The Philosophies of Software

Lindsay D. Grace

Chapter Draft dated 6/08

Found in:

The Handbook of Research on Computational Arts and Creative Informatics

Published: May 2009

Editors:James Braman, Giovanni Vincenti, Goran Trajkovski

ISBN: 1605663522

ISBN-10: 9781605663524
Length:579 pages

The Philosophies of Software

By

Lindsay Grace

Introduction

There is a simple logical proof that describes software's relationship to philosophy. Software is designed. Design prescribes philosophies. Since software is designed, it must also dictate philosophy. The existence of these philosophies, their sociological effects, and the need to critique these philosophies is the focus of this writing. This writing does not seek to define ontologies of philosophies, nor does it seek be an exhaustive examination of the many philosophies that have been institutionalized into the practices of developing and using software. Instead, this article seeks to highlight the existence of a few important philosophies in an effort to encourage practitioners to critically examine their relationship to software and its effects on their practice. In particular, critical assessment of software philosophies engenders fresh approaches to universal, original and effective design.

There are several existing areas where philosophy exerts an influence on software. Each of these areas is not only affected by inherent philosophies, but each area inspires the growth of their individual philosophies by the design and use of their systems. In some cases, the philosophy intersects to create a fulcrum on which multiple assumptions about the construction of the world express themselves. The following sections attempt to outline a few of the major philosophical undertones of common software applications as they relate to the Design of User Interfaces, Avatars, and the use of object orientation.

Careful examination of software decants the following key philosophical elements:

- The Heavy Use of Analogy
- The Application of Reductivism
- An Emphasis on Transferred Agency

Each of these elements directs users toward specific modes of operation, problem solving and creative efforts. This chapter concerns itself with the identification and evaluation of the philosophies resulting from the use, either successful or unsuccessful**,** of software built with these elements. The final section of this writing highlights how these philosophies instruct software users.

Background

For some, Philosophy is a term that should not be paired with software. Within this subset, philosophy is abstract, whereas computer software design is science. Granted that there are scientific underpinnings to software, it is important to recognize that software is used in increasingly abstract ways. It is used to create art, it is used to communicate, and it is used as an integral part of daily work that involves abstract thinking.

The philosophy of software is a topic of research and rhetoric in many disciplines. Although not always considered a philosophical examination, practitioners of law, education, commerce and nearly every software-effected discipline have discussed a kind of philosophy of software. These concerns include intellectual property rights, electronic learning, and the design of systems. The philosophical and commercial work of the Free Software Foundation, for example, is directed toward the specific effect software production philosophies have on the quality of software produced. Theirs' is largely an examination of how software production is practiced, not an examination of how software effects production. This writing seeks to expose the effects of philosophies so ingrained in the production of software that they are seemingly transparent. In the oft-used paraphrase of Marshal Mcluhan, we shape our tools and then our tools shape us (1994).

It is important to note that this discussion excludes an examination of hardware's role. This is because hardware finds design from the realities of physical sciences, where software finds design from logic. The critical evaluation of this logic decants priorities, ideologies, and value systems. Simply stated, it influences the foundations of existing philosophies**.** Those philosophies are encoded in the language and structure of software, and interpreted by the user.

Investigations into the effect of language on people's ability to understand specific ideas are perhaps more akin to the focus of this chapter. In Noam Chomsky's (2006) essays**,** (complied later in the book Language and Mind), he encouraged critical assessment of the relationship between linguistics and philosophy that opened for examination whole processes of communication and approach. The role of linguistics as a tool through which we produce and communicate meaning is similar to the role of software. Software serves as the tool through which much daily production and communication occurs, making it the lingua-franca of operation in most of the western world. It is the basis for communication and the primary vessel that facilitates communication. Programming languages are also the tools we use to direct the development of a solution.

The wide canon of writings describing human-computer interaction also serve as a solid foundation for understanding the tight relationship between psychology and sociology in the development of software. Ben Shneiderman's Software Psychology: Human Factors in Computer and Information Systems is a good starting point. Work in Human Computer Interaction is an easy avenue through which to discover the philosophies of software. It is at this edge of computer science that its many philosophies are first experienced. Human Computer Interaction is also an approachable subject for anyone who has ever used software, as the user has experienced at least one end of this relationship. As such, this writing extends much of its critique to the decisions made about how people must interact with computers.

Lastly, and perhaps most importantly, readers may find valuable related critique in the writing of Jaron Lanier. In particular his essays, One Half of a Manifesto, and Digital Manifesto, evaluate the relationship of computer technology to the society in which it is developed (Lanier, 2006). Where Lanier's concern is with the overarching social effects of computer development, this text investigates the narrower effect of software design on the process of problem solving.

Analogy

Design of User Interfaces

Software user interfaces are largely constructed through metaphor. The desktop of an operating system, the buttons, files, sliders, and others tools are digital implementations of real-world objects and interactions. As theorists have outlined, many of the metaphors are reinterpretations that fail to be wholly faithful to the representation of their original. The desktop, for example, is a shallow metaphor because it exceeds and misaligns the attributes of a real-world desktop. Among its weaknesses as a metaphor is the fact that actual desktops have three dimensions, edges, physics, and other elements not present in any of the dominant operating systems. Even more recent attempts, such as Bumptop, choose specific attributes to perpetuate the use of such analogy (Agarawala 2006). The result is a filtering of subject. This filter exposes the author's values and understanding.

This selected representation of specific elements through metaphor identifies the first hint at the philosophy in user interface. If there is a reinterpretation of the physical object into digital space, then the designer has selected from a list of properties those items that best meet the designer's understanding of the physical object. The result is a simplification, or a kind of wire frame, that exhibits only the items that are most valuable in identifying an object. This value is defined by the one who implements the interface.

Analogy itself is neutral, but the application of analogy is not**.** In writing, analogy is a rhetorical device employed by the author to make a point. Analogy is a device of argument. In the writing of software, analogy continues to make claims. It highlights what is important, and shadows what is not. Yet, unlike writing, software claims are not the subject of critique. Software is understood because it is a tool, and is not designed to withstand critical assessment as its primary function. Yet, an analogy speaks volumes about both the item critiqued, and the author. Every analogy resounds with the author's value system, simply because the process of analogy requires the author to identify wheat and chaff. That which is discarded is of no value to the author, yet in other contexts, that which is discarded is most valuable.

Consider the window as analogy. The dominant property of the operating system window is its ability to display content within it. The analogy is simple to understand. Windows in buildings show the content of the world; software windows display their subject content. Yet, that is not all windows do. Windows insulate, move, and provide multidimensional information about time of day, weather conditions and more. Windows also do more than open and close. Of the many properties of windows, only one dominates the analogy. Other properties are discarded, to simplify the analogy.

In its early fabrications, the graphical button is simplified to an item with two states, on and off. It is differentiated by size, color, and more recently shape. In these two simplifications, the philosophy of design manifests itself. From the dominant theories of computer science, the button is simplified into binary states. The button is either on or it is off. Yet, a literal representation of a button would allow for range. Do buttons in real world machines merely have an on and an off, or do they also exhibit other properties based on length of time depressed, speed of depression, and number of clicks? Interestingly**,** the extremely pervasive intermediary between user interface and user, the mouse, does exhibit these properties: click and hold and double-click. However, once the button becomes digital such properties are selected against. These selections permeate successive generations of software and in turn shape the way in which buttons are understood. The rarity of timer-based buttons, switching buttons or dial and push buttons, all historically useful physical interfaces, is a hint at the forgotten population of interface elements.

Instead, the designer of interface routinely works within the understanding of interface defined by their predecessors. Web Design and Business systems are particularly prone to such tendencies, as their production times are shortened in the race to bring the product to market. Yet, these are everyday interfaces, like the microwave and the television. The everyday interface may easily be among the first human-computer interactions for an individual. This initial experience will likely define an individual's expectations, and more importantly, their understanding of interface. The everyday interface is the gateway to software philosophy.

Philosophical Contradiction: The Acceptance of Non-Truth

To become aware of the philosophies of software, it is important to become aware of its values. Windows and buttons are shallow metaphors. They require substantial suspension of disbelief or belief when used. That suspension is aided by obscuring the user's conventional understanding of the item. Suddenly one window cannot be seen through another. Somehow, two buttons can exist in the same space. One such classic example is the original Apple Macintosh trash can interface element. The trash was a place to discard old files, but it also serves as the means of ejecting a diskette from the machine. To use the operating system, the user accepted the contradiction that disk retrieval occurs through the discarding process.

In these environments, the permeation of analogy inflicts perceptual and conceptual contradictions that the user must accept in order to use the system. Paradoxically**,** the interface becomes a world of same, same but different rules. Those rules are managed by an inherent value system, which enforces what is important to perceive and what is not. Conventionally, if a user tries to make a button stick down, they will fail. If a user tries to hold multiple buttons they will fail. Interface is thus, prescriptive. It instructs its value system by creating a reward system. If the user accepts its value system, the reward is success. If the user fails to accept its value system, the user is punished with impotence in the software environment. Beyond the interface, this instruction abounds in software from video games to business applications. Efficacy in a software environment cannot be achieved unless the individual subscribes to the applications rules. If those rules contradict each other or the individual's understanding of the world, then the individual must still accept them or surrender his ability to act within the software environment.

The dominant human-computer interaction model demands this prescription, but it is not the only approach available. Some video game environments have, in their constant treasure hunt for play**,** made spectacular inroads into non-prescriptive human-computer interaction. Sandbox games or environments in which the toolset may be constructed and manipulated by the user's definitions offer fascinating anti-prescriptive opportunities. Their potential for educators, and for critical assessment of process, is inspirational. Although the application of this method in non-play environments continues to be limited, sandbox environments such as Gary's mod are quality examples and offer alternatives to the most common modes in use today.

The Origins of Analogies

In order to critique the analogy, it is important to understand its genealogy. In some cases such choices are the result of iterations on the same initial design. In other cases, these choices are derived from an interpretation of other disciplines. The analogies of the early painting programs demonstrate a clear translation of paint technique and color theory. Each time, there are notable exceptions that illustrate a system of values in the application. Software that may be analogized to painting processes have focused on the brush and vehicle, but not the surface to which they are applied. Users may choose brush size, pattern, vehicle, and others yet the fundamental choice of the character of the surface to which the virtual paint is applied remains the same. The selected properties mimicked by the application do not compliment the process of painting, but they are prescriptive.

Such environments also preempt the possibility of alternative models of creative process. The process, its tools and their relationships are predefined. Consider how difficult it is to create a work in the tradition of Jackson Pollack, for example. His work relied heavily on a multidimensional approach to paint application.

Most applications focus on the tool and the narrow application of intended use. These applications do not employ algorithms that calculate brush momentum, gesture or brush material. The applications selectively mimic the process. The resulting omissions exclude specific forms of art process because they are beyond the software designer's initial definition of painting.

In compliment, selected art theory is applied to even the most mundane graphical user interface elements. The fundamental notions of graphic design extend into button design. Half of what is understood as graphic design's basic elements voices itself in the differentiation of buttons. As interface technology improves, it implements more of these elements. Shape, texture and hue, for example, are now standard button attributes where once they were not.

Yet graphic design prescribes only one practical means of understanding its subject – in finite space and time. The philosophy of graphic design is tethered by a long-standing relationship to permanent state production. In its philosophy, an item is created in a specific moment and has a set of attributes that remain true for the objects existence. Size, shape, texture, lines, hue and others are permanently identifiable. This is not true of interactive design, which is fundamentally impermanent. An interactive design has varying properties depending on multiple dimensions that include time, space, and event but are not limited to them. Interaction design may also be understood as a confluence of product of actor, state, use case and more according to the Unified Modeling Language (UML). If UML, a language designed to assist in the design of large software systems, describes interaction more completely than the software used to create interface, there seems to be a schism between understanding and implementation. The likely reason is reductivism.

Reductivism: The Finite State Machine

Reductivism defines an historical art movement of painting and sculpture that emphasized simplification. It is also a dominant practice in the construction of computer solutions.

The practice of analogy-based software implementations is reductive in nature. The relationships, interactions, and processes executed by the computer are reduced to their simplest forms. The fundamentals of computer science call for its authors to reduce their subjects to their interpreted atomic forms. For example, a button becomes an element with only two states, or a customer becomes a number.

The finite state machine, or FSM, is an excellent starting point for analyzing the effect of this reductionist philosophy. For computer scientists the finite state machine serves as a model of behavior. It decomposes its subjects into a finite number of states, then or and transitions between those states and actions. The finite state machine is a popular approach to engineering computer logic, including artificial intelligence and human-computer interaction.

Philosophically, the finite state machine defines computer science. Its first step is to make the seemingly complicated simpler. Like much of the scientific approach, it begins by identifying the fundamental elements of its subjects and then it constructs a simulation of behavior from those elements. This construction occurs through a process of deconstruction, where anything to be modeled is first dissected and labeled. Driving a car for example, becomes a flowchart of red light checks and speed monitors effecting driver and car.

All of the FSM modeling process hinges on the appropriate identification of states, transitions and actions. If an element is left out, or it is not related correctly, the resulting software fails to complete its accomplished goal. The process relies heavily on proper decomposition. If the subject is cut in the wrong way, the software may fail to be an accurate model. Yet, the subject must be reduced in order to fit the limitations of computer science. If it is to be modeled in software, the dictum reads, it must be simplified. Again, as in analogy, this reduction necessitates a set of selections. The initial designer of software must decide which elements remain, and which do not.

Quality software production, as currently defined, borrows its understanding of process, states, and the atomic units of its subject from study of the specific situation. A system designed to model chemical interactions, for example, will be informed by research in chemistry. This works particularly well for scientific disciplines, which share in the reductive approach to understanding problems. However, what happens to the disciplines that are not reductive? If a discipline or philosophical understanding of a discipline is not reductive, then computer science may fail to apply its theories.

Consider the contemporary definition of art, which does not assert itself as an understanding of singular necessary components. By some, art is understood as a deliberate arrangement. Deliberate can't be defined as a state, a transition, or an action. Deliberate arrangement is the quality of an action, but the FSM has never been well suited for qualities. The qualitative, that which rests on a non-finite judgment, is not simply categorical. It cannot be quantified, and as such becomes an unmodeled element. In most cases it is simply truncated as a non-calculable. When qualities are judged in computer science, they must still be converted to quantities. Consider the basic algorithm for drawing a curved line on a standard monitor display. If the curve is to be bitmapped for display, software must decide which square pixels will be lit to establish the curved line. All pixels are arranged in columns and rows, so a curve must be estimated. Simply, a curve must be forced into the categorical and decomposed into rows and columns. The result is something that looks like a curve, until it is scrutinized carefully. It is like a curve, but it is a simplification of a curve. The bitmapped curve serves as a proof in computer software design terms. It is analogous to the way software models subjects that are not scientific; it ignores that which does not fit its designed intention. The result is that the bitmap serves as a strong analogy, and will be used. The vector does not, and will be ignored.

Dominant computer software development techniques, such as object orientation, procedures, and others fail to sufficiently solve many problems that are wholly qualitative. In order to apply procedures**,** for example, the subject must still be decomposed. This is where a very wide gap distinguishes itself. If the subject of a software-ization is not reductive, it will more often be turned into reductive elements in order to fit dominant software philosophies.

For computer scientists, the many dimensions of interaction can be encoded into state machines. A common scientific view of interaction requires three processes; define a timeline on which to design, define a screen dimension, and create an event model. Yet, even when all three of these dimensions are combined into a single piece of software they prescribe a specific understanding of the world. Event models describe actions that are rigidly categorized to support the computers understanding of interaction. Events are trapped singly, as an intersection of space and time. Space is defined in absolute terms, as coordinates in a grid.

This model, included first by the software developer, is then adopted by the user of the software, a designer. The designer, eager to build with the tool they have been given, must accept this model in order to operate within the software's constraints. The designer must define their understanding of interaction to accommodate the software's abilities.

Before long, the other means of interpreting user input or of describing relationships simply eludes many designers. The other interpretations are not beyond comprehension, they simply fall in the shadow of other's successes. Much like the history of the electric car, which is as old as its combustion based peer, other software philosophies languish. They languish in the absence of research, and in the distraction produced by the show-stealing conventional approaches. After all, the logic reads, these approaches have served us well. Still, it is easy to be critical of such interpretations. In the case of interface, isn't space also understood in relative terms, as in proximities, neighbors, and distances? Aren't there degrees to interaction that indicate situation? The use of such software drives the user away from these questions. Instead**,** a predefined level of granularity, deemed appropriate by an initial designer, is accepted as useful truth. Eventually, a literal understanding of interaction is supplanted by a modeled understanding of interaction.

Finally, it is important to understand that the finite state machine sits near the intersection of linguistics and computing. Students who learn programming language design, for example, learn the fundamentals of regular grammars and the Chomsky Hierarchy. The hierarchy was theorized by Noam Chomsky, the linguist and philosopher. Chomsky's work provides the philosophical basis for the way many computer languages interpret instructions. Simply, every programming language that employs Chomsky's Hierarchy is employing Chomsky's philosophies. These are philosophies of communication, human behavior, and human relationship.

Reductivism: Object Orientation

Translating the designer's needs into binary terms that the computer understands is an artifact of digital design. The current philosophy states that for software to work, all things must be reduced for codification. This is due in part to finite state machines, but object oriented development, with its hierarchies and inheritance, also drives software development and use.

Software applications are typically developed under the master philosophies of object orientation and inheritance. This philosophy prescribes that there are distinct entities, which when categorized can be forced into an ontology that adequately describes all expected situations. The assumption apparent in this approach works well for producing specific types of software but is exasperating when evaluated from the creative perspective. The philosophy reads that the world is comprised of a finite number of blocks through which there are an estimable set of permutations. If so, this determination predicts a calculable end to creative potential. There are only so many ways that each object can be constructed in this finite world. Wouldn't that then leave the creative world toward an enormous game of Sudoku, where each artist is merely attempting to complete the missing permutations?

A challenge facing software developers can then be traced to the fact that they are taught to compose solutions from dissected components. Object orientation champions the process of simplifying and labeling. These simplifications are shallower than those of the analogies dominating interface because they have become the truth of the software system. Developers begin building their solutions from the already modeled objects that existed before they began their project. A conversation between two computers for example, may have been modeled as a group of listener and speaker objects. Through years of use, that definition of conversation becomes the only definition of conversation. If many applications have been built against that object model, and if there are no problems that arise from that understanding, then it is understood to be an accurate model. Yet, if everyone is working of off that model, and the understanding of that model is passed all the way through to the user, then how will problems occur? Who is left to reinterpret it? The model becomes truth.

Interestingly, science informs science, as computer science finds and defines its solutions by definition provided through other sciences. The terminology in object orientation, for example, is clearly borrowed from genetics. This is a type of incest, which begets solutions that serve themselves. To understand computer science, one must accept its cousins, namely genetics and math. But both these sciences, by their own admission, are incomplete and arguably self-affirming. Genetics has many questions in heredity to answer and assumptions to debunk. Contemporary mathematicians are in the middle of a fundamental reevaluation of math itself (Barrows). As Barrows (1992) described in his critical history of mathematics "our picture of the most elementary particles of matter as little billiard balls, or atoms as mini solar systems, breaks down if pushed far enough, so our most sophisticated scientific description in terms of particles, fields, or strings may well break down as well if pushed too far" (Barrrows, 1992, pp. 21) If "mathematics is also seen by many as an analogy" (Barrrows, 1992, pp. 22) then isn't the construction of software solutions the organization of analogy on an analogical foundation?

Also forgotten is the idea that science, as has historically occurred, can borrow from art or other non-scientific approaches. The producers of several great works in the mathematical realm include multi-practitioner philosophers ranging from Plato to Bertrand Russell. Like computer scientists who attempt to fit solutions into the analogy of object orientation, these practitioners use analogy to explain their philosophies. Yet, not surprisingly, these analogies do break down when pushed too far. Artists, after all, are encouraged to find this breaking point.

The results of this breakdown, the disparities between the pre-defined modular units of a software application and an individual's desired solution**,** occur routinely. When identified, the common resolution for such problems is to use the existing model to construct unexpected results. If, for example, an artist's 3D software creates only 4 primitives, they are instructed to create a 5th, previously undefined primitive by using the original 4. Yet, the 5th pyramid may not actually be a 5th pyramid; it may be a model of the 5th pyramid, lacking some of the 5th primitive's real properties. In casual software language, this is a workaround or hack. The solution is an un-planned retrofit of the solution provided. Too much need for hacking typically indicates insufficient design, yet for creative enterprises, the hack is often the fundamental work unit. New media artists, for example, are fully immersed in the process of hacking, simply to create their proposed solutions. In more practical terms, 3D animations are performed on stages, with rough simulations and environments, like backdrops on stages, and are often not represented in three dimensions, but in two dimensions. This fact then hints at an insufficient solution. The current solutions fail to meet individual's needs.

Returning to the 5th primitive, it is not important, in the eyes of the computer scientist, because it is not part of the original definition of its subject. The world, as defined in the initial software design, does not contain such objects. This is the case with non-Euclidian spaces, like the Mobius Strip or Klein Bottle, which, outside the original models of geometry created by software architects, are very difficult to construct in computer software. The artist is made impotent in a world of digital imagery that precipitates from a chosen philosophical approach, here Euclidian geometry, in to the representation of their image.

In a broader scope, this codification permeates our approach to solving many problems. Simply, Computer Science philosophies deteriorate our understanding of the world. It champions low fidelity, by encouraging the simplification of data, relationship, and multidimensionality. A good computer scientist, as the mantras dictate, converts complicated problems into a subset of simple ones. The mantra ignores its opposite. No computer scientist seeks simple problems and complicates them. Yet, artists often seek simple problems and complicate them. From this perspective, the computer scientist is trained in the act of decomposition. The artist is trained in composition.

As an example, war has a simple solution, stop fighting. The artistic philosophy seeks to unearth the complication in the solution. The artistic philosophy mandates a complication – why is it so hard to stop war, who is involved in war, what does it mean to stop war. The computer science philosophy, instead, seeks to simplify the problem of war so that it may be codified into algorithms. For the philosophy of computer science war is a collection of attributes, mini problems, hierarchical structures, and structures which, like atomic structures, combine to create a whole. Computer Science suggests that it is the responsibility of the designer to interpret those atomic parts before construction. The instructed exercise is simplification - moments become minutes, individuals become groups. To approach the resolution of a problem in any other way on a computer is futile.

The first governments were built on the identification of appropriate purpose of government. Yet that understanding changes over time, and the models changed with their understanding. For some, kingdoms turned to democracies based on assessment of need. Kingdoms, as defined by their kings, were for the benefit of their subjects. They were understood to provide a necessary top-down approach which enforced perpetual, informed management. As some governments evolved, they found a less hierarchical design met their needs. Through a series of wars, hierarchies were overturned for democracies. The democracy continued to offer perpetual, informed management, but the management moved from single silos to more complete, multi-dimensional perspective.

What these civil histories offer, even described in scientific terms, is a model for the potential evolution of software design. The kings of software design hear dictatorial truths about the process of creating solutions, confronting revolutions. Alternative models, such as the growth of open source software or the democratization of information encouraged through various web-based tools like Google maps, indicate a change in way software is constructed. The change is somewhat democratic. Where there was one architect, there are ten. Where there was one algorithm, there are now three algorithms, and twice as many authors.

In order for artists to continue their history of revolution from within the digital domain they must operate outside the inherent philosophies of the software they use. The artists must operate beyond the defined class with identifiable property and objects. They must find a creative space that does not dictate a master–slave relationship between hardware and software components. To accomplish this requires far more initiative and conviction than one might assume. Even the seemingly democratizing force of web art is inherently ruled by philosophies of super-users, IP checkpoints, and a cascade of style inheritance.

Fundamentally, codification means a reinterpretation from spectrums to silos. The only thing that changes is the resolution of those silos. Silos become wider, or thinner, but they continue to be silos. The process of codification is richly philosophical, requiring judgment, selection and interpretation. Critically, the decisions to codify are defined by science itself, leaving little space for other approaches. Yet, the opportunities for extending the reach and power of software may exist beyond the walls constructed by the dominant approach.

Reductivism: Examining the Reductive Language

The language of computing is binary. Its language does not operate on ranges, but its resolution is able to exceed human perceptual range. In the display of graphics, for example, color calculating algorithms are capped to the 65 million colors that are understood to be the perceptual range of human beings. This decision presumes many assumptions – there is no need to code for anything but the average person. The science of human color perception is complete, **so** development beyond perception is unimportant. Out of historical context, these assumptions seem reasonable. In the context of history, they are ideological. Did we once believe the world was flat? Did we once believe the entire world had been mapped? Did we believe the human body was made of humors? Critically, the act of simplification, the philosophy of deconstructing and codifying, abounds in the software we use.

Transferred Agency

The Use of Avatars

In recent years, avatars have become the dominant device for movement in 3D virtual spaces. They are a logical extension of the mouse pointer. What the mouse pointer, is to the index finger in a 2D user interface, the avatar is to the body in 3D space. Both, however, offer an inherent ideology – the user needs agency through a third party.

Good design has evolved from the pointer-facilitated navigation to the touch-screen. The result is a system that is easy to use and requires little training. It is, to use an often dangerous term in human-computer-interaction, intuitive. Touch screen use in automobile navigation systems or automated teller machines is likely easy to use because it removes an intermediary. People do not expect, or necessarily want, the computer to act for them. They want to act.

In education this is a point of critique. Should the teacher assist the student in coloring within the lines, or allow the student the freedom to color outside them**.** In software, the bias is toward prescription, not exploration nor adaptation.

The use of avatars in games is an interesting illustration of this tension. Games are designed to be immersive. They seek to envelope the user in a manufactured experience through a series of simulations and real world analogies. To seem threatening, fire, for example, must emulate the properties of fire in the real world.

A game avatar is a copy of self, in another environment. It is a live broadcast, with self as subject combined with a fiction. Paradoxically, the avatar is the person and it is not. If the player understands the character is *not self*, then they may sacrifice some components of the immersive experience. If they believe it *as self*, then they must subscribe to an arresting philosophy.

The logic is as follows. The avatar is not self. The avatar has great efficacy in the world. I do not have efficacy in the world. I can use the avatar as a tool to gain efficacy in the world. Efficacy is gained through the use of tools. The avatar reinforces the use of tools for agency, not just augmentation of agency. The tools within software encourage their necessity. Using software reinforces, at least philosophically, the need for software.

The dynamic of avatar based software is one of master to slave. The slave, or avatar, is only useful when they are faithful to the commands of the master, or user. An avatar that fails to heed commands is buggy or useless. The prescribed use of avatars indicates that avatars must take commands and take the user's risks.

Avatars are also identified by an outward appearance. In the multimedia world this is a balance of sight and sound. Yet, this is another shallow interpretation of identification. If science has been wrong in its perceptions, how can perception be decanted to what is reproducible in vision and sound? The dominant theory is that more life-like performance is delivered from increased data resolution. If pixel resolution extends beyond our understanding of human perception, then theory dictates that it will be perceived as a real image. Yet, theory does not offer resolution to the dilemma that perception is a multi-dimensional equation. To sell a better image as **"**more real**"** is to sell a larger steak as more cow. The whole of perception extends beyond the silos of sight and of sound. One philosophical view is to consider sight and sound, not as the computer treats them, but as they may be in the real world – a codependent harmony. The deconstructive tendency of Computer Science encourages the use, treatment, and display of perceptual elements independently.

Although quickly discredited in scientific communities, the notion of self and of image of self has been argued to a point beyond what science perceives. Psychological and sociological basis both find themselves as under-represented social science minorities when analyzing avatar implementation. It might be argued that avatar compliments, however, do provide social-psychological informed equivalents in crowd simulation and other models of social behavior. Yet, critical review of these systems finds an initial iteration that is first informed by physics simulation and then roughly layered with pseudo-socio-psychological logic. This logic is, of course, reduced to mini-module logical expressions that reduce the social sciences to computable patterns. Simply, the aforementioned social science minority finds representation in an avatar world, but it is neither significantly represented nor wholly represented. Where Newtonian physics find pervasive application, the soft sciences are episodically integrated into the 1st order science worlds. As anecdotal proof, I offer the disposition of a course introducing the relationship between psychology and computer programming:

"It was hoped that this course would have encouraged participants to view software engineering as a human activity, as well as a more formal discipline. However, the course was cancelled by the powers-that-be after its first semester with lack of student interest being cited as the reason." (Lenarcic, 2004, pp. 257)

The exclusion of social sciences in the development of software systems, even systems that seek to emulate human behavior, is seemingly absurd. How does the science that researches behavior find itself noticeably absent from the science seeking to simulate its philosophies? How did social sciences come to occupy the radical edges of computer science when the computer was designed to serve human needs?

Most likely, the reason is in the philosophy of software. Social sciences are scant to reduce problems to a few small causes. Good social science, the ideology dictates, recognizes the complexity of relationships between the myriad of factors causing specific situations. Good computer science seeks to reduce those factors.

Consider the Boids algorithm. It is an attempt at coding the movement of animals. It reduces the intelligence of movement to 3 key factors, separation, alignment and cohesion. The result is a believable simulation of flocking movement. Successive iterations expand on or use these factors. Yet the foundation for this behavior is primarily physical. A social science description of the primary factors might begin with factors such as intention, drive, desire, and social affinity. Since neither implementation is actual executed in a physical space, but in a theoretical virtual space, both has as much applicability as the other. Yet, the 3D world is driven by a previously existent definition of its world based on a definition of 3D spaces. Hence, an animator finds themselves demonstrating emotion through physical gestures, and later working in sound. The entire basis is physical, with emotion and the soft sciences deriving their reception from perception. Could it be that there are other modes of received information? Social scientists have conducted experiments with alpha waves and other tools ad infinitum, which computer software has completely left out of its definition of world. The direct result is that they have been omitted from the possible expressive means of the users of their software.

Consider the dilemma of relationship building. Computer science decants human-relationships as a computable system of categories. Matchmaking systems find matches through data in a relational database. Social networking systems do the same, and offer computed scores as feedback. Users acquire thousands of friends with little regard for their qualities. Friends become binary, they either are friends or they are not. The quantity is what matters, the quality is non-calculable. Even when the number exceeds what might be considered the literal definition of relationship, the counter keeps climbing. If social science research calculates the maximum number of human relationships at 150 (Dunbar, 1992), computer science defines it as infinite. A scientific peek under the skin of such systems hints at some astounding suggestions. Are friends to be collected, like points in a game? Are friends to be removed for poor performance? Are friends or partners to be determined by categorical matchmaking? Is the sociology and psychology of friend-making simply an equation of demographic data? Do these systems hinder exploration outside the software designed silos? Who directs a search in these environments, database tables or human need?

What We Learn From Our Software

Try evaluating the user interface as whole. Interfaces encapsulate a variety of anti-explorative philosophies. They teach users how to be author led. The environment of an interface is strewn with expected paths, wrong turns, and caution signs. Interface is defined with a push-pull between human and computer.

The need to simplify is emphasized in the abstracted icons and half-analogies of many digital interfaces. Software limits resolution, determining the adequate detail to which a designer designs. The interface of design software emphasizes its own approach. It requires its world to be understood though the same system under which the designer must design.

By using these interface elements, authors are prescribing to these philosophical decisions. Interestingly, few authors of interactive works think critically about these rules to which they are subject. They simply understand the world in which they create to provide rules under which they must work. If these rules are oppressive they do little to thwart them. Instead, they make every effort to use them, and the artificial divisions that are constructed for them. These systems teach even the most revolutionary how to operate under constraints. That the rules of this interchange between human and computer are unchallengeable is another fundamental given assumed by both author and user.

However, this is not solely the fault of the creator of these systems. The author received instruction, and more than likely that instruction included mandates about when one control is used over another. A common introduction to interface design often includes a list of controls and when to use them.

Graphical user interface instruction focuses on what is, not necessarily what should be. Interface design is routinely taught as an exercise in organizing pre-defined interface elements, not as an exercise in creating new interface elements. Designers, in particular, are driven toward a junkyard mentality, acquiring interface elements as they are offered by computer scientists that design them. Yet, to do so is much like painting with only primary colors. If it weren't for the cloud of innate philosophies in software, designers would see the potential in blending interface elements. In its simplest, there would be use for a drop-down list button or a check-box-button-image-list. Instead, many designers are using an out of the box approach, creatively employing use of the set interface elements provided with their chosen application.

Designers also have tremendously untapped potential in the custom design of interface. This extends beyond the common use of specialized controls, since controls themselves are one of many solutions to the dilemma of soliciting feedback from a user. Using the analogy of the real world, progressive designs have requested gestural input. Gestural input represents a fundamental shift simply because it breaks from of the computer as machine analogy that permeates control oriented software. Universal design has also adopted audio interface as a hybridized solution to the dilemma of users with encumbered hands. These two approaches demonstrate a human-computer interaction that attempts to emulate human-human interaction and it could be argued, deteriorates the analogy of graphical user interface as machine interface. Yet, these solutions do not move far from analogy. Speech communication, could for example, be derived from human-human interaction. Such design is then not limited to the creative limits of the designer, but by their ability to find analogy in relationships other than human-machine interface.

As an example-limited design, Microsoft PowerPoint has often been described as a limiting force in presentation. As Tufte (1993) suggests in the Cognitive Style of PowerPoint, it guides discussions in linear paths and decants content into simple bullet points. It also changes the way users organize information, as its attempt at simplicity guides the formation and organization of information. Even its organization of information indicates a value system, where, for example, style is simplified to color, the software precludes the use of angles, and changes the order in which commands might be executed. Users of PowerPoint then become subjects of the PowerPoint philosophy.

What Software Teaches Users

Sociologists believe that specific social systems effect the way that their members perceive and act. The corollary is that members of specific technical systems, through which they work and socialize, will have a similar experience. The software we use on a daily basis effects the way we understand and act in the rest of the world. Our systems have already defined new language, like emailing to im'ing. These new verbs define asynchronous modes of conversation. They describe conversation initiation without invitation. They describe new ways to converse.

The effect extends well beyond language. Software systems define the way in which we interact. Where once machine mediated communication was full-duplex, allowing the communication of our message at the same time we are listening to our message, many popular electronic modes of communication are not. Asynchronous systems, although in some ways technologically minor, are the dominant mode of electronic communication. Email, message boards, blogs, and critiqued posting are all single duplex message systems. These software systems encourage users to talk, and then listen. An email message is sent, and the user waits for the response. Other responses to other questions may arrive in the interim and messages may simple go unresponded. The social equivalent is talking into a crowd with no expectation of your message being heard. In the case of community posting websites the notion of communication is profoundly alienated. Communication becomes a system where one talks and others wait to find that which interests them enough to bother talking. A user posts a movie, and other users browse the catalog of movies, and may decide to critique the recently posted movie. There may be no response at all, akin to speaking in a classroom and getting no response from teacher or student. There may be a flood of responses, but those responses may drift from the subject of the movie to the appearance of the poster. This is similar to a presenter receiving critique on their posture, instead of the content of their presentation.

The notion of being conversant, listening and talking, are replaced in these environments by a new model of conversation. This model is information provider heavy, and information consumer bereft. The consumer famine, is not for lack of information, it is for lack to consume. Software systems make it easy to ignore and even easier to skip. These are the conveniences of software systems. They are also philosophical loaded.

In the standards of design, systems that do not offer the autonomy to choose what to see and what not to see are failing the user because they fail to provide what the user needs. It is considered draconian to dictate, or to take control of the information received by the user. Yet, are there not compulsory experiences that require the attention of the user. Is a film not an entirely different experience if the viewer skips through the center of it, or watches it while watching another? Are the world's greatest speeches effective as web broadcasts?

One philosophy dominates the software industry, and that philosophy is that freedom of choice is positive. This is perhaps a remnant of a developed society which champions choice, or the opposite, a society which has reveled in the choices provided by its systems championing that which it believes empowers it. Does it improve the movie viewers experience to be able to move through the film or is that choice a remnant of an analogy to an archaic device whose translation disagrees with the philosophies of software design and development. Media player software, for example, uses the same control concepts as a cassette recorder. Yet, isn't digital media unbound from the limitations of its predecessors? Aren't the choices provided by digital media players analogy based, but use deficient? The choices they provided are not necessarily appropriate; they are informed by previous systems. The choices given are no better than they choices we had. In such cases the choices given are not given by some well-designed analysis of need, they are given by precedent. Choices given by precedent may become superfluous choices.

What is the need for a choice when its results are negligible? A conventional tape player offered play, stop, pause, fast forward, and rewind and eject. Any media player that provides all six options offers superfluous choice. Eject lacks proper analogy in the digital domain because files are switched, not removed and replaced. Pause lacks application because digital files are either played or stopped; there is no need to leave the tape head pressed against the tape to preserve the current time slot. Yet design dictates that this choice, in particular must be preserved. The result is that many media players retranslate the stop. Stop becomes stop and reset to the beginning, where pause remains the literal stop playing media. The result is a misappropriation of concept.

Where once there was analogy, there is an artificial preservation of choice and reapplication of concept. Stop is redefined. Stop means stop and reset, pause means stop. The relationship to the philosophy of communication returns. If one pauses a message, it lingers. If one stops a message it must be started again, for there is no communication that continues where it was left off. Here the nature of communication is being dictated through choice. Communication, whether it is an asynchronous messaging system or the message in digital media**,** has a few properties defined by software. These include:

- Messages may be broken into segments; the whole is equivalent to its parts.
- Messages are navigable
- Messages are sequenced by quantifiable units

All of these properties are dominated by contemporary computer science approach. The message can be reduced to its smallest parts and in doing so the message can be simulated. Speak with many artists and this can be an inflammatory idea. Can a film be understood by anyone of its 60 minutes? Can an oratorical discussion be preserved as a list of items which can be skipped through, sorted by topic, or returned by topic relevance? Is this essay reducible to a single paragraph? Is a mash up of sound bites representative of its subject?

The destructive power of computer science reduction demonstrates itself. That which was whole becomes parts. What does such daily behavior teach people about their world? Does it encourage us to long for an opportunity to fast-foreword through our monotonies? Does it encourage us to find the shortcut and get to what matters to us? Does it teach us to perceive the parts instead of the whole?

Consider other philosophical approaches to message. What does a non-navigable message communicate? What does the definition of quantifiable units do to the subject? Can message units also include objective items outside the analogy of time-based linear systems?

What Software Teaches Masters and Novices

It is important to remember that software systems have the ability to confirm our perceptions. When the application of an idea proves successfully, it proves itself. If editing a movie in a linear editing system like Adobe Premier proves effective, we are encouraged to subscribe to its philosophies. It is only when the system fails to be successful that we begin to be critical of it. We then ask the important critical questions. Why didn't it work? What is the software doing, that it should not? Where is the incongruity between my understanding of the situation and the system itself?

These are the questions that more often arise from either the masters of the systems (e.g. expert users and hackers) or from the uninitiated who have not been indoctrinated with the philosophical grounds of the application. The hacker exploits the shortcomings of the software system, its lack of proper granularity, or its inability to handle specific situations.

The beginner experiences the software without confining definitions. Yet, each software system requires the beginner to understand its definitions, whether original or derived, for use. If the beginner fails to understand, they fail to use the software, at least in its intended use.

This creates an interesting situation. Those people most capable of critical assessment of software philosophies are those at its ends. People who have never used it and people who have learned it very well have the best perspective to provide critique. The distribution of those two populations varies widely between software systems.

The result is a varying quality of criticality. The more specialized system receives the least number of highly expert, highly critical assessments. The least specialized and highly used application does receive critical assessment at the expert level, but little assessment at the beginner level. Herein lies the dilemma. Experts offer their critique from within the constraints of the system. They can do comparative analysis, and understand the shortcomings of an application from within the application's design. Beginners are outside the application and its inherent philosophies. They have not been indoctrinated with the rules of use. Their unfamiliarity gives them an important perspective from which to critique. As software designers, much of the critical assessment comes from the expert user. Critique of systems by beginners is instead often used to understand the critical gateway to indoctrinating the new user into the software systems philosophy. The goal of many software design assessments is not to radically alter the approach, it is instead to confirm, refine, and improve. This is another philosophy exuberantly promulgated in the philosophy of software. Systems design is not in need of revolution it is only in need of constant revision.

Conclusion

The world of software design is due for a reinterpretation of its values in the same way that historical societies have revolutionized themselves by deep assessment of their universal assumptions. To even describe such revolutions as *next generation* is a failure in critical evaluation of the pervasiveness of these assumptions. Users of technology have been a part of a wide sea of universal assumptions that have at their heart clear philosophical character. As the population of software users increases, the visibility of design flaws has naturally increased. They make themselves apparent in every days design challenges, the daily hacks created not by poor design, but by insufficient design philosophy. Any user of software has experienced these philosophical disconnects. Yet, because of the philosophical disparity in arts and education in particular, the difference between software philosophies and practice philosophies becomes most clear. Art practice champion's approaches are somewhat ignored by software. Education seeks a more complete approach than what computer science deems practical.

The need to look at the design philosophies inherent in software is a real. The science is maturing from a childhood stage of rule accepting, to an adolescence of rule-bending. Its historical structures are showing their wear as its users rock its pillars. In order for new work to break free from the loops of software design**,** artists can explore opportunity in the undefined regions of software implementation. The map is incomplete. These new approaches are not limited by art creation. They include examining alternative programming paradigms, such as declarative programming and building software apart from prepackaged design suites and application programming interfaces.

References

Agarawala, A., Balakrishnan, R. (2006). Keepin' it real: Pushing the desktop metaphor with physics, piles and the pen. *Proceedings of CHI 2006 - the ACM Conference on Human Factors in Computing Systems.* p. 1283-1292

Barrow, J. (1992). *Pi in the sky: Counting, thinking and being*. Clarendon Press

Chomsky, N (2006). *Language and mind*. New York, NY: Cambridge University Press.

Dunbar, R.I.M. (1992). *Neocortex size as a constraint on group size in primates*, Journal of Human Evolution 22: 469-493

Lanier, J. (2003). One half a manifesto. In J. Brockman (Ed.), *The new humanists: Science at the edge* (pp. 233-262). New York, NY: Barnes and Noble.

Lenarcic, J (2004). *Behavioral Issues in Software Development: The Evolution of a New Course Dealing with the Psychology of Computer Programming*. Journal of Issues in Informing Science and Information Technology. P. 247-252

McLuhan, Marshall (1994). *Understanding Media*. Cambridge, Ma: MIT Press.

Nielsen, J. (1993). *Iterative Design of User Interfaces*. *IEEE Computer* Vol. 26, No. 11 (November 1993), pp. 32-41

Reynolds, C. W. (1987). *Flocks, Herds, and Schools: A Distributed Behavioral Model, in Computer Graphics*, 21(4) (SIGGRAPH '87 Conference Proceedings) pages 25-34

Shneiderman , B (1980). *Software Psychology: Human Factors in Computer and Information Systems*. Boston, Ma: Winthrop Computer Systems Series

Tufte, E (1993). *The Cognitive Style of PowerPoint*. New London, CT. Yale University Press