DEMOCRATIZING SOFTWARE DESIGN THROUGH FUNCTIONAL INDIVIDUALIZATION of CREATIVE SOFTWARE

CAN SELİMOĞLU
Submitted in partial fulfillment of the degree of
Master of Design

HAIG ARMEN
Associate Professor

Emily Carr University of Art + Design
Vancouver, BC, Canada

2017

TABLE OF CONTENTS

ABSTRACT

The evolution of creative software is informed by the traditional computer-science methods of development. This results in creation of generic software with task-oriented perspective. User testing data and feedback that sampled from different individuals is used in a way that ignores people's individual differences.

Creative software like Adobe Photoshop have become the digital toolboxes of digital designers, similar to the physical toolbox of a craftperson. While physical craftspeople have the liberty to choose and create each one of their tools to individualize the physical toolbox, digital toolboxes tend to be collections of immutable software tasks, packaged into user interfaces that allow for minimal amounts of customization.

The creative software user is not involved in the part of the software development conversation due to high levels of entry to learning programming, apart from suggesting software features in crowded web forums. This issue has not been addressed by the existing open source software culture.

This research contemplates a systematic approach for enabling all users to democratically participate in the design process of creative software, individualize and extend the application logic in order to bridge the gap between their intent and the output. A new creative software suite that is open to future appropriation through modularity and social extendability for the purpose of dynamically adapting to individual differences is designed and presented as a proof of concept.

Keywords: HCI, adaptive software learning, direct manipulation, visual programming, appropriation of technology, functional individualization, online collaboration, individual differences, intelligent user interfaces, open source, creative software, functional modularization.

1 MATERIALITY AND PEOPLE

1.1 RIGHT TOOL FOR THE RIGHT JOB

On February 1 2003, space shuttle Columbia burned up while re-entering Earth's atmosphere due to an undetected hole on its wing. Everyone on board died. Tufte (2004) notes that NASA officials knew about the risks for two weeks but they dismissed them due to the optimistic tone in the engineers' reports as well as emails within the organization that followed the same content structure of the reports.

These reports were prepared in Powerpoint. In summary, Tufte argues that Powerpoint format encourages imprecise statements, foreshortening of evidence and thought, an intensely hierarchical single-path structure as the model for organizing every type of content and thinly-argued claims, unfit for the purpose of serious technical work such as a real-time engineering analysis of threats to the survival of the shuttle.

These limitations can be considered an aspect of the materiality of Powerpoint as a documentation medium and as a software tool with its own affordances and constraints. "Affordances describe the workable capacities of a medium. Affordance implies a finite budget of opportunities, and so it is complemented with the idea of constraint which is a source of strength because constraints define the materiality of the digital medium. Affordances and constraints together shape expression practicality" (McCullough, 1996). "Despite the lack of physicality there exists a growing possibility of constructing the experience of a medium in the world of the computer. Furthermore, there exists a growing collection of such rich symbolic contexts: a digital repertoire. Intentional differences in symbolic data structure, forms of interaction, and types of indirect constructions yield distinctions between a growing variety of digital media" (McCullough, 1996).

Surprisingly in line with McCullough's 1996 definition, while proposing the system named Sketchpad, MIT computer scientist Ivan Sutherland (1963) named the tools and functions "constraints". It's important to note that Sketchpad is the origin of the contemporary toolbox analogy that we have in creative software such as Adobe Photoshop and Illustrator. Sketchpad was operated through interacting with the screen with a light pen device. The users could state how they intended the pen to behave through manipulating the physical set of buttons, named constraints, on the side of the screen. The pen could be made to draw like a contemporary digital brush tool, copy and drag existing objects similar to working with symbols in illustrator, erase items or transform them geometrically, similar to how we alter the functionality of our mouse pointers by choosing between the tools in our contemporary toolbox interfaces. This legacy, even today, shapes our expression practicality.

1.2 RIGHT TOOL FOR THE RIGHT PEOPLE

Tufte (2004) argues that the Columbia incident could have been prevented by using the right tool for the job. According to him, using Microsoft Word or a similar non-propriatery clone of it would be accurate and adds that this would make the audience smarter (Tufte, 2004). In a similar context, pioneer computer scientist Alan Kay attributes a commonly known quote with a disputed origin to Marshall McLuhan: "We shape our tools and thereafter our tools shape us." (Kay, 1996).

Powerpoint and Word are general utility tools, designed to work with their respective documentation media. Providing general utility for people requires making general assumptions about them. Assumptions that attribute right tools to right jobs work for general scenarios but this approach reduces individuals into personas. "As we overcome the residual notion that computing is for objective documentation only, we must cultivate expressive sensibilities. Chances are that appropriate artifacts and descriptions will engage us through rich and transparent tools, built on newfound densities of symbolic notation and personally experienced as a medium" (McCullough, 1996).

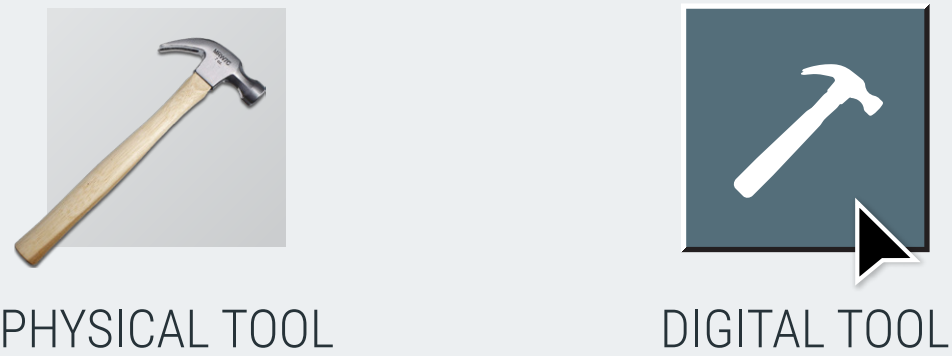


Figure 1: The concept of a digital tool is analogous to real life tools.

In the case of craft, we should talk about the right tools for the right people, rather than right tools for the right job. For a carpenter, choosing a hammer involves being specific about the purpose and how balanced it is in the individual's hand. McCullough (1996) defines the realm of digital craft, amongst other points, as a place "where personal knowledge combines with practical intent, where the expression is as much functional economy as aesthetic stance, where the products are individual and idiomatic and where the medium is the basis for mastery".

1.3 INDIVIDUALITY AND MASTERY

"Tools provide a path, a context, and almost an excuse for developing enlightenment, but no tool ever contained it or can dispense it" (Kay, 1996). "No biological organism can live in its own waste products" says (Kay, 1996) while expressing the frustration caused by the newly realized inadequacies of the tools he designed himself, adds that "inadequate tools and environments still reshape our thinking in spite of their problems, in part, because we want paradigms to guide our goals" and "they resemble the systems themselves, not a new idea" (Kay, 1996).

In the same way that people change their levels of discourse based on who they're conversing with, software has the opportunity to better know its users' preferences, expectations, dislikes and aspirations. A software that does not evolve alongside the needs of the user is problematic because people accumulate experience and their relationship with computer systems changes over time. As individuals become more skilled, computer systems should accompany them through their journey to mastery.

Individual differences of creative people can be seen in the diversity of their work output. Their needs are unique and they inherently differ in the way they do things. It is true that our output and thinking is shaped by our tools but the rate of this is increased in proportion to the specificality of the intended expression. Documentation tools are similar to typewriters whereas a craftperson's tool is similar to a guitar in the potential of shaping the output. An instrument of expression, like a guitar shapes a musician's outcome much more than a generic tool like a typewriter might change someone's writing. "Individuals differ mainly in system related user characteristics, personal characteristics and preferences and previously acquired knowledge and abilities. These broad categories can further be detailed as age, gender, personality, cognitive abilities, cognitive style, learning style, experience, psycho-motor skills, background knowledge, goals and requirements, preferences, interaction styles, motivation and, expectations" (Granic, Nakic, 2010).

The definition of the right tool for the right person also changes over time. "Users change behaviour as their experience with a system develops. It may be expected that there will be a need for different interfaces for the same user and task at different stages. The system is required to arrange itself to match the user’s competence. Users learn different things about the system at different levels at different times" (Benyon, Murray, 1993). While customized tools of expression are not for all creative software users, there is a great deal of evidence that once users reach the level of mastery, they begin to alter and individualize their tools to best suite their intentions.

2 LEAVING THE CORPORATE BOX

2.1 ESTABLISHED CULTURE

While Sutherland curated the constraints and affordances of Sketchpad into functions, knowingly or not, he made way for a specific type of creative output, so do the people who design our contemporary creative software today. Today software producers create design tools that only suit the way certain designers work. Instead of the software growing with your individual creative experience, we are forced to become more proficient with ever-changing software and align with Adobe's vision of software mastery. Contemporary creative software is unable to adapt to or grow with the experience level of the individual. In today's software design culture, user feedback and user testing data that has been funneled from the requests of thousands of users down to is approximated to produce simplified functions of achieving a task for everyone, ignoring their individual differences.

As evidenced in a paper on the "The Process of Redesigning Adobe Acrobat" (Lin, Scull, Walsh, 2002), user input is only taken into account during the production phase between released versions. User tests are conducted to sample user data and the end product is always designed to offer a single way to achieve task. This results in generalized assumptions about individuals. With this approach, the design is always left incomplete for all individual users. Because there are different types of creative minds, forcing people into using generic ways of creating with their creative tools leaves a gap between their intent and the output.

"Computer systems have to be used by a wide variety of people. A nomothetic approach to design excludes people who lie outside the norm" (Benyon, 1993). I personally knew a graphic designer who was using Adobe Photoshop predominantly and was also color-blind. Currently established model of developing software provides no feasible way or intent to adapt to such specific conditions.

IT TOOK ADOBE 47 YEARS TO RE-INVENT THE SHAPER TOOL

1968 GRAIL, Rand Corp.

2015 Adobe Illustrator CC

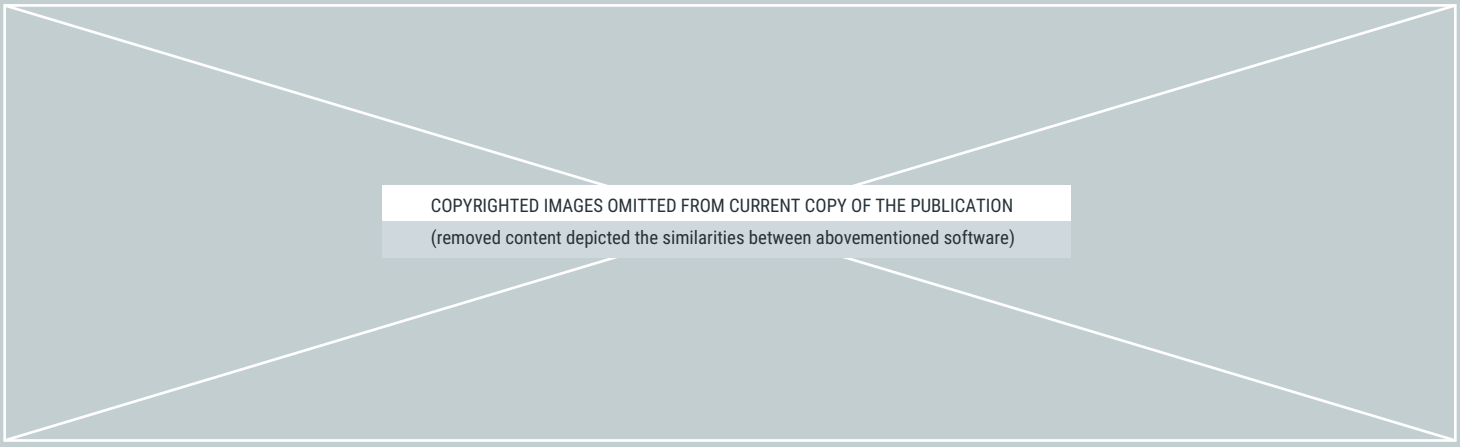


Figure 2: Providers of software may abruptly fade alternative ways of accomplishing tasks into obscurity or recall them back to popularity, regardless of the affinities of the individual users.

Furthermore, corporate culture that surrounds contemporary software design culture, gates access to already invented definitions of affordances and constraints. One such example is a function recently integrated in Adobe Illustrator in 2015 under the name of the "Shaper Tool". It is actually based on design principles of the GRAIL environment (Ellis, Heafner, Sibley, 1969) that were first proposed in 1968 but ignored by the mainstream software providers for decades.

2.2 DESIGN NEGOTIATIONS

Unfortunately, these root issues are not properly acknowledged by the industry and band-aid solutions are provided to their perceived symptom called "bloat". Bloat is explained (McGrenere, Baecker, Booth, 2002) as the way software products become overfeatured over time due to the way new features are added at iteratively.

Two user-interface based solutions, and varying degrees of their combinations are commonly proposed as a means to counter bloat (Bunt, Conati, McGrenere, 2007). An "adaptable" approach is presented as a means to let the user selectively hide less relevant functions from the view in favor of the needed ones whereas an "adaptive" approach is intended to automate this process (Hurst, Hudson, Mankoff, 2007). There have also been attempts to have multiple user interfaces for the same program (McGrenere et al., 2002).

These methods are indeed employed by the industry. Users can choose which functions to display on screen and which ones to hide, alter layouts, close tabs and save their settings as workspaces however alleviating the symptom is not a cure and making unused features invisible to the user counter-intuitively promotes bloat because it encourages an unsustainable culture of iteratively adding underutilized functions that repeatedly miss the mark of catering to the individual users.

Effective design solutions to bloat require acknowledging that a system-based problem can not be solved solely at the user-interface level without reconsidering the way functions themselves are designed. It should first be acknowledged that bloat is a result of the industry's failure to address creative individuals' needs of being able to redefine their own affordances and constraints of their creative tools based on their individual needs and differences.

In order to truly individualize a software product, people must be enabled to redefine their actions that achieve tasks by breaking apart and rewiring established false affordances and actions that don’t work for them. Functional customization goes deeper into the application logic and makes it possible to change the behaviour of an application and is more advanced than customizing only the user interface elements and the layout of an application, leaving underlying functions as they are (Zeidler, Lutteroth, Weber, 2013). With the suggestions provided his thesis, I'm making technology more accessible for non-programmers by lowering people's barriers to building their own tools and functions for creative software, creating a user experience to let them have their own sets of tools according to their individual differences.

2.3 TARGETED USER GROUP

"A programming language was written to enable, for example, children to write storytelling and drawing programs and musicians to write composition programs. In this vision, there was no distinction between a computer user and a programmer. Thirty years after these optimistic ideas, we find ourselves in a different place. A technical and cultural revolution did occur through the introduction of the personal computer and the Internet to a wider audience, but people are overwhelmingly using the software tools created by professional programmers rather than making their own." (Fry, Reas, 2007).

Creative software is being designed like general utility software, aiming to provide the right tool for the right job. While arguing so, this thesis also acknowledges that there are software users who might not feel the immediacy for a change probably because their expectations from creative software are similar to general utility tools or their expectations are accurately catered by the software providers at the moment. However, there are craftspeople who do feel this immediacy, who appropriate, hack and even build their own software depending on their computer literacy.

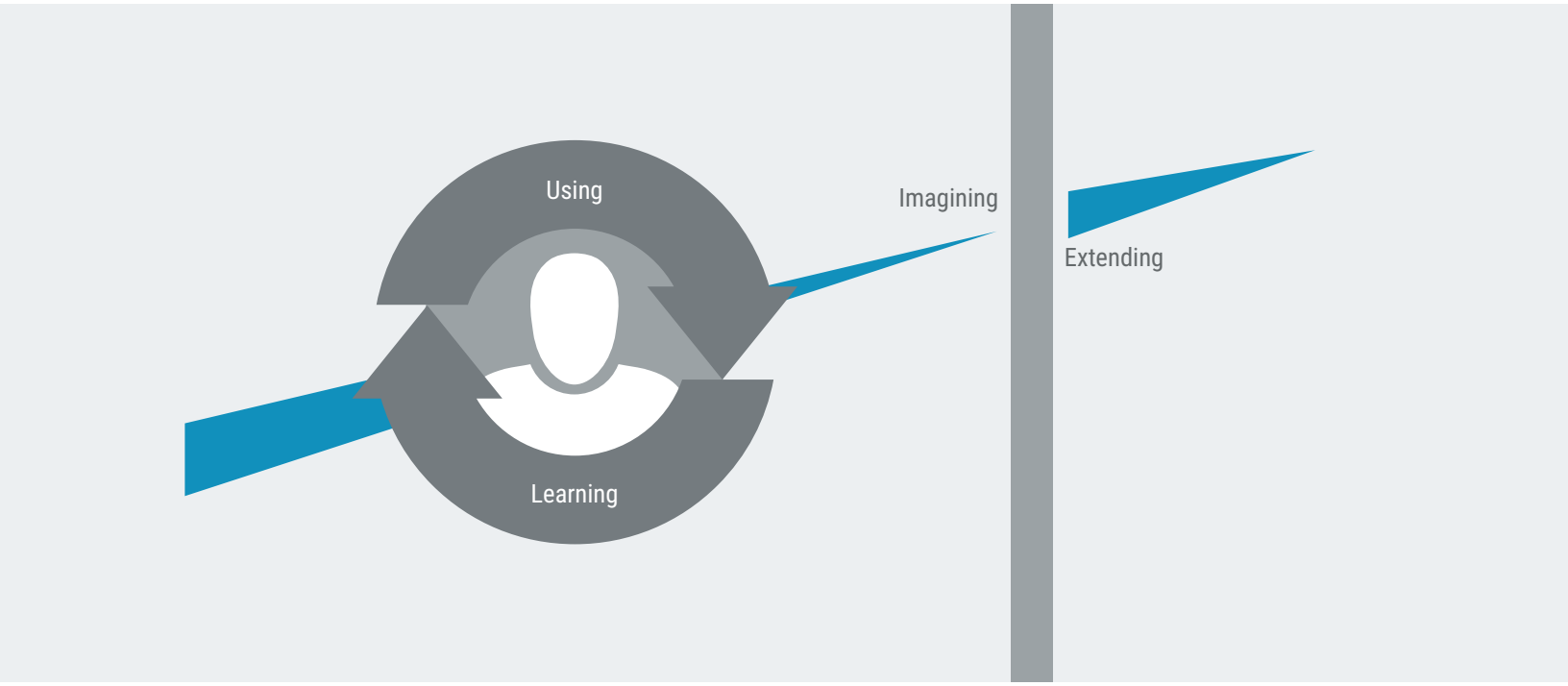


Figure 3: For some, the intent to extend software functionality and the journey towards it, emerges through being a user-learner with expressive sensibilities but due to individual differences, this is not a linear path of progress for everyone. Also the gap encountered between imagining and extending is more personal and social than technical.

Software usage and learning encourage one another. The user-learner group is extensive and is consisted of people with a variety of expectations. Amongst them, there is a sub-group of people who favor individual expression beyond the methods of accomplishing tasks already provided by their software. Some members of this sub group, imagine ways to extend said software functions. Those with the intent to extend, do so with varying degrees of success depending on their proficiency with programming and hacking. Those with little software literacy, such as people from art and design backgrounds, often face a barrier in doing so. This thesis aims to explore the ways to fill this niche. These concepts are further explained in the following sections.

It should also be noted that people transition between these groups if their expectations and industry provided solutions misalign over time. This thesis sees merit in building a culture that promotes methods of developing software that adapts to the individuals at the functional level, blurring the definition between the developer and the user roles by including the users in the dialogues of extending the logic of their applications.

3 OPPORTUNITY SPACE: THE LITERACY GAP

3.1 HACKING

Users first resort to workarounds to circumvent the shortcomings of mentioned inadequacies when their needs aren't met by the provided functionality by appropriating existing elements. "Appropriation of technology is a process by which users complete the work of designers by making interactive systems functional within the frame of their situated activities" (Belin, Prié, 2012).

Appropriated solutions are not internally recognized by the computer system as they only exist at the user's perceptual level. In that sense, they are workarounds that sometimes require giving up on the intended functionality of the repurposed tools. Their outcomes are not system-wide and potentially break when the original software design is altered by the developers.

However, the severity of this shortcoming depends on the vision of the developer. It is suggested (Dix, 2007) that software can be designed for future appropriation in mind and it's a concept that can be embraced by the software provider. In other words, creating hackable software makes extending software through hacking, a viable model for providing software.

A good example for this is the text editor named "Atom" which looks and acts like a regular text editor at first but is completely designed for appropriation. It is advertised as "a text editor that's modern, approachable, yet hackable to the core - a tool you can customize to do anything but also use productively without ever touching a config file" ("A hackable text editor for the 21st Century," n.d.).

At one point in my career as an interaction designer, I found myself in the situation having to use a prototyping tool that only worked on Mac computers when I didn't have access to a Mac computer. Luckily, I found an online guide (Chen, n.d.) to hack the Atom text editor into mimicking the same functionality of that prototyping tool on the Windows computer that I had. That day, for this specific scenario, Atom provided me enough room for appropriation. While there are other software tools designed for varying degrees of user appropriation in mind, such as Ableton Live for music ("LiveCreate, Finish, Perform", n.d.) there is both an unaddressed opportunity to utilize this concept for visual and interactive creative software and a wide gap to further progress into extending software beyond hacking.

3.2 PROGRAMMING

The next step for a computer user towards the path of being able to define their own constraints and affordances is to learn programming and create software or directly edit the source code of existing software. It's important to note that corporate software companies do not sell but license the rights of usage of their software, retaining the intellectual property therefore it is not technically or legally possible to work with the source code. Luckily there are open source alternatives to commercial creative software that is on the market. "Open-source software depends on the availability of its source-code to allow users to debug, customize, and extend it: presumably to free its users to do what they want with it" (Cheung, Chilana, Kane, Pellett, 2009).

Despite that, between appropriation and programming, we have the software literacy gap. "For users of open-source software who are novice programmers, source code can be as impenetrable. Opening up the source code is only the first step for making software more customizable. The challenge then is to ask how HCI principles can work hand-in-hand with open-source to promote customizability" (Cheung et. al, 2009).

Software developers, including open source developers, act as their own researchers and have their own idea about usability. Online open software development and discussion environments such as Github are practically exclusive to people with programming skills. Layman, even designers are not enabled to participate in this type of dialogue. There are many role-sets in open source software design projects such as the module lead, module developer, quality assurance lead, veteran tester, patch contributor, test case contributor, bug submitter, feature requester alongside the module developer and the passive users and observers (Jensen, Scacchi, 2007) but there are no roles that can be fulfilled by non-programmers, including designers. Decision-making processes are exclusive to programmers.

This is in detriment to both parties, programmers and non-programmers, because programmers are barely familiar with designers' process. To envision software and user experience solutions that are truly expandable, we must explore a variety of approaches; programmatic and design-native approaches as well as hybrids of the two. In the case of GIMP, acronym for "GNU image manipulation tool", the open source alternative to Photoshop, software design is an almost exact copy of Adobe Photoshop but there is also an added difficulty curve in user experience for the developers felt the necessity to differentiate the UX from Photoshop but couldn't provide a designerly solution to do so without creating a gap between the user and the already established user experience norms.

3.3 THE PROCESSING TOOL

The only alternate avenue that comes close to close to addressing the gap that I've described for is the creative coding tool named Processing which was specifically designed to target the people who wished to extend the functionality of their software tools. "It integrates a programming language, development environment, and teaching methodology into a unified system. Processing was created to teach fundamentals of computer programming within a visual context, to serve as a software sketchbook, and to be used as a production tool for specific contexts" (Fry, Reas, 2007).

The creators of Processing, Casey Reas and Ben Fry first acknowledge that "as a group, artists and designers traditionally lack the technical skills to support independent software initiatives" (Fry, Reas, 2007).

Fry and Reas (2007) support the idea that "Programming is not just for engineers. Many people think programming is only for people who are good at math and other technical disciplines. One reason programming remains within the domain of this type of personality is that the technically minded people usually create programming languages. It is possible to create different kinds of programming languages and environments that engage people with visual and spatial minds" (Fry, Reas, 2007).

While stating one of his personal goals with Processing, Fry (2007) explains that he wishes to allow designers taking control of their own tools. Afterwards, he goes on to define the gap that is also the focus of this thesis and wishes that designers can be enabled to build their own tools: "As designers have become fed up with available tools, coding and scripting has begun to fill the widening gap between what's in the designer's mind and the capability of the software they've purchased. While most users of Processing will apply it to their own work, I hope that it will also enable others to create new design tools that come not from corporations or computer scientists, but from designers themselves" (Fry, Reas, 2007).

3.4 OPPORTUNITIES

Even with the huge success of the Processing libraries and community, the software itself, although still easier then most programming environments, does not provide a gradual path of entry and still has a steep learning curve for designers and visual learners. "Processing does not present a radical departure from the current culture of programming" (Fry, Reas, 2007) because it ultimately aims to change people into accepting programming as it is suggested today. Interface designer and computer scientist Bret Victor argues that in order to get people to understand programming, programming itself should be changed and turned into something understandable by people (Victor, 2012).

He compares the learning process of Processing to a ruthlessly abbreviated cooking show. "First, you're shown a counter full of ingredients. Then, you see a delicious soufflé. Then, the show's over. Would you understand how that soufflé was made? Would you feel prepared to create one yourself? Of course not. You need to see how the ingredients are combined. You need to see the steps" (Victor, 2012).

Seeing, in this context, is meant literally. "Programming is a way of thinking, not a rote skill, a programming system should support and encourage powerful ways of thinking. Processing environment addresses neither of these goals and ignores decades of learning about learning. People understand things that they can see and touch. In order for a learner to understand what the program is actually doing, the program flow must be made visible and tangible. The environment can make the flow tangible and visible" (Victor, 2012).

4 AUGMENTING THE LEARNING PROCESS

4.1 DIRECT MANIPULATION

Computer scientist and human-computer interaction researcher Ben Schneiderman (1983) prefers the term "direct manipulation" when mentioning tangibility of visibility of the environment and the flow. "Direct manipulation depends on visual representation of the objects and actions of interest, physical actions or pointing instead of complex syntax, and rapid incremental reversible operations whose effect on the object of interest is immediately visible."

Schneiderman mentions ease of learning as a merit of direct manipulation and suggests that programming itself should utilize it. "Novices can learn basic functionality quickly, usually through a demonstration by a more experienced user; Experts can work rapidly to carry out a wide range of tasks, even defining new functions and features; Users gain confidence and mastery because they are the initiators of action, they feel in control, and the system responses are predictable. Performing tasks by direct manipulation is not the only goal. It should be possible to do programming by direct manipulation as well" (Schneiderman, 1983, 1997).

According to McCullough, direct manipulation suspends disbelief, promotes participation and reduces the obscurity when interacting with the system. "The psychological dimensions of human-computer interaction determine the degree of engagement with these symbolic manipulation worlds. Thus, the nature of the computer as a medium began with the introduction of direct manipulation. Establishing both design worlds and psychological engagement depends on building adequate mental models. This is the most essential requirement for the computer to be understood as a medium. The best way to approach these questions is to understand software as a representational context: software designed and used properly creates a world of possibilities within whose assumptions and parameters we operate" (McCullough, 1996)."

4.2 OBJECTS TO THINK WITH

Fry and Reas (2007) list a project by Seymour Papert as one of the origins of ideation for Processing. Papert, with this project in 1960s, aimed to teach children how to program by making use of a robotic drawing device that resembled a turtle. There were also on-screen elements that symbolized virtual turtles. In that sense, it employed direct manipulation with close references to real life (Papert, 1980). The turtle analogy went beyond a drawing device and virtual on screen representations and turned into a bridge between the user and the system. " The turtle became more than a drawing device. It was a creature with certain behaviors which are interesting to study and might help us understand ourselves" (Solomon, Papert, 1976).

We can't speak of a similar engagement model in Processing. It's notable that, in their decision to shape the learners around established concepts programming and not vice versa, Processing developers had to ignore fundamental features of something they were inspired of. "My interest is in the process of invention of "objects-to-think-with" says Seymour Papert and adds "objects in which there is an intersection of cultural presence, embedded knowledge, and the possibility for personal identification" (Papert, 1980). "The Turtle is a computer-controlled cybernetic animal. It exists within the cognitive minicultures of the -LOGO environment-, LOGO being the computer language in which communication with the Turtle takes place. The Turtle serves no other purpose than of being good to program and good to think with. Some Turtles are abstract objects that live on computer screens. Others, like the floor Turtles shown in the frontispiece are physical objects that can be picked up like any mechanical toy" (Papert, 1980).

Victor argues that "the turtle serves a number of brilliant functions, but the most important is that the programmer can identify with it. In Processing, by contrast, the programmer has no identity within the system. There are no strong metaphors that allow the programmer to translate her experiences as a person into programming knowledge. The programmer cannot solve a programming problem by performing it in the real world" (Victor, 2012).

4.3 THINKING WITH SYSTEMS

Of course visibility, tangibility and direct manipulation of digital elements do not have to involve representations of real life objects such as a turtle. "Procedural turtle graphics just wasn't it" says Alan Kay (1972) as he explains Dynabook, another project that is listed to have had an influence on Processing even though its fundamental principles too, are unimplemented in Processing (Fry, Reas, 2007).

Kay's Dynabook further abstracts the analogies of direct manipulation and frames them as a means to allow people learn to think with systems. "Systems became a more thinkable topic in the latter part of the 20th century because a medium for dealing with complex dynamic interactions was also invented in mid-century: the computer. We can just as reasonably think of the computer as a qualitatively new way to understand many kinds of complexity" (Kay, 2012).

The Dynabook concept utilizes interactive on screen abstractions in the form of boxes. While mentioning the user tests he performed on children as young as 12-year-olds, Kay expresses exciting outcomes such as tools built by children that were functional painting, illustration, music and circuit design systems (Kay, 1972).

This was made possible by shaping the learning environment around the learner, in this case the programming environment around the layperson to programming. Kay argues that "The more different and difficult the medium, the less attractive-or even visible-it appears" (Kay, 2012) and further acknowledges the importance of adapting the environment to the learner: "Each of us comes to a particular subject with different predispositions, both genetic and from experience. Some will need very little help, some will need some help, and some will need a lot and different help. Some will be very motivated, some will not be at all interested" (Kay, 2012).

5 STRUCTURAL PRINCIPLES

5.1 MODULARITY

As carpenter choose each of their tools, they design their own toolboxes. This is how people using creative software should be able to design theirs, by choosing their tools in a modular way, at a functional level. Changing the way creative software is designed to an individual centric way requires function-based modularization. While modularity is utilized by developers as well as users, it is not a technical term but a fundamental part of thinking with systems. "When writing a modular program to solve a problem, one first divides the problem into subproblems, then solves the subproblems, and finally combines the solutions. The ways in which one can divide up the original problem depend directly on the ways in which one can glue solutions together. Therefore, to increase one's ability to modularize a problem conceptually, one must provide new kinds of glue in the programming language" (Hughes, 1989).

There are examples of online communities built around modular software platforms where developers share their solutions in a modular way. WordPress is such a content management system with a vast community developing modules for it. ("WordPress.org," n.d.; "WordPress Themes from ThemeForest," n.d.) WordPress is modeled around multiple collaborative projects with their own group of collaborating developers. Each of these modules are associated to a specific function. This way it is possible to build alternative modules for each function, all maintained separately by their contributors. There is also a main project that maintains the core of the software which is designed for external modules to attach to and communicate to one another. This way users choose the functions of their software by choosing their modules. Designing software with the potential for online collaboration towards continuing the design process in a collaborative manner is essentially designing software for future appropriation as (Dix, 2007) suggests.

"Modularity is the human mind's lever against complexity. Breaking down a complex thing into understandable chunks is essential for understanding, perhaps the essence of understanding. Processing's lack of modularity is a major barrier to recomposition. The programmer cannot simply grab a friend's bouncing ball and place it alongside her own bouncing ball variables must be renamed or manually encapsulated; the "draw" and mouse functions must be woven together, and so on. One can easily start from an existing Processing program and modify it, but the language does not encourage combining two programs" (Victor, 2012).

5.2 CROWDSOURCING

In 1291, Venetian Republic ordered all glass makers to move their foundries to Murano island. Clustered in this specialized space, the glass makers concentrated on their profession. The high quality of Venetian glass is known even today. Silicon Valley in California is hosting technology institutions in a similarly clustered way. This pattern shows that the results improve when the density of people with similar goals and interests increase.

Likewise, utilizing the density of creative people already online, provides an opportunity in socializing the development process of creative software. This is related to a very individual-centric area of DIY and maker practice that involves personalization and democratization of technology by utilizing human computer interactions. The concept of hacking started off as a solely software-related term and ended up involving a tangible layer. The physical aspect of the DIY movement made the concept of hacking accessible to non-programmers and cultivated a culture and community around it in which people are part of a whole but as individuals. (Tanenbaum, Tanenbaum, Desjardins, Williams, 2013).

Creativity needs the synergy of many and individual and social creativity can and must complement each other in complex design problems. Sociotechnical environments are necessary for communities to collaborate and bring social creativity alive: to express themselves, combine different perspectives, and generate new understandings. In large and heterogeneous groups working together for long periods of time over complex design problems, as well as in communities including individuals with diverse but converging goals and intentions, distances and diversity between contributing individuals can enhance creativity rather than hinder it (Shneiderman, 2007).

5.3 OPENNESS AND AIMS

Emergence of crowdsourcing amongst industry leading companies serves the purpose of enhancing a privately-owned proprietary software by the help of unpaid outside digital labor. Adobe, for example, introduced Photoshop Design Space as an alternative interface for Adobe Photoshop for the purpose of designing mobile applications. At the time of writing this document, Adobe was crowdsourcing its design and development on Github and the interface itself is built using common web technologies such as html, css and javascript, making community intervention possible. In addition to that, while the end product has collaborative qualities, all effort is aimed towards the perfection of a single official solution both interface and function-wise. Most importantly, the openness of Adobe Design Space is only inclusive to programmers since Adobe doesn't provide a solution to involve non-programmers in the development.

Even though corporate software providers may utilize modular features or community labor, they do so for their own ends. In the case of WordPress, the core is a finished software product, is maintained by a moderating entity that is also the owner and many features are enforced. When the core of the software system acts like this, modules often communicate with the system at a superficial level, failing to perform as well as and as consistent as the functions provided by the software provider. For this reason the core of the software system should act equally towards the functions regardless of who is making them and should be function-neutral by itself.

This thesis acknowledges existing suggestions such as Processing's and aims to avoid their shortcomings while making the concepts such as openness, crowdsourcing, modularization, thinking in systems and direct manipulation do the work of smoothening the progression path for people who want to extend their applications but don't possess adequate computer literacy. This study suggests that negotiating a solution to manipulate software logic beyond the user interface, involves challenging the hierarchical relationship between the provider and the user of the software by making the user an active agent in software development, promoting them to a co-developer state. To this end, this research presents the proof of concept of a framework with the potential to change currently established paradigms about software development models.

6 METHODOLOGY

Four creative software users with some level of proficiency in 2D image creation software such as Adobe Photoshop, Adobe Illustrator or GIMP were recruited as participants during the primary research phase I conducted. They were asked to work with creative tools designed to acquire data from them on the topics of creative software learning, providing accessible levels of entry to extending said software and online socialization for this purpose.

The research methods included interviewing, usability testing through a mix of observation, think-aloud protocol and cognitive walkthrough activities that involved observing the participants while accomplishing known and new tasks and having them verbalize their actions in all phases. Participation involved working through a set of one-on-one activities, including low-tech, paper based exercises as well as screen-based interactive exercises. The questions and figures are included as Appendix A.

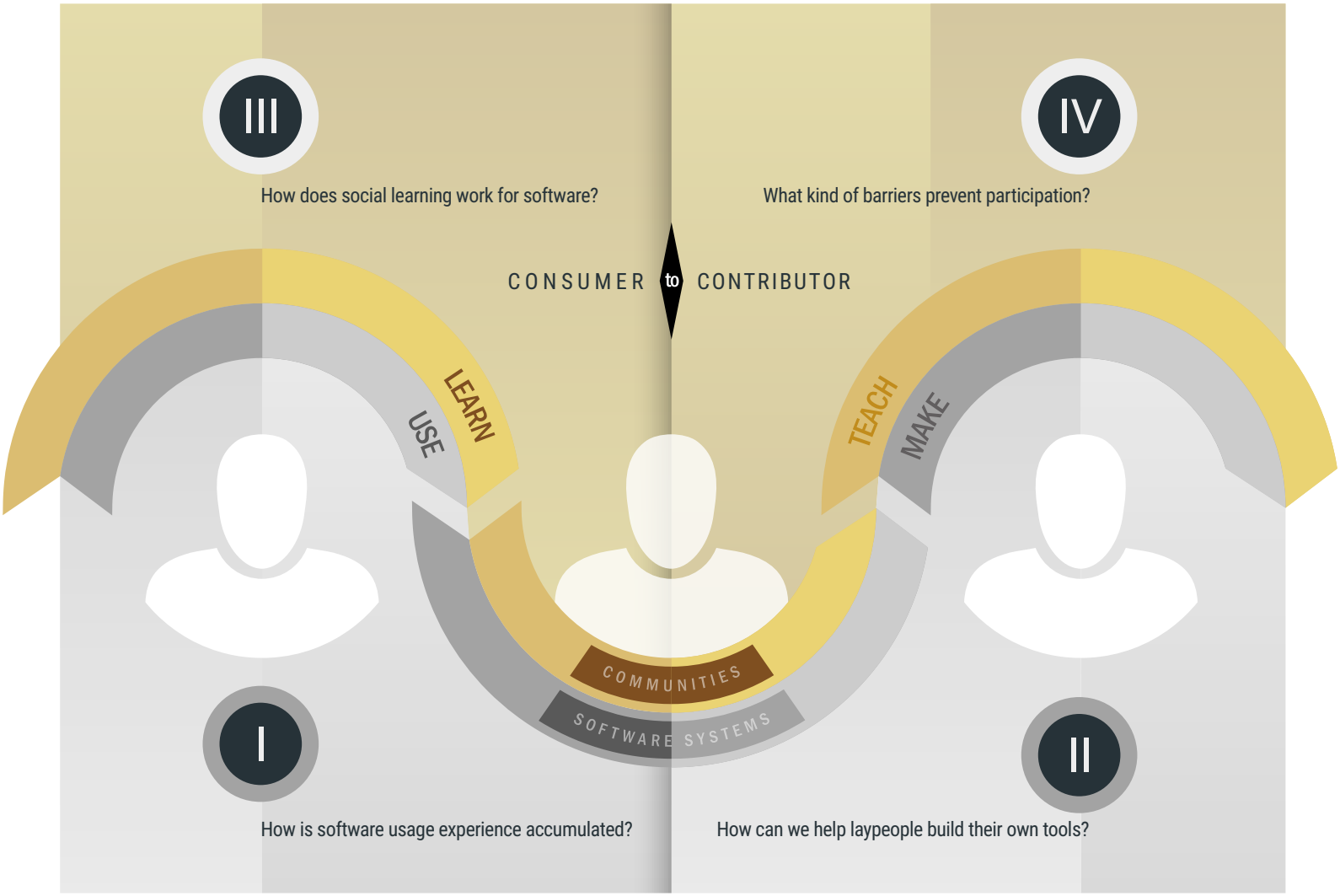


Figure 4: The methodology is established on a two-axis model between social and technical sides of the issue as well as the individual's journey from a consumer to a contributor role.

6.1 INTERVIEWING AND PARTICIPANT PROFILING

Semi-structured interviews were conducted to gather data about the participants' software literacy, software usage habits, how they accumulated experience by themselves and by the help of others. They were asked about whether or not, how and why they participated on specialist online social platforms. This data was used to create anonymized profiles of the participants.

Three of the participants were experienced designers. From now on, they will be referred as A, B and C. One participant was a software developer and will be referred as X. Some questions were based on the creative tool of their choice. All participants chose Photoshop as their main creative tool with the exception of A who was uncertain between Photoshop and Illustrator.

Interview questions were aimed at discovering ways to turn the layman into an active agent who participates in the evolution of their own tools. The questions were categorized under the categories of social learning and participation as well as individual learning and extending. "A" was the only designer with decelerating learning performance and she wasn't motivated about learning anymore, except while working on hobby projects. She would search for information online but would only read top search engine results for key points. She would socialize online for the purpose of learning if there was an easier alternative to reading and writing. "A" had no significant programming skill. As a user with relatively low proficiency, "A" was overwhelmed while learning because she was getting lost due to lack of reference points in software:

> A- *"If I don't know the existence of a method, there's no way I can learn it since I can't learn something I'm oblivious to. If something feels impossible at the moment, I assume it is impossible for me and I immediately try something else that feels easier."*

This comment signified that losing perception of the scale and the constraints of the software environment negatively effected software learning and that in addition to specific tools and functions, our entire systems, software environments have their own constraints and affordances. Therefore their scale and boundaries too, should be made tangible and visible.

"B" was intermediately experienced and had worked with actionscript for 12 years. His learning was getting faster and he was seeking to learn how to do things faster and more efficiently because of the pressure at the work environment.

"C" was the most proficient designer in the context of this research. He was seeking to learn the optimal ways of doing things as well as to play with the limitations of tools. He stated that he was experienced with the entirety of Photoshop's functions with the exclusion of those added after version CS3 because he did not see them as an inherent part of the whole. He criticized proprietary software with these comments:

> *C- "Photoshop versions after CS3 are overfeatured, ineffective and unintuitive. Adobe keeps adding new features because their teams probably have some kind of productivity criteria they have to meet within the company. They aren't designing for the users anymore but themselves. These new functions cause accidents that make me lose my way."*

"X" was an active Github contributor and server-side developer but he did not read online design communities or watch tutorials. His photoshop learning had started off fast but stopped early on. He was using design tools repetitively and inefficiently. New programs, functions and interfaces like the ones in GIMP were unfamiliar and confusing to him. For that reason he couldn't customize design tools. Unofficial plug-ins in creative software were distracting him from the experience because people who were developing them were substituting the designer roles themselves. According to him, because design thinking was not involved in the creation of plug-ins, user experience ended up being too different than the original program and the user's expectations. This can be interpreted as a great opportunity to invite designers participate in software development to the benefit of everyone involved.

When asked about their social learning habits, all designers declared their preference of volunteer communities over official sources in different ways (Appendix A: Answer Set 1). Further answers revealed that designers were interested in extending their tools to varying degrees and those with more experience had more specific intentions, to the point of breaking and glitching the software in an intentional, and an individual-specific manner (Appendix A: Answer Set 2).

These answers showed that the act of individualization did not necessarily have to mean having tool for a perfectly optimized output; it could also mean individualized expressive freedom through freedom from curation.

Interviewing also revealed that those who imagined new functions on their own were facing difficulties due to lack of means to extend software that adapts to their existing skill set. They also felt over overwhelmed and unwelcome in development communities (Appendix A: Answer Set 3).

These answers provided insight on how the acts of using and making software, concepts that once went hand in hand, have now split into separate sub cultures with skeptical stances towards one another. The answer also led to the interpretation that also hinted that a solution to this could be more likely to be formulated by fostering dialogue between software users and volunteer developers compared to between software users and corporations.

Another realization upon profiling the participants was that software usage and software learning were happening simultaneously without one preceding the other and learning would accelerate and decelerate at different phases of their software usage histories for different people. This is considered as an example of individual differences in learning.

6.2 VISUAL PROGRAMMING EXERCISE

After the interviewing, a low-tech, paper based exercise was performed. Several prototyping tools and applications with visual programming components were researched in order to come up with the design of this exercise. Resulting data from this exercise was analyzed to propose a design solution for lowering the participants' mental barriers to building new tools and functions to suggest individually-adaptive software prototyping and building methods.

The participants were presented some cards with graphics on them that depicted different visual programming methods and a visual representation of a screen transition in Adobe XD, a prototyping application. They were explained a scenario, in which they could use either of these methods to change the attributes of this screen transition by the use of an extra button labeled "Advanced Options." This button or such intimate HCI methods don't actually exist in Adobe XD.

The participants were asked to rate each of the approaches according to their potential to freely produce, sustained long term use and legibility. The method labeled "events" was chosen for long term use. Scripting scored lowest in legibility but highest in potential. When asked to summarize their impressions with a few words, the participants answered as shown in the third figure in Appendix A.

When asked to pick one method as their favourite, all designers picked "events" and the developer picked "blocks". Considering all factors, not even the developer wanted to choose scripting as a sole option even with its perceived advantages. When asked about factors that would change their choice, such as the characteristics of the intended end result or their experience level with the program, people were open to refining their choices. They preferred to mix more than one method and in this case, everyone wanted to combine scripting with a visual programming method (Appendix A: Answer Set 4).

These outcomes showed that a programming environment, scripting, that only displayed coding syntax had a distinct effect on people. It was considered both intimidating and intimate at the same time. Visual programming methods on the other hand, were considered more for practical purposes and were seen less powerful in expression but the perception of practicality as well as the perceived potential for expression were higher when scripting could exist alongside visual programming.

According to the data, participants with design oriented backgrounds would agree upon a different visual programming method compared to the participant with a programming background but this is not considered conclusive based on the sample size. In general, the answers did not provide reliable evidence to rank visual programming methods amongst each other but their distinctiveness from the scripting solution was evident.

6.3 MENTAL MODEL EXERCISE

The participants were described an end result that was attainable with an image creation software and were asked to verbalize the steps to accomplish this goal by thinking aloud. The actions involved drawing two, five-sided stars, one red and the other black and also one being approximately 10% larger than the other. Afterwards, they were provided access to a computer and asked to actually accomplish the task. Purpose of this was to assess the difference between the individual's methodological expectation of accomplishing a task compared to its actual representation in the software system.

The more proficient half of the participants, "B" and "C" successfully performed what they verbalized whereas the perceived and idealized methods of "A" and "X" were inconsistent with the reality.

"A" used the "transform" function. Actual experience was inconsistent with the expectation. Expected user experience was a mix between Illustrator and Photoshop. "B" used the "transform" function. Actual experience was consistent with the expectation. "C" used the "contract" function. Actual experience was consistent with the expectation. "X" used the "contract" function. Actual experience was inconsistent with the expectation. Got stuck and did not know alternative ways.

Results showed that the consistency between the individual's methodological expectation of accomplishing a task compared to its actual representation in the software system varied with the proficiency of the user. It is fair to say that ways of accomplishing mentioned tasks were perceived unnatural by the participants with less experience but it's not possible to say whether or not the opposite is true for the experienced participants because their consistency in thinking similar to the software system might as well be a conditioning that also went against their natural way of accomplishing tasks.

If it was possible to conduct a second set of user tests, we could provide symbolic structures that emerged as people interacted with the creative tool as Victor (2012) suggested. This would provide the opportunity to more accurately observe less experienced users as they got lost and could potentially be used to help them advance to a higher level of experience. While conducting this second set of tests was not possible, I saw this as an added opportunity to emphasize direct manipulation techniques in my design proposal. Another important thing to note is that the participants' choice of methods for accomplishing tasks were independent of their success rate or experience, this is considered an aspect of the diversity of individual differences.

6.4 MODULARITY EXERCISE

Using the image creation program of their choice, the participants were asked to, while thinking aloud, select an area of the canvas and reducing the size of the selection by 10% with methods of their preference. Afterwards, they were shown cards that depicted two new function designs that were intended to be modular alternatives to the existing functions as well as to each other, possessing different accessibility to function ratio for users with different experience levels and expectations. One of them was rich in features, whereas the other was simple, focusing on speed and ease of use. Both involved more direct methods of engagement compared to the existing user experience.

After this, they were asked to perform the first task again and compare their new experience to the initial one. The purpose was to observe the cognitive effects of being introduced to modular function alternatives to existing tool. The participants were asked to verbalize their impressions and make comparisons to the methods they used. They were encouraged to think about the occasions they would prefer either of the approaches over another or together, also while taking their own learning into account. All participants were open to the idea of replacing existing functions and to switch between them depending on context. Although "B" did not like programming he didn't feel threatened by the way commands were presented (Appendix A: Answer Set 5).

Both methods were perceived to be useful depending on context but not necessarily depending on individual as no one felt uncomfortable with the one that employed scripting unlike the way they did during the visual programming exercise. It is fair to assess that this way due to the way scripting was presented in a context, alongside a visual representation and in an amount that would assure people that they would not lose control over it. This interpretation helped define the design proposal.

Finally, they were asked to perform the initial task again and compare their new experience to the initial one. The purpose of this was to analyze the cognitive effects of being exposed to function alternatives and modularity in the context of constraints and affordances of the existing tools (Appendix A: Answer Set 6). Participants expressed discontentment when performing the task after talking about the alternatives and some started expressing their own ideas that were different than both the existing methods and shown alternatives. Having shown other ways things could have been, seemed to have sparked their interest. Because they were exposed to the idea of extending software by another person alongside a visualized context, they saw merit in it. This outcome is reflected on the design proposal and helped shape how the entry points to the system were designed for people.

7 THE DESIGN PROPOSAL

THE DESIGN PROPOSAL

Similar to how the people who built Processing realized that they had hit a ceiling with creative tools and proposed their solution to extend, I'm proposing an alternative based on the realizations, analyses and user observations mentioned in this thesis so far.

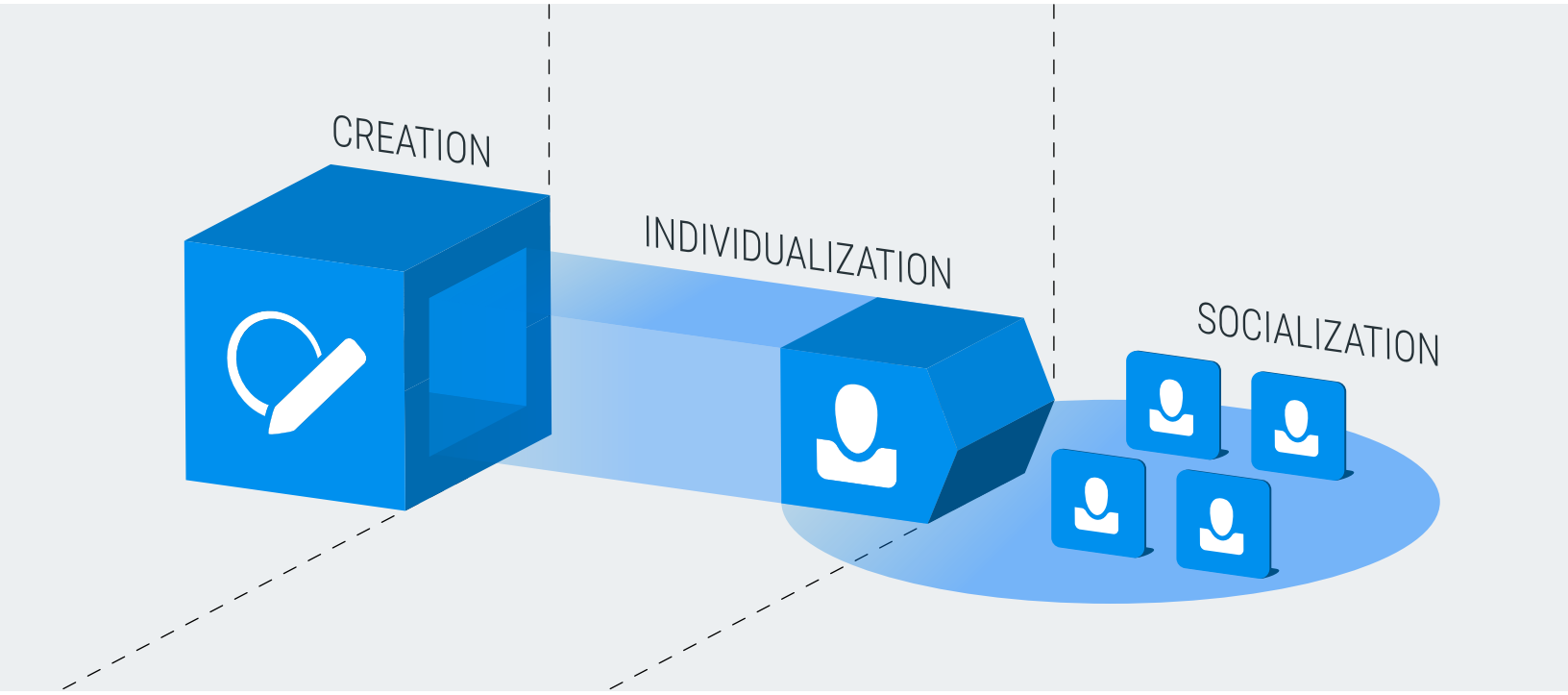


Figure 5: The proposed design aims to smoothen the steep learning curve of extending software for non-programmers and to turn it into a natural part of engagement

The proposed design is a set of three interlinked user modes with distinct purposes. The first one is a creative environment, that appears similar to any other image creation software we are accustomed to but restructured according to the goals of this project.

The second environment is embedded inside the creative environment and made for extending the creative environment through methods that scale to the user's expertise. This can be compared to embedding Processing inside Photoshop with added benefits of direct manipulation.

Finally, the social environment is designed to promote dialogue between people with different skill sets and also acts as an entry point to the system. This can be compared to adding the appeal of social media inside the aforementioned system. The synergy between these environments create a system of perpetual change. Visuals are included as Appendix B.

Once a user without programming skills launches the first mode, she finds it immediately usable because the initial look and feel of this environment is not significantly different than the defaults she is accustomed to. When she decides to make interface-level changes in this environment, she reveals a basic option menu from the relevant panels, similar to the way it is done in extant image creation software. At this point, she notices the addition of further steps she can take by pulling the same button further away. Every step leads up to a gradual increment in complexity and power. As her experience with the system develops, she feels encouraged to experiment with these additional steps because she can do so through direct manipulation. She becomes literate about programming and the ways of manipulating the software logic underneath the user interface through this individualization mode that incrementally adapts to her comfort levels of modifying things. Ultimately, this individualization journey leads up to a final, social environment in which helps accelerate her learning by joining the individually-acquired experience with others' through a non-hierarchical collaboration model.

The developers of Processing also attempted to provide an alternative to the hierarchical relationship between the provider and the user of the software and wanted to promote the user to a co-developer state in software development. They also thought that doing so would blur the difference between the user and the developer roles which would in turn democratize software building but they still expected people to think like computers did. In fact the way programming was taught should have adapted to people's way of learning. Realizations during the primary research suggest that such misalignment of expectations between programmers and non-programmers are perpetuated by the lack of dialogue between these groups.

These realizations, analyses and user observations showed that, in order to allow people have individualized constraints and affordances with their software functions to bridge the gap between their intent and the output, the steep learning curve of extending software should be smoothened and made more gradual for non-programmers, turned into a natural part of engagement through means of direct manipulation. Additionally the social dialogue between programmers and non-programmers should be improved.

7.1 THE CREATION ENVIRONMENT

Analyzing extant image creation software such as Photoshop, Illustrator and GIMP as well as primary research revealed that the user experiences for achieving tasks were not presented in a consistent manner in these tools but were obscurely scattered throughout the interfaces. User data derived from the primary and secondary research suggests that the culture of incrementally adding features while disregarding real life scenarios

resulted in inconsistent software packages as a whole. Primary research showed that less experienced users, like participant "A" needed to perceive the boundaries of the environment better in order not to lose their way whereas more experienced users like participant "C" would deliberately wish to redefine it to the point of glitching.

As opposed to having access to several cross linked functions from menus, button and panels, as it is in Adobe programs, functions are modularized and their user-expected roles are resituated in perceptually consistent boundaries in this proposal, making it possible to categorize user interaction elements that serve towards a task logically. This modularization and categorization is done according to the "levels of tangibility" of the functions, ranked by their desired proximity of access.

While we are working, the canvas is the main focus of our perception, it's our paper. The brush tools act as on-screen extensions of our physical input devices such as the mouse or a stylus so they come in second to the canvas in regards to intimacy. On the other hand software elements that detach us from the immersion of the craft, create varying degrees of disbelief. Inbetween, we have some user interface elements that relate to the more tangible tools, and some that relate to those that break the immersion.

This modularization and categorization allows us, for example to have a typographical sub-system, consisted of the entirety of typography related panels, tools and menu items and represent the materiality of the typographical experience. Same applies to other concepts such as color. This way the, relationships and hierarchies between sub-systems can be defined more clearly, direct manipulation techniques can be applied accurately and the boundaries of the system is perceived in a consistent manner.

This way every functional sub-system -module- may be considered a community project on their own so that the user may choose between alternative modules maintained by different community members according to their own expectations. These expectations may be task based or individual focused. These proposed changes essentially mean adjusting the affordances and reconstraining the materiality of the tool which is a way to make software adapt to the needs of the learning users and also a way for future appropriation and expandability for the benefit of the advanced users.

7.2 THE INDIVIDUALIZATION ENVIRONMENT

As primary and secondary research suggests, upon reaching a certain expertise level with a tool, people desire to extend that tool for specific purposes. For some people, as in participant "C", desire to extend may emerge as a desire to have unforeseen outcomes through glitching. The definition of glitching is very subjective so everyone would answer differently when asked about what a "glitchbrush tool" would mean to them.

Figure 6: A "Glitchbrush" tool is a very subjective concept by nature.

A typical corporate creative software approach might be to offer designers a generic glitch tool, which goes against the notion of establishing your own personal way of glitching imagery. The only way to provide people the means to have such tools is to let them build their own creations and alter them as their expectations change over time.

The individualization environment resides as an extension to the creation environment and is intended to provide such means through a mixed use of direct manipulation techniques and traditional programming methods. The user experience is intended to adapt to the proficiency level of the extender. A smooth transition between different levels of complexity is provided to let people gradually reveal and conceal basic and advanced methods, offering lower levels of engagement with the mechanics at every increment.

Scalability of this environment helps blur the line between the developer and the user roles adaptively. In sequence, a basic tool configuration interface expands to a personal assistant mode which in turn expands to a coding abstraction mode, finally multiple coding abstractions and access to actual code is provided at the same time, side by side, employing a "what you see is what you get" (WYSIWYG) approach. This approach encourages people to occasionally move outside of their comfort levels of modifying things because doing so doesn't require any dedication. People who expand into a difficult stage may return and continue from a mode they feel comfortable with.

This environment and the tools themselves are intended to be built by common web technologies such as html, css and javascript. Web technologies have been successfully used for making desktop applications before; Adobe Design Space being an example of that. As of the date of this thesis, the most popular open technologies for this purpose are Electron ("Electron", n.d.) and Node Webkit ("NW.js", n.d.).

In order to demonstrate this concept, visual abstractions from the primary research stage are re-used. They are intended to be replaceable modules themselves but for the purpose of this demonstration, they are curated through a list of existing approaches from prototyping tools such as Framer, Marvel, Scratch, Origami, Principle, Sketch, Avocado, Flinto, Atomic, UXPin, Prott, Pop, Webflow, Adobe XD, Proto.io, JustInMind, Balsamiq, Mockplus, Invision, Form, Omnigraffle and Axure as well as programs that aren’t intended to create prototypes but include a visual programming component like Construct, Antares Universe and Unity Playmaker.

7.3 THE SOCIAL ENVIRONMENT

The social environment is a web based collaboratory space, accessed from within the software as well as from any web browser, including mobile devices just like a regular website or packaged as a social mobile application. It provides an accessible alternative to existing collaboratory development environments such as Github by making difficult concepts visible and tangible, making the experience similar to a similar to a DIY contributory space through means of direct manipulation. Unlike the existing open source software development project organizational models, it's aimed at making non-programmers a part of the development process with a non-hierarchical collaboration model that includes roles for both designers and developers, allowing people to contribute according to their skills.

Github and existing graphic user alternatives to Github such as GitKraken and Github Visualizer as well as contemporary social media environments were significant inspirations for the imagining of this environment. This environment is intended to be built by the same web same technologies mentioned earlier and the project files are intended to be stored with a cloud storage solution.

Here, each module is treated as an independent open, community project within their own scope, maintained collaboratively by teams consisted of people with different development and design skill sets. As mentioned earlier, multiple projects may exist for the same purposes at the same time, by different teams and people are encouraged to undertake the role of maintaining a copy of an existing project according to their own visions by creating a split instance of the project. This is the process called "forking".

As a part of the dialogue, users package their assets and upload them to this environment just like sharing images on social media. These assets may contain visual mock-ups for collaborating with the programmers, complete code and design solutions and anything in between. Concepts and terminology such as "library, SDK, version control, forking, extendability" translated from the world of software development are visualized and made a part of the user experience, making them more human readable and interactible.

Integrating a social component in the software system also lowers the barrier of entry for extending the system. User experience elements borrowed from social media allow people to see what others are doing and are therefore invited to be a part of the dialogue. This invitation to extend is intended to trigger awareness about the boundaries of the extant functions on people who have never been engaged with these concepts before. People, welcomed through this approach, are going to see merit in the proposals outlined in this thesis even before they reach a certain expertise level with their tools.

8 INSIGHTS AND FUTURE DIRECTIONS

INSIGHTS AND FUTURE DIRECTIONS

It is true that making helps us think but making does not only have to be a means to this end. Building finished systems during design research also has merit. Making high fidelity prototypes and delivering them to actual users allow us acute task observation opportunities in real life scenarios and in turn, help us think through problems with real user data. This research could have utilized such user data if the designed system could have been built as a finished product.

Having built the software would provide the opportunity to conduct iterative user tests for a series of assessments and revisions. Right now it is presented as a proof of concept because building it within the available time frame would require teaming up with actual programmers.

Just as developers leave out designers from their procedures, my process lacks development skills which would allow for a more accurate representation of the final design intention and provide opportunities to test social features as well as further opportunities regarding adaptivity and learning. As the ultimate goal for this project was to allow people build their own tools, the community should have been integrated in the design process as early as possible.

Further user tests could be used to experiment on integrating adaptivity beyond the user interface since there is no reason to consider adaptivity mutually exclusive with appropriation, hacking or building your own tools. The benefit of intelligent interfaces can go as far as detecting the expertise level of the user and it has already been experimented on GIMP (Hurst, Hudson, & Mankoff, 2007). A system that adapts to the user at the function level may improve the design proposal of this thesis.

This project could also be developed with alternative input devices and hardware based HCI solutions in mind however it was a deliberate decision not to propose hardware-specific designs at this point. The cultural shift in software design needs to be triggered in software first. As the tools and processes get more democratized, it is going to be possible to bring that culture to hardware. An example hardware that is open to user appropriation is Ableton Push (PushMusic at your fingertips. n.d.) It works alongside the previously mentioned music software Ableton Live ("LiveCreate, Finish, Perform", n.d.) that is designed with appropriation in mind. It remains possible to further explore this opportunity in the future.

There have also been some experiments to take automation to an online synchronization level by accumulating user data in an online database. In the case of "CommunityCommands", this data is used to build a recommender system for the local user (Matejka, Li, Grossman, & Fitzmaurice, 2009) whereas with "ingimp", accumulated data is stored as a means to profile user types. (Terry, Kay, Van Vugt, Slack, & Park,2008) In this research, a layer of artificial intelligence in the form of a recommender could have been implemented to learn about individual users and encourage them to move from the safety of the defaults to popular alternative modules. Accumulating enough information this way, the recommender would eventually start making more individually appropriate suggestions.

Finally, although the solutions in this research are proposed specifically for creative software, the underlying principles may be genericized and applied to any software project in general. This too, is more feasible to do once a prototype has been built. For now, these opportunities are reserved for the future.

APPENDIX A:
PARTICIPATION ASSETS

INTERVIEW QUESTIONS

I) HOW DO WE LEARN THROUGH SOFTWARE USAGE?

Amongst contemporary image creation software, which one do you possess the most experience with (will be referred as <*>)?
Out of a scale of 5, how would you rate your level of experience with <*>?
How many years of experience do you have with <*>?
What change would make you grade yourself higher? What are you lacking?
What percentage of the functions in <*> are you experienced with?
Do you use the GUI of <*> as it is or do you switch between built-in, premade interface alternatives?
Do you customize the interface by moving the buttons and panels around, according to your needs?
Do you create and store your own interface sets?
Do you customize the keyboard shortcuts?
Amongst contemporary scripting languages, on which one do you possess the most experience?
How many years of experience do you have with this language?
Out of a scale of 5, how would you rate your level of experience?

II) HOW DO WE GET DESIGNERS TO BUILD THEIR OWN TOOLS?

Have you heard of development related concepts such as SDKs, version controlling, forking?
Do you know their meaning? If not, do they sound approachable?
Have you considered appropriating the functions of the software beyond interface customization?
Have you ever wanted a function from another software to exist and work the same way in <*>?
Have you ever wished that a new and original function that you imagined on your own existed in *>?
Have you searched online to see if the functions of a software product could be appropriated?
Have you searched to see if you could appropriate a software with a function that you imagined?
Were these queries done in search engines or in particular specialized online communities?
How far did you take this idea? Why / Did you give up?
What kind of technical shortcomings did you encounter on the way?
What kind of social shortcomings did you encounter on the way?

III) HOW DOES SOCIAL LEARNING WORK FOR SOFTWARE?

Is your learning, accelerating, decelerating (or neither) since you started using <*>?

What triggers your interest to learn more of <*>?

Can you find your own way in learning <*>?

Do you watch video tutorials for <*> online?

What makes you want to watch video tutorials?

Do you prefer official & professional or amateur video tutorials by community members?

Do you visit online communities, such as forums about <*>?

What makes you want to visit those communities as a reader and learner?

What do you learn from those places?

Do you visit online communities for software developers, such as Github?

What makes you want to visit those communities as a reader and learner?

What do you learn from those places?

IV) HOW DO MENTAL BARRIERS EFFECT PARTICIPATION?

Do you contribute to designer communities with your own posts?

Do you contribute to developer communities with your own posts?

Do you post answers to other people's questions in either community at all?

Would you do so if there were no experts around.

Would your decision change if the online community was moderated by volunteers?

Do moderation policies have an effect on your incentive to contribute?

Can you name a few personal motivators and demotivators in contributing?

ANSWER SET 1

A - "Professional tutorials impose a certain way of using a program. Videos from volunteers are better for learning appropriation tricks that aren't intended by the developers. I learn more from volunteers."

B - "I only watch video tutorials made by the volunteers. I would only contribute to a volunteer community. If there are people who get paid for being there, I expect them to respond instead. Paid staff on the official forums are able to fake credibility and knowledge behind the brand identity and I don't like that."

C - "I don't learn from Adobe because they advertise themselves too much but Lynda is straightforward. The popularity of volunteer-made tutorials prove their credibility. I follow Quora because the community experts are credible. Credibility equals clarity."

ANSWER SET 2

A - "Software development would actually excite me more than learning about design software because I don't know anything about it and everything I'd learn would be new knowledge."

B - "Once I imagined a way the clipboard (copy paste) function could work with multiple items at once. I talked to a friend about it and we concluded that it could be done somehow but I can never do it myself because this must require coding and I don't know how to do it. "

C - "I don't want to design a specific function but I want to glitch and randomize them. I also want to explore new interaction methods like holding interface elements and shaking the mouse. "

ANSWER SET 3

A - "Development terminology belongs to a world I'm not a part of and I can't be a part of as it is. The concepts should be expressed in ways that are more welcoming to the layman."

B - "I hesitated to ask online since no platform exists that I can comfortably talk about these things. Everybody must be asking such things. The terminology in the development world alienates me. I would do it if there was a graphic user interface to do it with."

C - "I tried to penetrate Github but I couldn't figure out how it worked at all so I couldn't socialize there. It's too esoteric for non developers. There is no such tutorial on “how to Github”. The terminology is written in an alienating way for the purpose of leaving the layman outside the circle. It is as if they're asking “if you’re not a developer, why are you here?”"

ANSWER SET 4

A - "If I were less experienced I would choose "nodes" because it shows everything at once. Combining "events" and "scripting" would allow me try new things safely. It makes me happy to modify values without learning the syntax."

B - "If I were more experienced I would choose "scripting". I would enjoy being able to switch between the methods like switching between viewports."

C - "I would combine "events" and "scripting" because it would be exciting to design with a GUI and modify the code afterwards."

X - "I would switch to "scripting" for precision. If I were less experienced, I would choose "nodes" because it is still precise. Combining "blocks" and "scripting" would allow anyone try new things easily."

ANSWER SET 5

A - "I would mix A and B because B is too complex but I still want to see numeric values. I would use B for more complex functions such as 3D transform.

B - "They both seem legible even though coding means "illegibility" to me. Commands in B are simple enough to appeal to me. I would use A as the default and B for precision."

C - "I'd like to switch between methods like photoshop presets. I would use A for casual projects, B for precision. I'd use both over the default. This is a promising idea since existing the development formula is designed for company growth, not the users' needs."

X - "I would use A for casual projects, B for precision. I'd prefer both over the defaults because they're both more immersive."

ANSWER SET 6

A - "I realized that the existing method requires unnecessary steps in the beginning. Your alternatives made me think about using arrow keys for precision."

B - "I realized that I were clicking and moving around more than necessary. I wish these new tools existed but they're hard for me to create. If they could be created with visual programming I would make them."

C - "I realized that the existing method was imprecise the second time I used it."

X - "I'm noticing that the default method feels very stupid the second time I used it. I'm having difficulty with it because now I know there could have been an easier way."

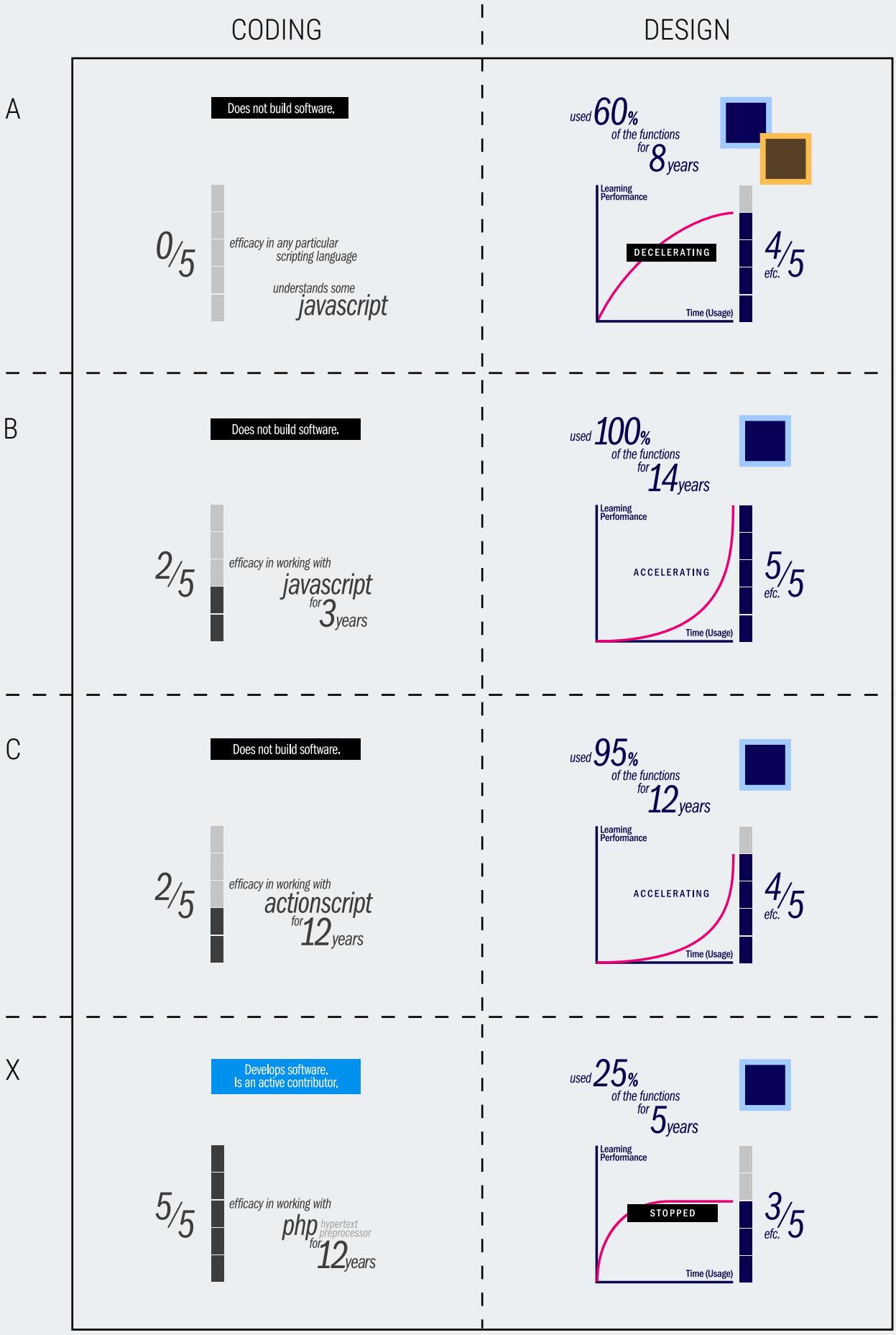
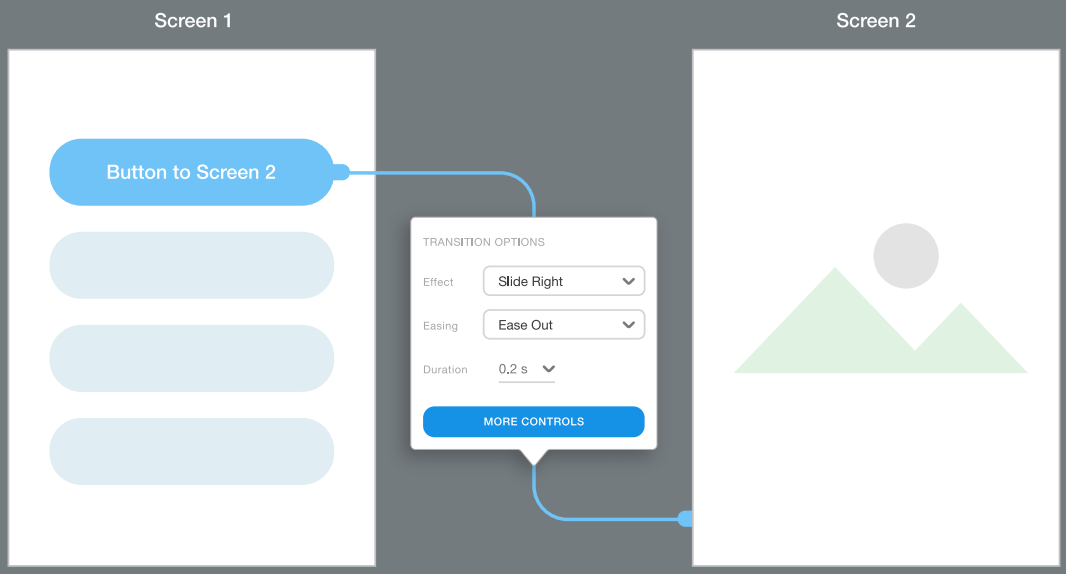
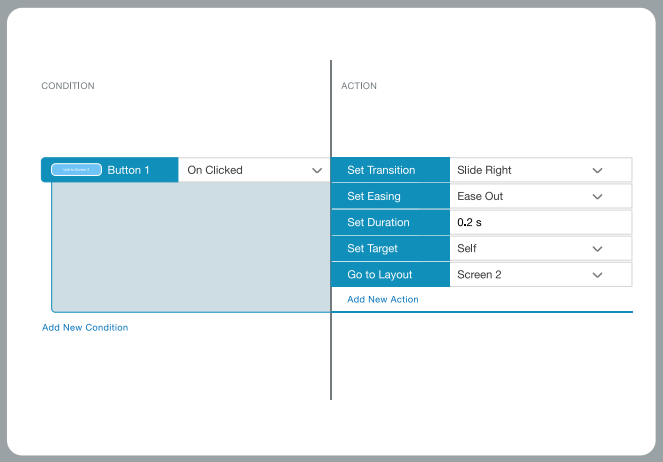


Figure 7: A visual representation of the participant profiles.

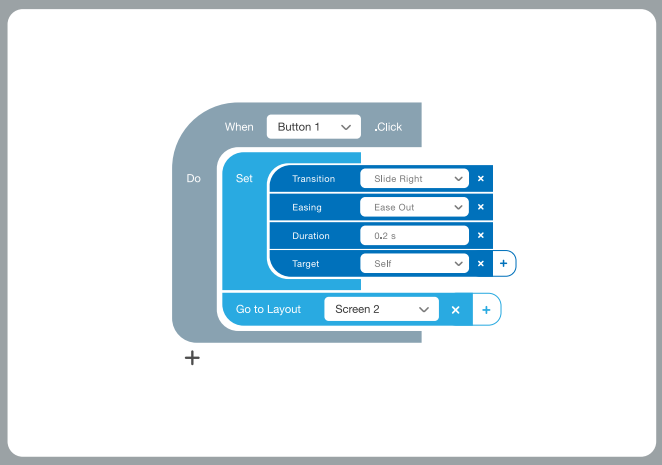


Scripting

Events



Blocks



Nodes

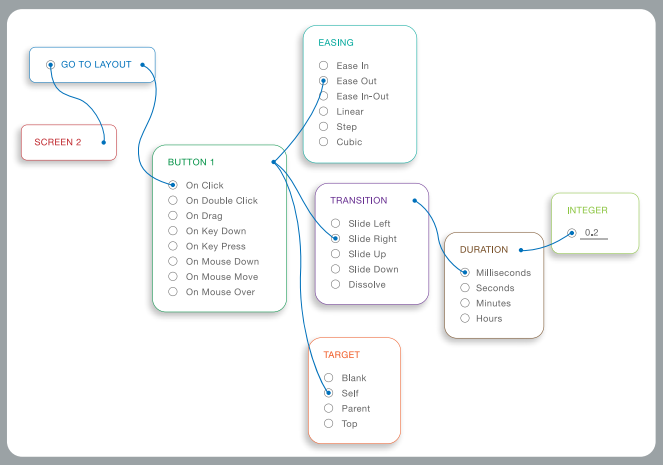


Figure 8: The assets used during the low-tech visual programming exercise. Top: Researcher's rendering of Adobe Experience Design CC software.

PRODUCTIVITY
85%
SUSTAINED USE
65%
LEGIBILITY
45%

PRODUCTIVITY
80%
SUSTAINED USE
90%
LEGIBILITY
80%

PRODUCTIVITY
70%
SUSTAINED USE
65%
LEGIBILITY
75%

PRODUCTIVITY
75%
SUSTAINED USE
60%
LEGIBILITY
65%

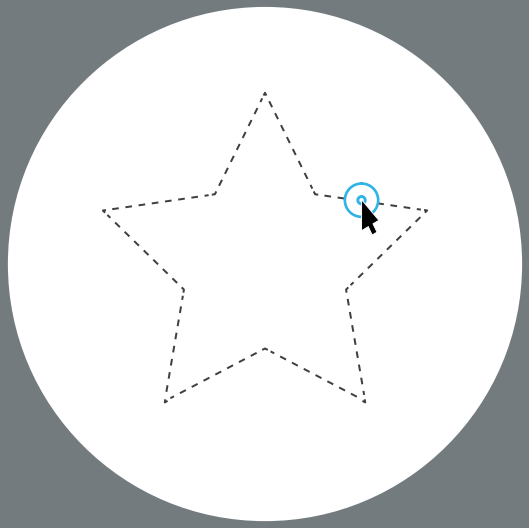
A - Difficult
B - Boring
C - The core
X - Not for everyone

A - Precise
B - Familiar
C - Inviting
X - Familiar

A - Disfunctional
B - Complex
C - Convoluted
X - Accessible

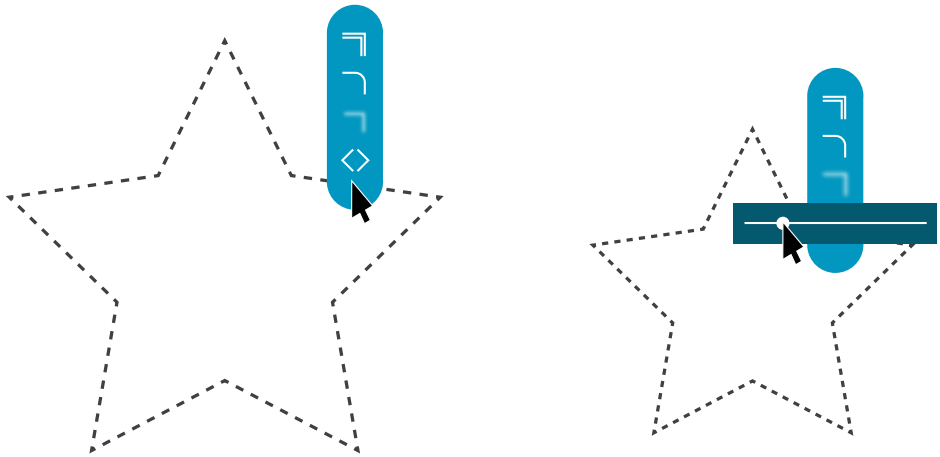
A - Conceptual
B - Neutral
C - Too much
X - Difficult

Figure 9: Some of the outcomes from the low-tech participatory exercise.



UX & Function Alternatives for Modifying Selection

A



B

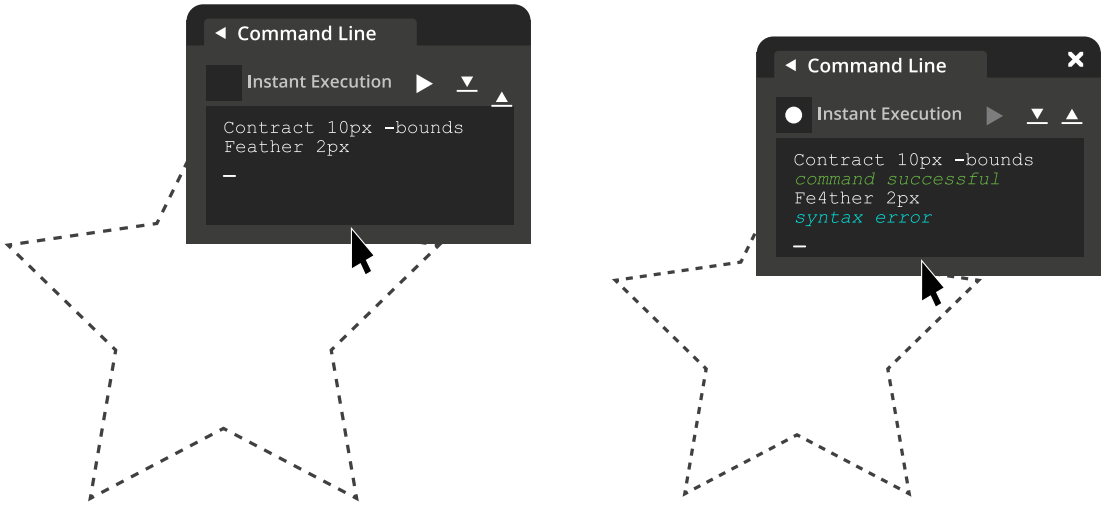
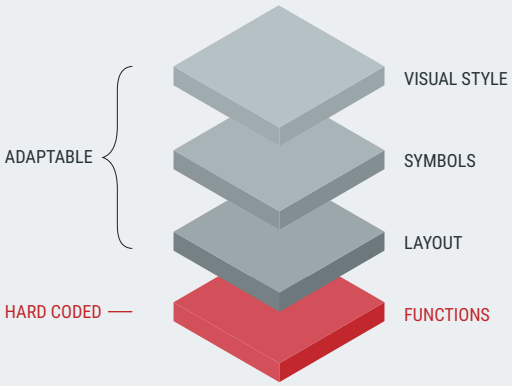


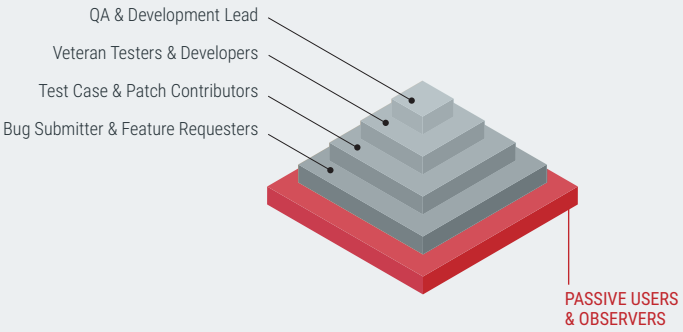
Figure 10: The assets used for the modularity exercise.

APPENDIX B:
DESIGN ASSETS

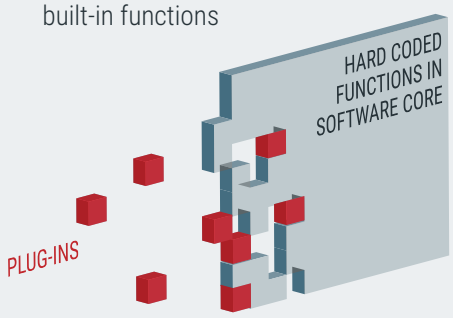
Functional customization is impossible.
User adaptations are superficial
interface adjustments.



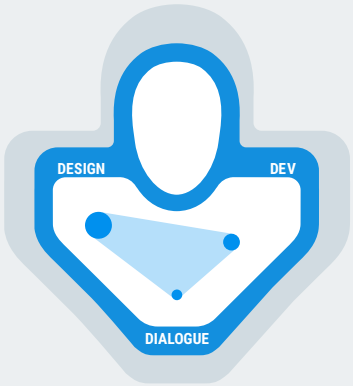
Laypeople can not penetrate
the world of open source
software development



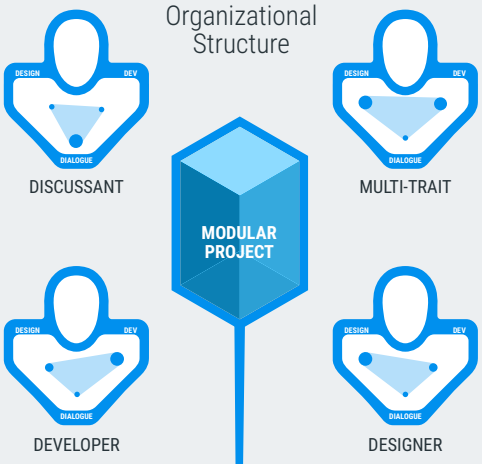
Modularity can not be used
as a means to challenge
built-in functions



Social Collaborator
Profiles



Organizational
Structure



Modular Local
Client Adaptation

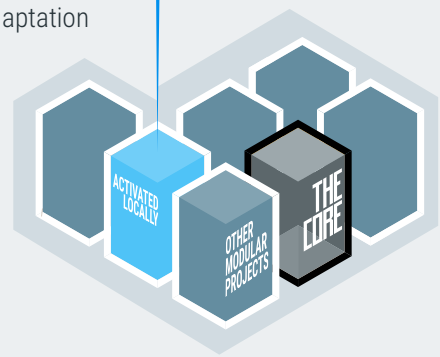
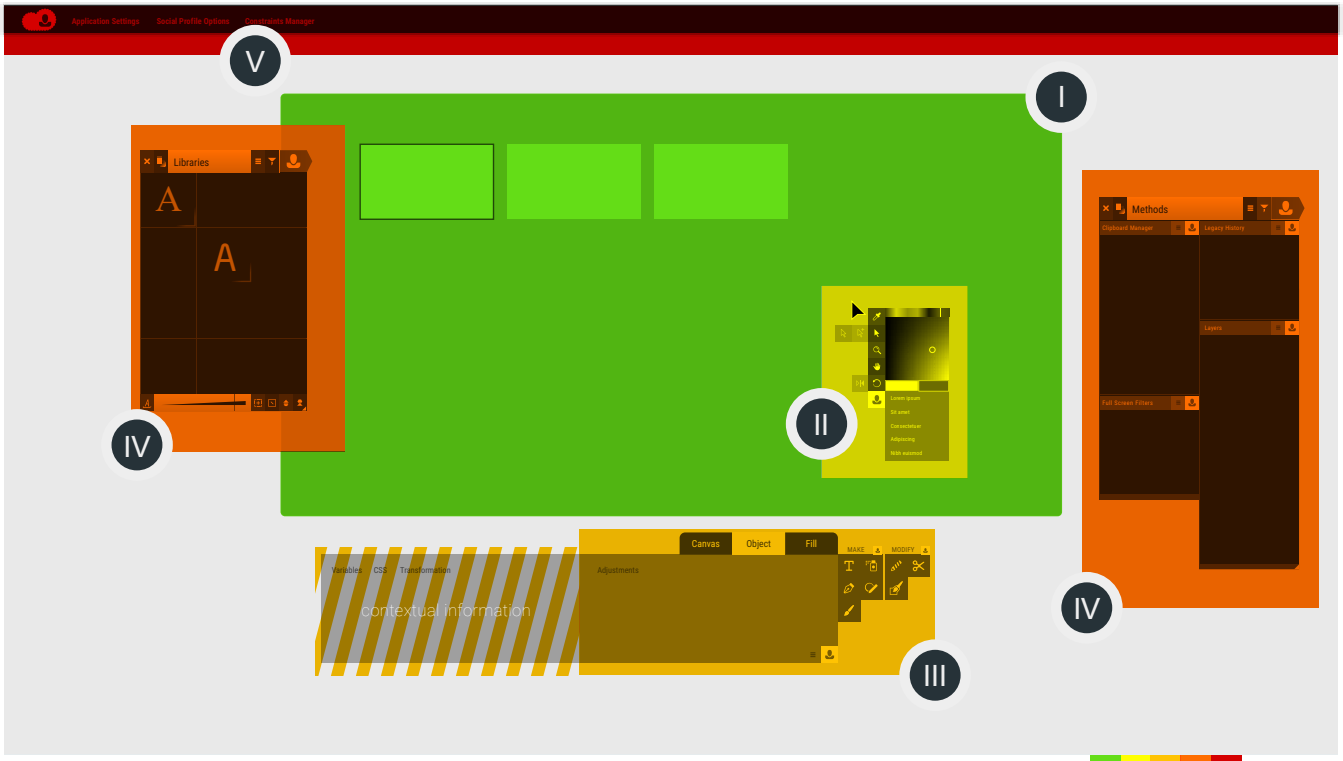


Figure 11: A comparison between the existing software development culture (left) and the changes formulated with this thesis (right).



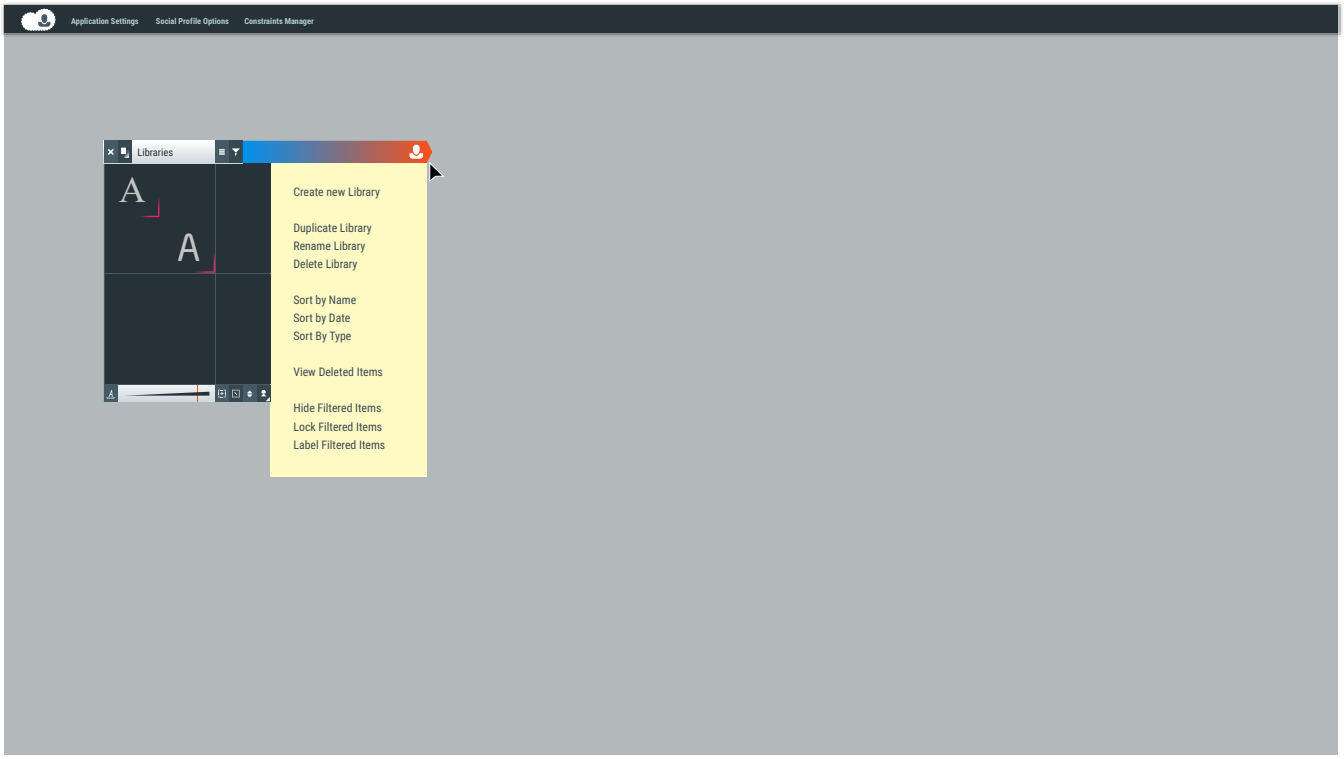
CREATION ENVIRONMENT
LEVELS OF TANGIBILITY

MOST

LEAST

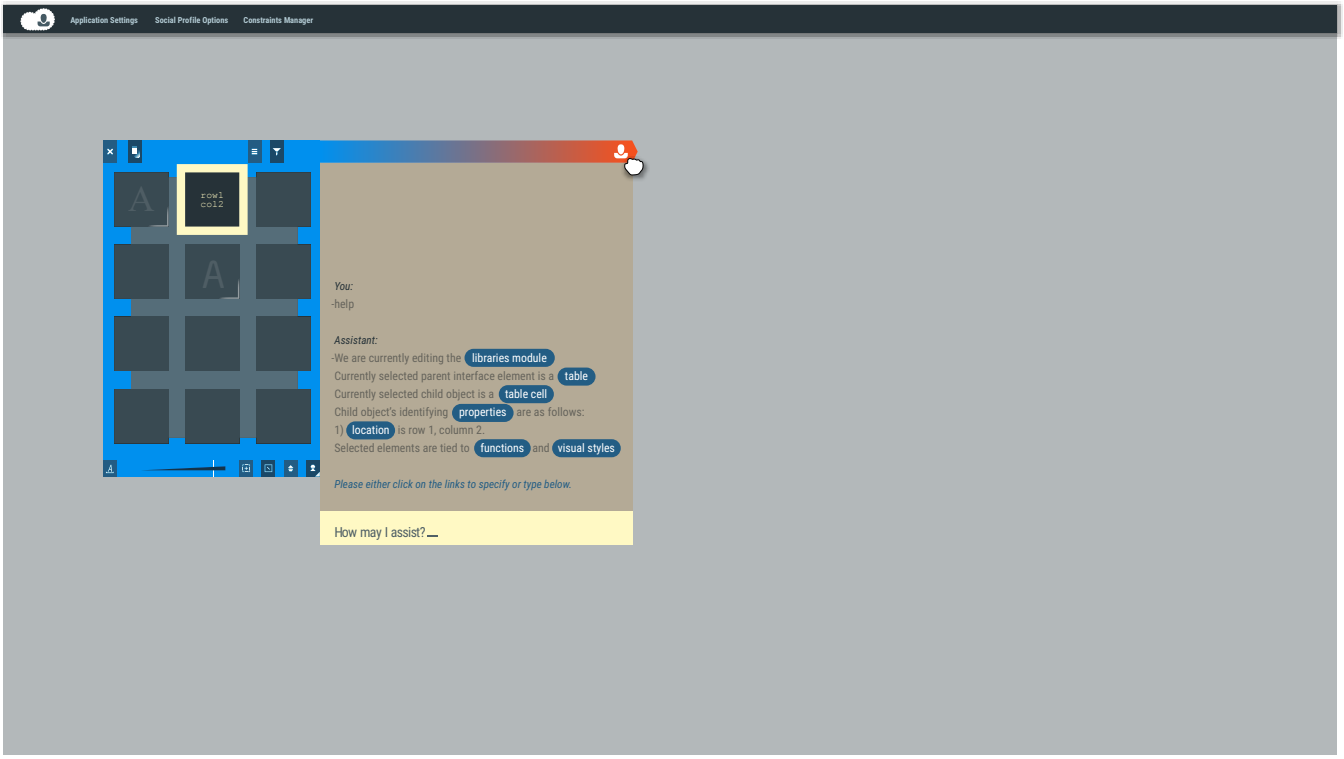
ENVIRONMENT 1:
CREATION

Figure 12: The default appearance of the creation environment with the concept of -levels of tangibility- visualized.



TRANSITION BETWEEN
CREATION &
INDIVIDUALIZATION

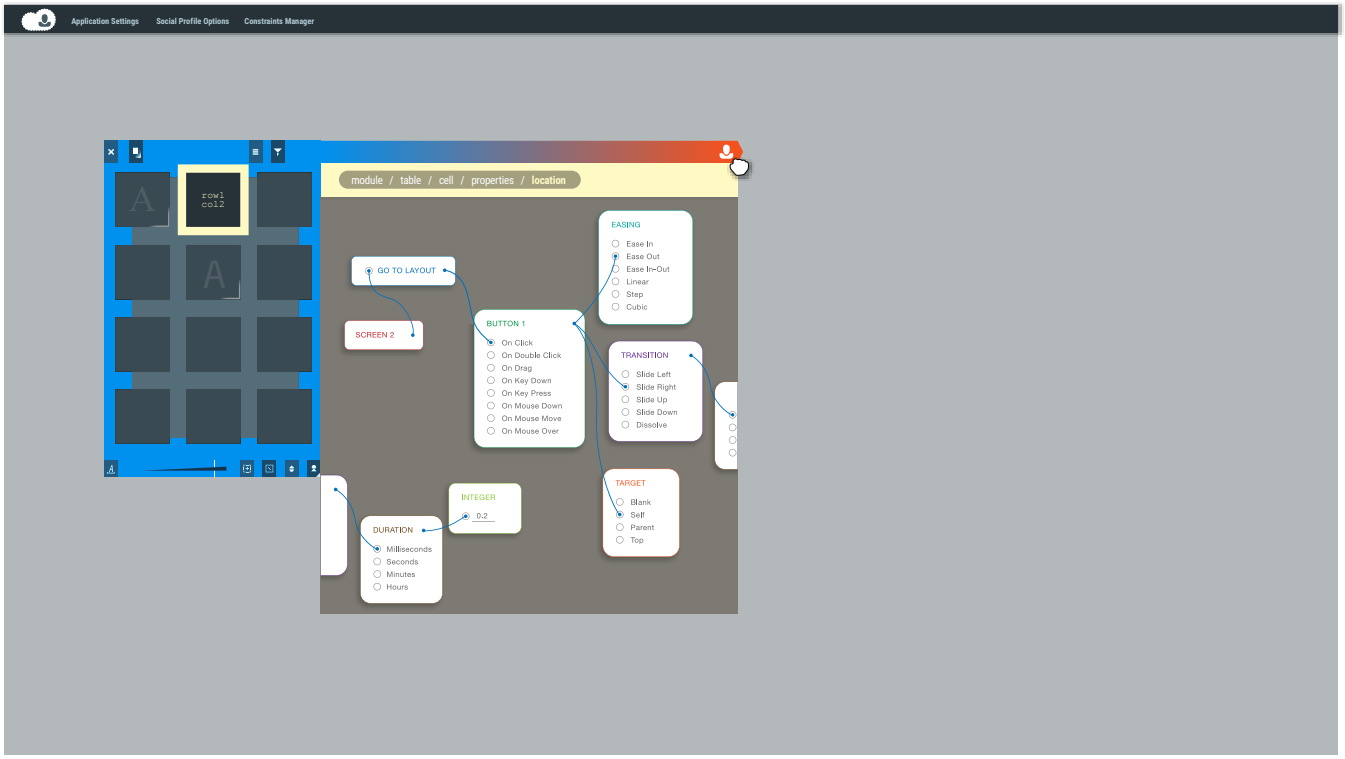
TOOL OPTIONS



ENVIRONMENT 2:
INDIVIDUALIZATION

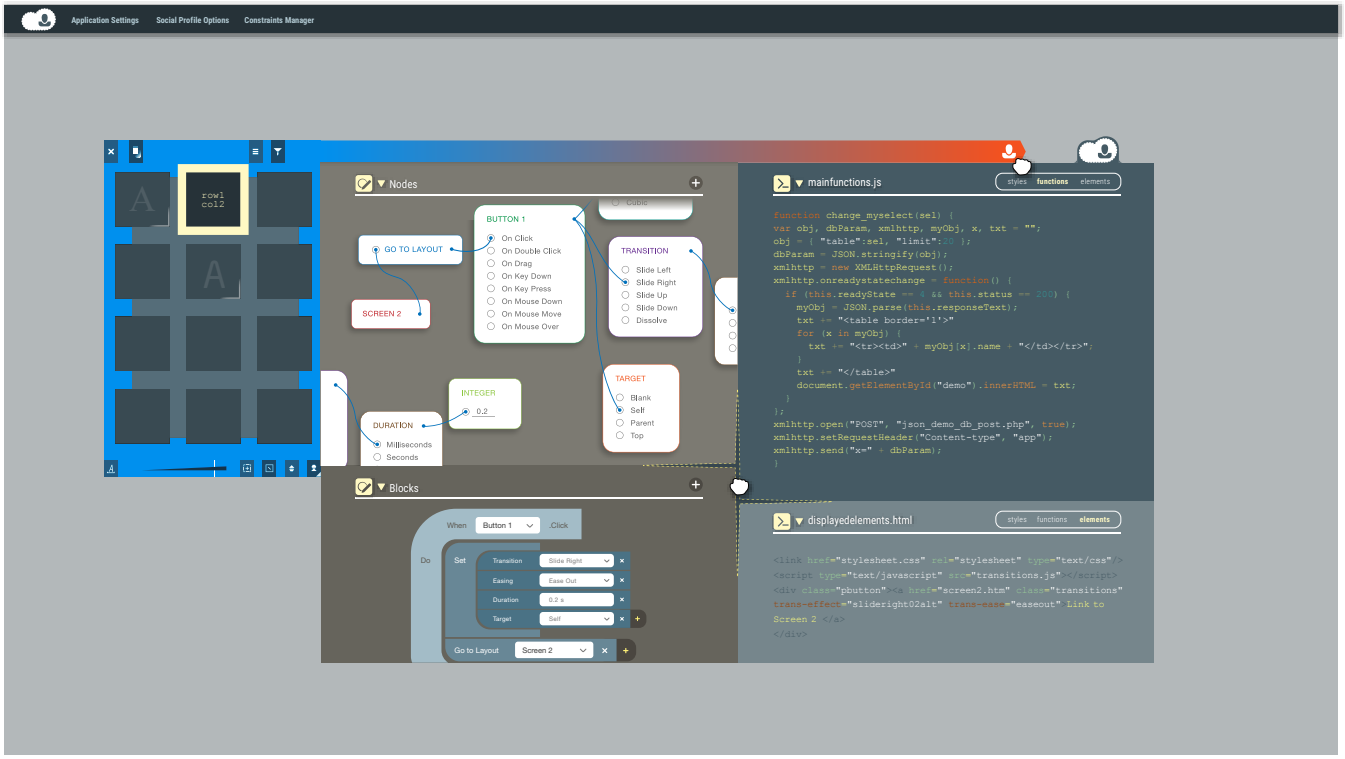
INTERACTIVE
ASSISTANT MODE

Figure 13: The initial transitionary phases of the individualization environment.



ENVIRONMENT 2:
INDIVIDUALIZATION

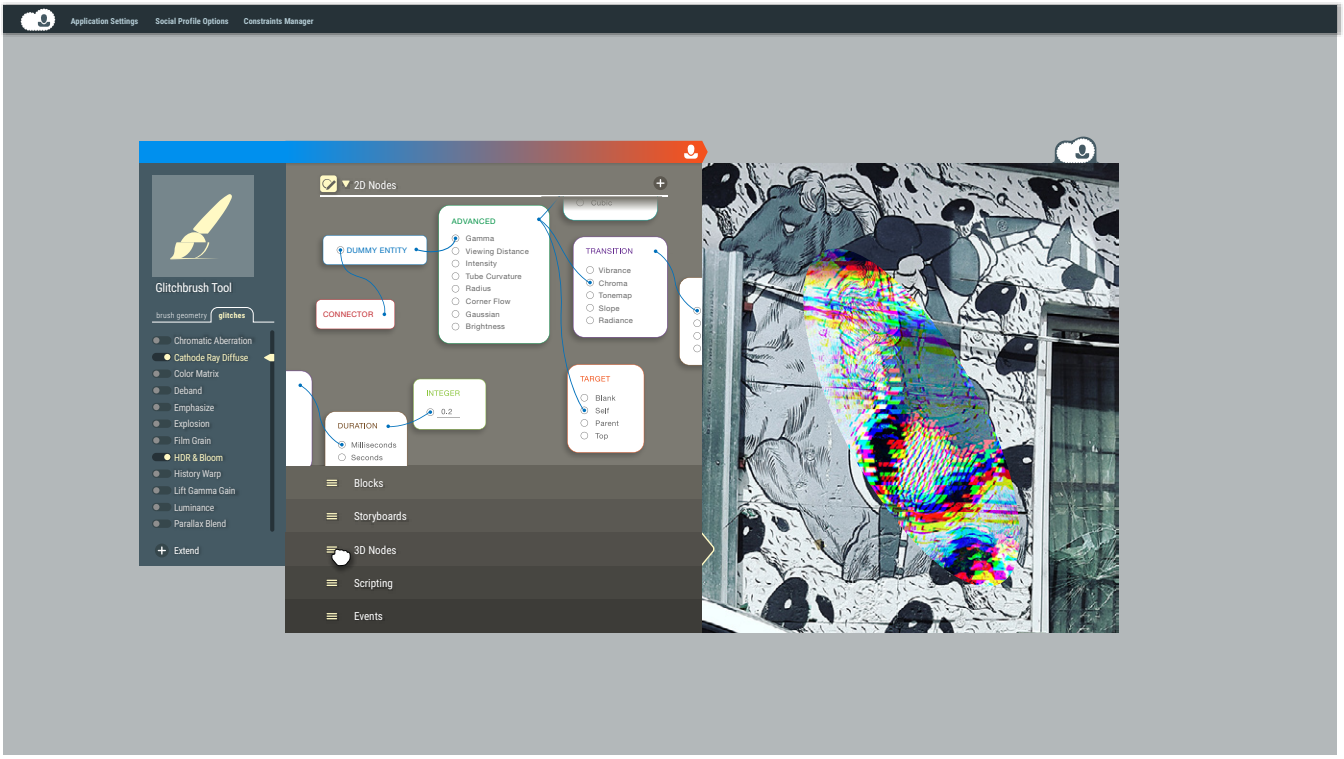
EXTENDING VIA
CODING ABSTRACTIONS



ENVIRONMENT 2:
INDIVIDUALIZATION

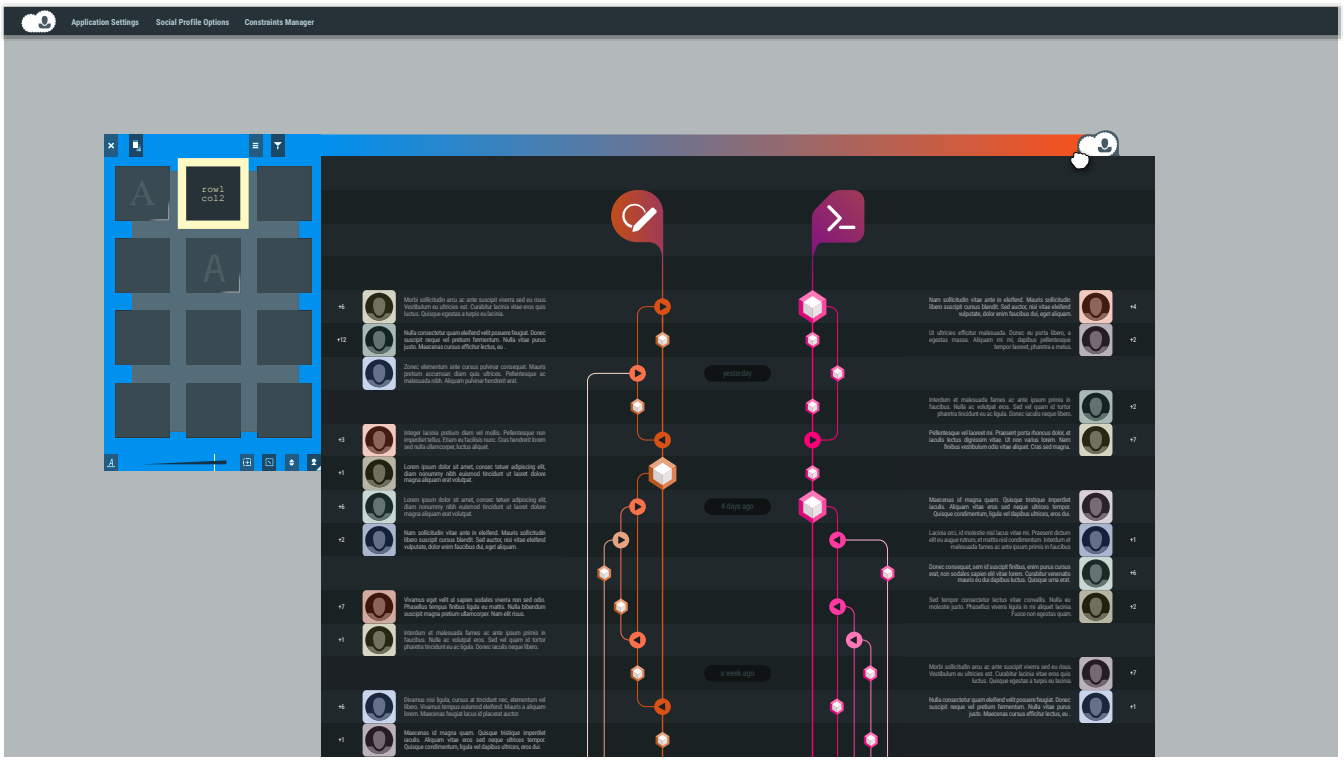
PROGRAMMING ALONGSIDE
MULTIPLE ABSTRACTION METHODS

Figure 14: Advanced modes of engagement from the individualization environment.



ENVIRONMENT 2 ALTERNATIVE:
TOOL INDIVIDUALIZATION

GLITCHBRUSH TOOL
WITH PREVIEW



ENVIRONMENT 3:
SOCIALIZATION

DESKTOP MODE

Figure 15: An alternate view from the individualization environment (top) and the transition to the social environment (bottom).

Figure 16: The social environment as it is accessed from a mobile device.

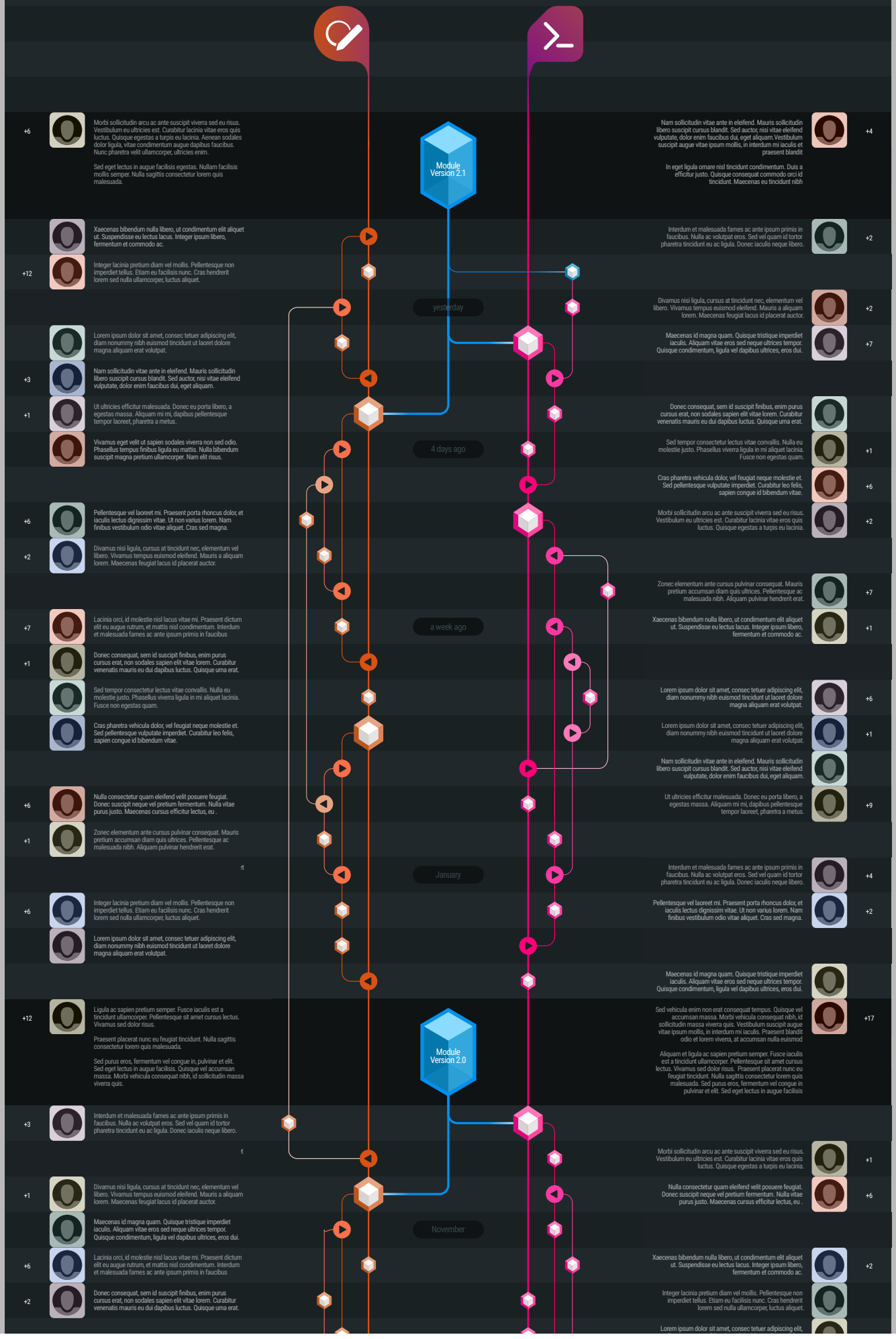


Figure 17: An expanded view of the social environment.

REFERENCES

A hackable text editor for the 21st Century. (n.d.). Retrieved March 16, 2017, from https://atom.io/

Belin, A., & Prié, Y. (2012). DIAM: Towards a Model for Describing Appropriation Processes Through the Evolution of Digital Artifacts. In Proceedings of the Designing Interactive Systems Conference (pp. 645–654). New York, NY, USA: ACM.

Benyon, D. (1993). Accommodating individual differences through an adaptive user interface. Human Factors in Information Technology, 10, 149–149.

Benyon, D., & Murray, D. (1993). Developing Adaptive Systems to Fit Individual Aptitudes. In Proceedings of the 1st International Conference on Intelligent User Interfaces (pp. 115–121). New York, NY, USA: ACM.

Bunt, A., Conati, C., & McGrenere, J. (2007). Supporting Interface Customization Using a Mixed-initiative Approach. In Proceedings of the 12th International Conference on Intelligent User Interfaces (pp. 92–101). New York, NY, USA: ACM.

Cheung, G., Chilana, P., Kane, S., & Pellett, B. (2009). Designing for Discovery: Opening the Hood for Open-source End User Tinkering. In CHI '09 Extended Abstracts on Human Factors in Computing Systems (pp. 4321–4326). New York, NY, USA: ACM.

Dix, A. (2007). Designing for Appropriation. In Proceedings of the 21st British HCI Group Annual Conference on People and Computers: HCI...But Not As We Know It - Volume 2 (pp. 27–30). Swinton, UK, UK: British Computer Society.

Electron. (n.d.). Retrieved March 26, 2017, from https://electron.atom.io/

Ellis, T. O., Heafner, J. F., & Sibley, W. L. (1969). The GRAIL Project: An Experiment in Man-Machine Communications. Santa Monica, California: Rand Corporation.

Granić, A., & Nakić, J. (2010). Enhancing the Learning Experience: Preliminary Framework for User Individual Differences. In Proceedings of the 6th International Conference on HCI in Work and Learning, Life and Leisure: Workgroup Human-computer Interaction and Usability Engineering (pp. 384–399). Berlin, Heidelberg: Springer-Verlag.

Hughes, J. (1989). Why functional programming matters. The computer journal, 32(2), 98-107.

Hurst, A., Hudson, S. E., & Mankoff, J. (2007). Dynamic Detection of Novice vs. Skilled Use Without a Task Model. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (pp. 271–280). New York, NY, USA: ACM.

Jensen, C., & Scacchi, W. (2007). Role Migration and Advancement Processes in OSSD Projects: A Comparative Case Study. In Proceedings of the 29th International Conference on Software Engineering (pp. 364–374). Washington, DC, USA: IEEE Computer Society.
Kay, A. C. (1972). A Personal Computer for Children of All Ages. In Proceedings of the ACM Annual Conference - Volume 1. New York, NY, USA: ACM. https://doi.org/10.1145/800193.1971922

Kay, A. C. (1996). The early history of Smalltalk. In T. J. Bergin & R. G. Gibson (Authors), History of programming languages II (pp. 511-598). New York, New York: ACM Press. doi:10.1145/234286.1057828

Kay, A. C. (2012). The Future of Reading Depends on the Future of Learning Difficult to Learn Things. In B. Junge, Z. Berzina, W. Scheiffele, W. Westerveld, & C. Zwick (Eds.), The digital turn: design in the era of interactive technologies. Berlin: ELab/Weißensee Academy of Art Berlin.

Lin, L.-C., Scull, C., & Walsh, D. (2012). Focusing Our Vision: The Process of Redesigning Adobe Acrobat. In CHI '12 Extended Abstracts on Human Factors in Computing Systems (pp. 629–644). New York, NY, USA: ACM.

LiveCreate, Finish, Perform. (n.d.). Retrieved March 16, 2017, from https://www.ableton.com/en/live/

Matejka, J., Li, W., Grossman, T., & Fitzmaurice, G. (2009). CommunityCommands: Command Recommendations for Software Applications. In Proceedings of the 22Nd Annual ACM Symposium on User Interface Software and Technology (pp. 193–202). New York, NY, USA: ACM.

McCullough, M. (1998). Abstracting craft: the practiced digital hand. Cambridge, Massachusetts: MIT Press.

McGrenere, J., Baecker, R. M., & Booth, K. S. (2002). An Evaluation of a Multiple Interface Design Solution for Bloated Software. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (pp. 164–170). New York, NY, USA: ACM.

NW.js. (n.d.). Retrieved March 26, 2017, from https://nwjs.io/

Papert, S. (1980). Mindstorms: children, computers, and powerful ideas. New York, New York: Basicbooks.

PushMusic at your fingertips. (n.d.). Retrieved March 16, 2017, from https://www.ableton.com/en/push/

Reas, C., & Fry, B. (2007). Processing: a programming handbook for visual designers and artists. Cambridge, Massachusetts: MIT Press.

Shneiderman, B. (1983). Direct Manipulation: A Step Beyond Programming Languages. Computer, 16(8), 57-69. doi:10.1109/MC.1983.1654471

Shneiderman, B. (1997). Direct manipulation for comprehensible, predictable and controllable user interfaces. Proceedings of the 2nd international conference on Intelligent user interfaces - IUI '97, 33-39. doi:10.1145/238218.238281

Shneiderman, B. (2007). Creativity Support Tools: Accelerating Discovery and Innovation. Commun. ACM, 50(12), 20–32.

Solomon, C. J., & Papert, S. (1976). A case study of a young child doing turtle graphics in LOGO. Proceedings of the June 7-10, 1976, national computer conference and exposition on - AFIPS '76. doi:10.1145/1499799.1499945
Sutherland, I. E. (1963). Sketchpad. Proceedings of the May 21-23, 1963, spring joint computer conference on - AFIPS '63 (Spring). doi:10.1145/1461551.1461591

Tanenbaum, J. G., Williams, A. M., Desjardins, A., & Tanenbaum, K. (2013). Democratizing Technology: Pleasure, Utility and Expressiveness in DIY and Maker Practice. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (pp. 2603–2612). New York, NY, USA: ACM.

Terry, M., Kay, M., Van Vugt, B., Slack, B., & Park, T. (2008). Ingimp: Introducing Instrumentation to an End-user Open Source Application. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (pp. 607–616). New York, NY, USA: ACM.

Tufte, E. R. (2006). The Cognitive Style of PowerPoint: Pitching Out Corrupts Within. In Beautiful Evidence (pp. 156-185). Cheshire, Connecticut: Graphics Press.

Victor, B. (2012). Learnable Programming. Retrieved February 19, 2017, from http://worrydream.com/LearnableProgramming/

WordPress.org. (n.d.). Retrieved March 16, 2017, from https://WordPress.org/plugins/

WordPress Themes from ThemeForest. (n.d.). Retrieved March 30, 2016, from http://themeforest.net/category/WordPress

Zeidler, C., Lutteroth, C., & Weber, G. (2013). An Evaluation of Advanced User Interface Customization. In Proceedings of the 25th Australian Computer-Human Interaction Conference: Augmentation, Application, Innovation, Collaboration (pp. 295–304). New York, NY, USA: ACM.