proprietary codebase into the commodity known as Mac OS X.[3]

Mac OS X: Process Hybridity in Action

As the Apple platform has long been the preferred environment for professional designers—since before OS X's first release in 2001—it provides the counter-point for our future discussion regarding design in FLoSS operating systems and desktop environments. It allows us to investigate the concept of *process hybridity* that I introduced within the introduction. Mac OS X is a multi-layered system that contains both open and closed source elements with distinct historical lineages. As an assemblage, it can be represented as containing a high degree of process hybridity as it combines not only various separate projects into unified commodity, it also bridges ideological boundaries by combining proprietary, open, and free code into a single commercial object.

This hybridity reaches into the very core of OS X, into the kernel itself. The kernel of an operating system handles the specific task of delivering the instructions deriving from software to the specific pieces of hardware that pertain to those intructions. A simplistic analogy can be made to a traffic cop, whose role it is to direct traffic in a way that the competing interests at an intersection proceed in an orderly fashion. Kernels come in two flavors: the *monolithic kernel* and the *microkernel*. The distinction between the two resides in the size of the tasks they are expected to perform, as well as a material distinction between where operating system code should be executed (*kernel* memory or *user* memory) (*Kernel (computing)* 2010).

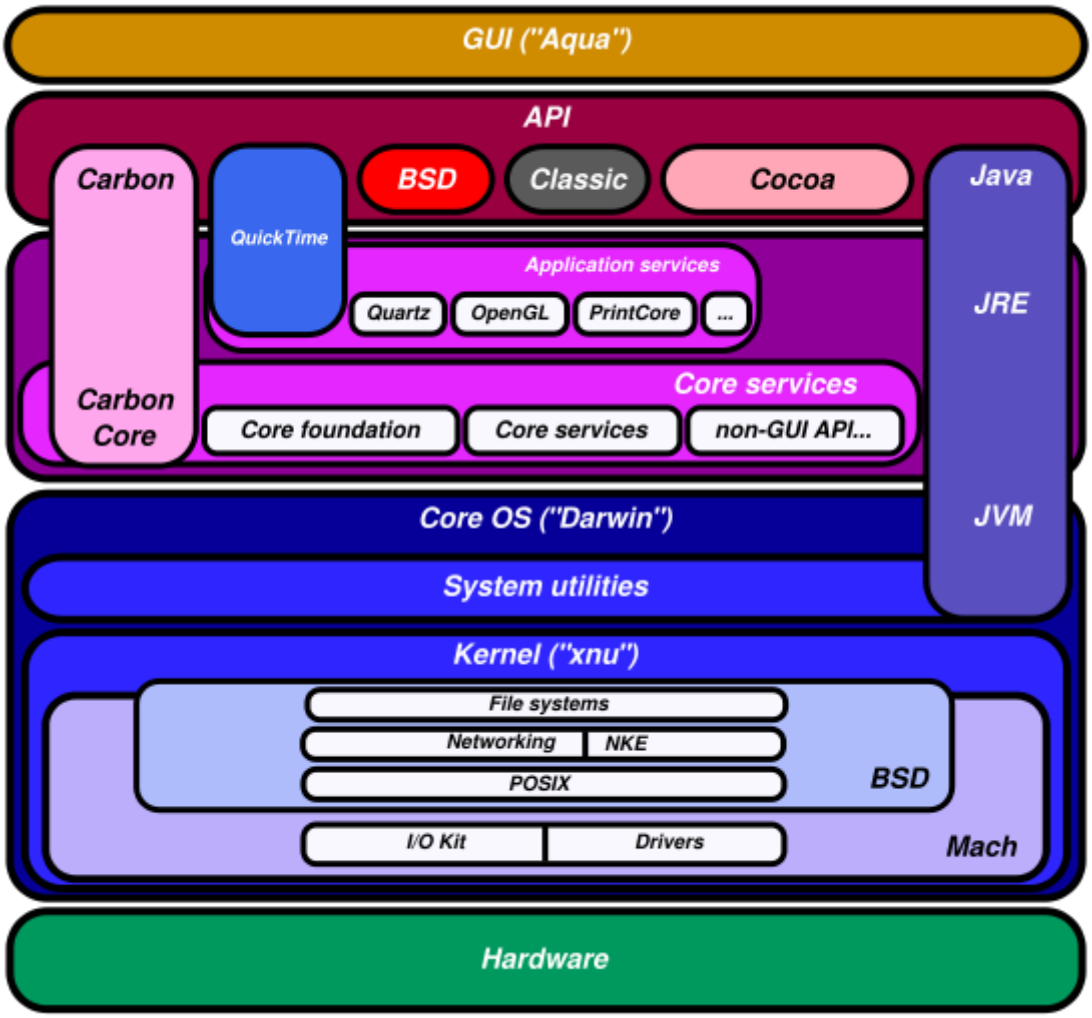
Without entering into a lengthy discussion of the differences between the two—a discussion that has sparked a great degree of vitriol between their respective advocates—let us note that a monolithic kernel contains the code for everything from networking to video display drivers to encryption mechanisms. A microkernel, on the other hand, delegates

[3] In his text "The Ideology of Free Culture and the Grammar of Sabotage," Italian media theorist Matteo Pasquinelli provides a critical analysis combining Marxist discourse on *resource extraction* with Michael Serres' conception of the *parasite* in an analysis of contradictory intersections between capitalism and FLoSS (Pasquinelli 2008).

these processes to *servers* that run outside of the kernel's memory space in the user memory (or *userland*, as it is often called). The result is that there is a great deal less chance of crashing a system that runs on a microkernel, as a bug in user space cannot affect the operation of the hardware whereas a bug in kernel space necessarily can. Microkernel's are also much more modular, as servers can be replaced with (theoretic) ease due to the fact that it sits outside the kernel's code. OS X's kernel, however, is a hybridization of the monolithic and microkernel designs (*XNU* 2010). Due to speed concerns, it was decided that elements of the FreeBSD project's monolithic kernel would be welded onto the Mach microkernel.

Moving up from the kernel we encounter the system utilities. These are directly imported from BSD Unix, specifically FreeBSD. This is the last layer of what constitutes the "core OS", which is released under an open-source license. It must be noted, however, that the vast majority of code within this part of the operating system was licensed under terms that allow Apple to withhold any and all code whatsoever (i.e. *open source* licenses such as BSD or MIT). The major exception to this is the GNU C Compiler (GCC) and its attendant libraries, which are *free* (as in freedom) software and is licensed under terms that force Apple to make available the code of their modifications whenever they publicly release new binaries of their version of the GCC. On the other hand, specific improvements to BSD-licensed code is available only at Apple's whim. It is impossible to monitor whether the source code they make available represents all the improvements they have made or whether they have been selective with their source in order to maintain their proprietary advantage.

Above the core OS level we find further evidence of OS X's storied history. The roots of Mac OS X reside in the operating system NeXTSTEP, developed at Steve Jobs' NeXT, Inc. after he left Apple in 1985. Like all GUI-based operating systems, NeXTSTEP necessarily incorporates a large number of metaphors developed first at Xerox PARC. In addition, however, it was built from the ground up to be easier to develop applications for than other operating systems at the time. To this end the Objective-C language, object-oriented and heavily influenced by Smalltalk, was adopted. While certain systems (such as the microkernel) might be written in C, applications were developed in Objective-C and all the servers above the kernel spoke exclusively with applications in this language. The programming environment was also a leap forward in ease of use: GUI windows could be designed in a WYSIWYG fashion, where their component widgets (buttons, menus, etc) could be easily tied to blocks of code which would inform the computer what was to be done upon a given widget interaction. NeXTSTEP thus represented somewhat of a resurgance for Alan Kay's original vision of personal computing. However, NeXT's first computers were priced at $9,999 dollars—resulting in slow adoption outside of academic



The architecture of Mac OS X. Note the hybridization of BSD and Mach within the kernel section of the diagram. Image courtesy of Wikimedia: http://en.wikipedia.org/wiki/File:Diagram_of_Mac_OS_X_architecture.svg

or institutional contexts. It was within such a context that Tim Berners-Lee developed the first web browser, WorldWideWeb, on a NeXT computer.

Three other hybridities occur within this top level of Mac OS X: Carbon, Classic, and Java (the programming environment of NeXTSTEP was renamed Cocoa in OS X). Carbon was developed as a new way of writing Macintosh applications that would allow developers an easier transition path from Mac OS Classic to OS X. Applications written for Carbon could run on both the older Mac OS as well as OS X, allowing companies

like Adobe to refashion their code without changing programming languages. Classic was an emulation layer that allowed "non-Carbonized" Mac applications to run on OS X (this hybridity has been removed since Apple's transition to Intel microprocessors). Java is a programming language and environment designed to allow universal execution of programs regardless of underlying architectures: if Java has been ported to an OS, then theoretically any Java program can then be run on that OS[4].

The hybridities present in Mac OS X have material effects, allowing for a vast assortment of applications from various lineages and paradigms to coexist and interoperate within a single operating system context. Short of labelling each of these hybridized elements a "medium," there is currently no proposed language within new media for articulating the hybridity of the assemblage as a whole. In this fashion, *process hybridity* provides a mechanism with which we can describe the facets of OS X. Perhaps UNIX is a medium, perhaps not. If it is instead labelled a 'process,' the problematics of "what defines a medium" need not be engaged. In an environment where 3D cinema is not even presented as a new medium in popular discourse, the term itself seems to have lost its utility. Further in the thesis I will introduced more localized and dissectable instances of process hybridity. At this point it should be convincingly demonstrated that Mac OS X hybridizes not only material processes (code) but also ideological processes (licensing terms). This allows OS X its status as the only successful, commercial UNIX for a desktop as well as its status as an easy-to-use, GUI driven development platform (something no other UNIX can claim as convincingly), while maintaining a clean integration with the lineage of previous versions of Macintosh operating systems.

Design Beyond the Proprietary

Much of the prominent research within the field of software studies thus far has been dedicated to investigating the operation of various proprietary applications (Fuller 2000; Manovich 2008). Culturally speaking this may make sense: study what the users actually use. However, from a moral or political angle it can easily be seen as irresponsible or, at the very least, incomplete. Use of proprietary operating systems and application software still seems to dominate the field of new media. This is in spite of Hans Magnus Enzensberger's classic critique of 'repressive' media—if you do not control the mechanisms of your medium, those mechanisms can be used to control you.

Whether for political or economic reasons, there is a small, yet increasing, number of designers and artists who have begun to use FLoSS either exclusively or at some stage in their workflows. Economic considerations should not be underestimated: Adobe charges

[4] However, there are many different versions of Java, making this ideal of "write once, run everywhere" somewhat problematic.

no less than $599.99 for a new (non-upgraded) version of Photoshop, while bundles of their software sell for $1699 (Production Premium CS5) or $1899 (Design Premium CS5).[5] That makes Photoshop more expensive than an entry level desktop system, and the bundles each more expensive than a higher-performance computer! The long-standing solution to this situation is simply piracy. The newer solution, really only available for the last decade or less, is to use FLoSS software to accomplish these goals instead.

Though at a superficial level one can highlight FLoSS design programs as mere reproduction of proprietary process—the GNU Image Manipulation Program (GIMP) reproduces the process of photo editing that is the domain of Adobe Photoshop—Florian Cramer declares that free softwares are not simply cheap imitations or reproductions of proprietary software (Cramer 2010). Instead, he argues, they represent new and unique avenues for accomplishing a given task.

The degree to which the shape of our tools shapes that which we create with those tools has long been established within media theory, though the focus on the dynamics of given medium specificities often overshadow the processes through which those media are filled with content.

Why do relatively so few professional designers use FLoSS? For designer and free software advocate ginger coons[6], the answer lies in a feedback loop between academia, industry, and the designers themselves:

Graphics professionals learn proprietary software in school because the industry runs on it. Employers demand knowledge of specific software because it is the norm. Connected industries, like printing, run on those de facto standards because of their clients. This feedback loop cements the place of proprietary software in the graphics industry. (coons 2010)

Part of the problem lies in the evolutionary nature of the open source development model. Whereas proprietary programs are expected to be "feature complete" with any given release, FLoSS integrates features as they are added. Any potential proprietary competitor to Adobe Photoshop, for instance, has to ship its first version under the expectations of proprietary software: a) it should offer at least the features that are used every day in Photoshop; and b) it should be stable. Otherwise there is no reason for a user to purchase this new proprietary option over the older, more mature one. Also, if coons' formulation of the feedback loop supporting proprietary software is correct, a new proprietary entrant into the field of image manipulation faces vertical economic barriers to adoption: it isn't an industry standard, so the industry isn't hiring those who only know

[5] All prices sourced from the Adobe online store on 18 June 2010 (**http://store1.adobe.com/cfusion/store/index.cfm?store=OLS-US&view=ols_cl&nr=0**)

[6] ginger coons does not capitalize her names for moral reasons related to linguistic processes of capitalizing some nouns over others.

that software. Thus, coons' formulation not only explains why professionals are slow to adopt FLoSS alternatives, it also serves to demonstrate why there is no competition in the space of professional proprietary image manipulation programs—competition, insofar as it exists, is solely within the consumer space where relatively few features are needed to meet user requirements.

In a personal conversation between myself and professional web designer Andy Fitz, he explained how he came to use FLoSS. In college, the Windows operating system on his computer broke down. His roommate restored the computer, but replaced the operating system with a variant of GNU/Linux. After realizing that this meant no more working with Photoshop, Andy gave GIMP a try. Frustrated by the difficulty of transitioning, he complained to his roommate that GIMP simply was not powerful enough. It didn't have all the features of Photoshop. Nothing was in the same place.

"How long did it take you to learn Photoshop?" his roommate replied.

"Five years," replied Andy.

"Learn GIMP for five years. Then you can complain."

Five years later and counting, Andy Fitz is no longer complaining.

Software Time

Five years is both a long and short time for FLoSS. It is long because a piece of software that hasn't been updated for five years not only seems horribly outdated, the chances are high that it will not even compile against the versions of the software it depends on that exist today. It is also a short time because there are many projects for which five years is just a small slice of the time it has been in development. For instance, GNUStep (a free software implementation of the same NeXTSTEP technologies behind Cocoa in Mac OS X) has been in development since the mid–1990s, which means it predates the development of Cocoa by five or more years itself. The typesetting system TeX began development 32 years ago—while the main program is no longer extended, new variations that fix long-standing issues with the original implementation continue to evolve. Five years can be a long time in terms of new features and code growth, yet it is not necessarily a long time in the lifespan of a FLoSS project.

"With FLOSS, the resistance of the tool is now for us such a daily meal, that it has become a work field, an investigation space, and a playground" (OSP 2010).

Generative Design: FLoSS' vector into the design world