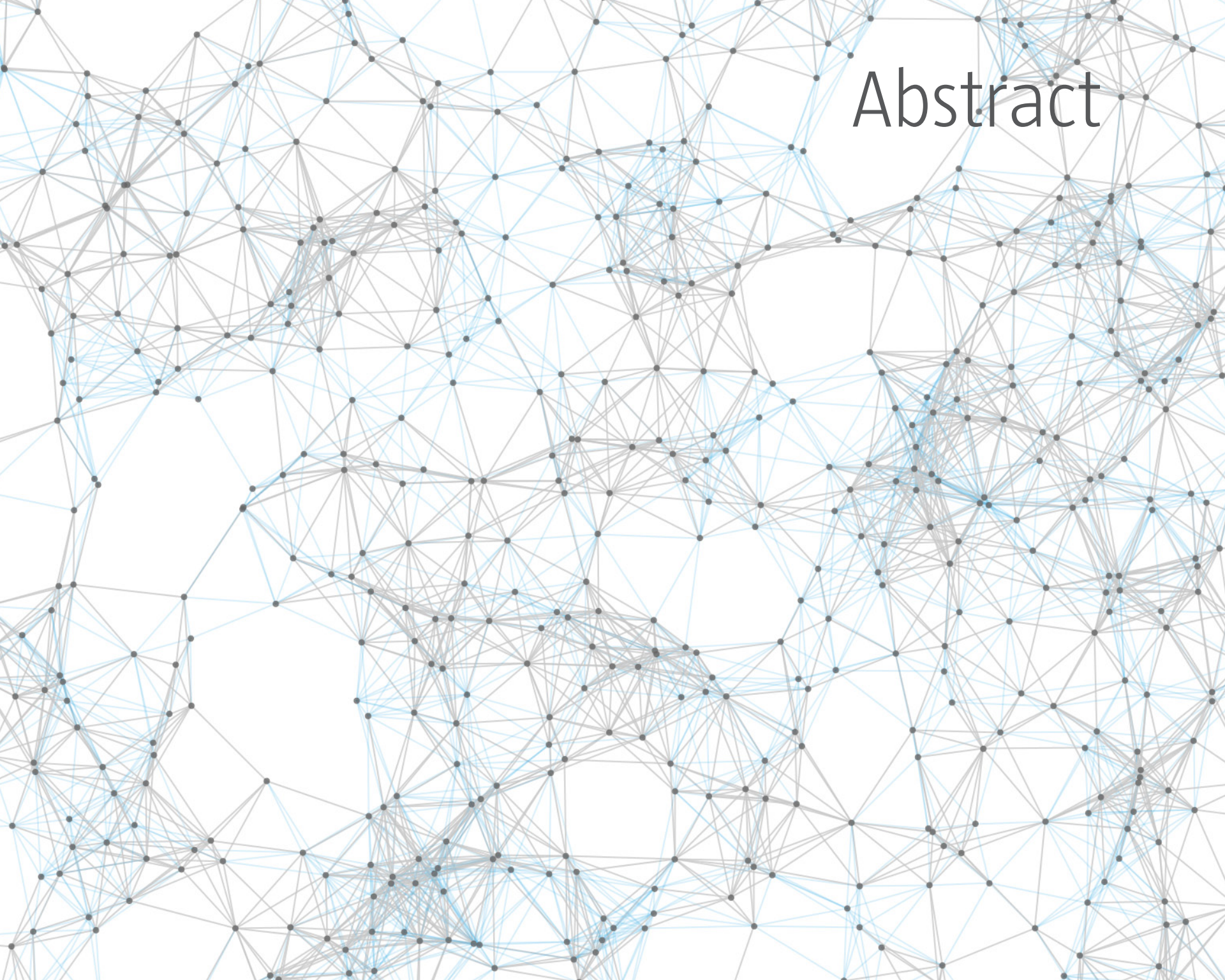
ATLAS

Practical Computational Literacy for Designers

John Gruen
Advisors: Gill Wildman + Nick Durrant

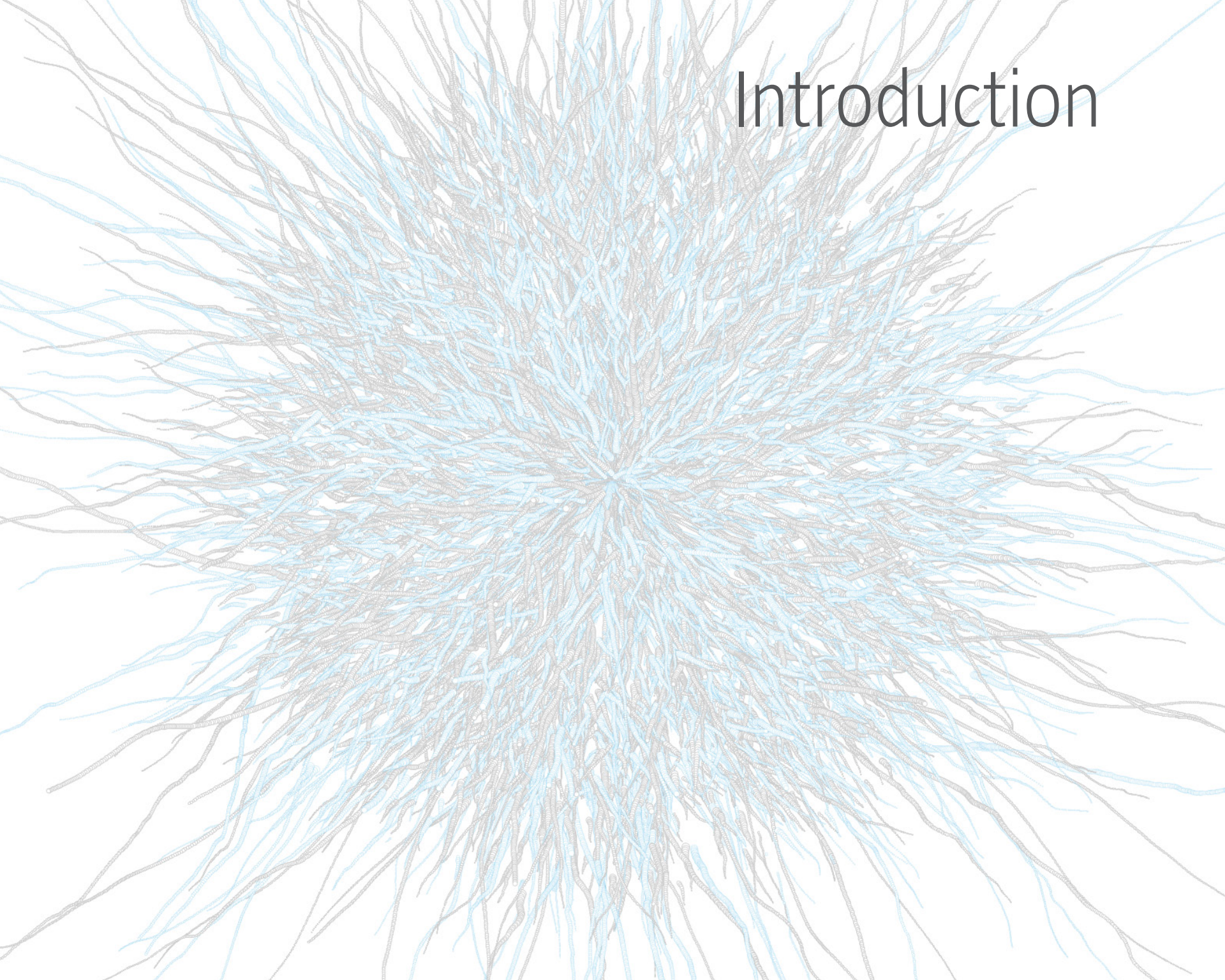
A thesis submitted to the School of Design, Carnegie Mellon University, for the degree of Master of Design in Interaction Design

Abstract



"Atlas: Practical Computational Literacy for Designers" is a masters thesis exploring the relationship between design learners and code. In this document, models of learning and motivation serve as a launch point for discussing goals and fears of designers as they relate to code and programming. Using interviews with design learners, design professionals, computer scientists, and developers, this document isolates three choke points for many design learners relating to code: lack of immediacy and feedback, lack of orientation, and an *a priori* fear of failure. It goes on to explain how how only the lack of immediacy can be fully compensated for within code-learning contexts, and suggests that the other two originate as contradictions in design learners attitudes about code's evolving relationship to design practice. The document concludes by introducing Atlas, a system to help designers model real-world computational systems and better articulate their goals and interests relating to these systems.

Introduction



A Challenge for Designers and Code

Prior to returning to school to pursue a Masters in Interaction Design, I spent several years teaching programming courses to young creative professionals at 3rd Ward, a New York City-based Design Studio. In my time there I led hundreds of students in courses on HTML and CSS, JavaScript, and Processing. The former three languages comprise a set of tools used to create nearly every website in the world. Processing serves a different purpose entirely; it is a wrapper on Java designed to give artists the ability to code for creative ends. Processing is wonderful for many things, but it would be a terrible language with which to make websites.

As a teacher, I did my best to ensure that my students understood the syntax and structure of whatever language they were learning. I was quite chagrinned when, on more than one occasion, a student in one of my Processing courses would approach me several weeks in, and ask when we would begin building websites.

This was my first exposure to a problem that this thesis attempts to solve. By design, my courses were built around the acquisition of grammatical knowledge. I taught language syntax, but lost in my teaching was any though of preemptively and explicitly establishing a link between student goals and a particular course of study. My students believed that they ought to learn to learn to code, but as a teacher, I failed them by not asking at the outset, why? For what purpose did they want to learn? What did they want to make? What were their goals? Were they right class? Was an introductory programming course even the most expedient way to help them achieve their goals?

A design field changes over time. New tasks appear carrying new demands and challenges created by new hopes and desires. Taken together with more concrete changes-such as new material, technology, and additions to the body of knowledge of the field- we find that traditional assumptions concerning competence are constantly undermined. It is also common that style and preferences change over time, thereby influencing what is thought of as good design. Consequently, design ability becomes a moving target.

-Jonas Löwgren & Erik Stolterman

What is this Document About?

The practices and artifacts that emerge from computation deeply affect contemporary design practice. For many designers such products may constitute both the tools and products of practice. While design methodology and practice cannot be reduced to the sum of its digital components, it is nevertheless within the digital space that many designers find avenues for professional practice. Put simply, many of us use some set of digital tools to design digital products and services.

However, the digital tools designers use (a) software packages reliant on packaged graphical user interfaces (b) tend to emerge from traditional modes of design. The Adobe Creative Suite, (recently changed to Creative Cloud, a clear sign of the times), among the most well used digital design tools, consists of a set of applications which mimic and vastly extend on analog tools and traditional design principles. Using a term borrowed from computer science, we might say that such software tools afford top-level abstractions for their users.

In computer science, an abstraction is a "model for thinking about a problem and... the appropriate mechanizable techniques to solve it." Abstractions are frames for problems which allow the most concise trajectory toward a solution while masking underlying implementation details. As computer scientist Andrew Koenig put it: "abstraction is selective ignorance."

I describe traditional design software as affording a top-level abstraction because within a computing system, the user interface such software provides 'sits on top' of a heterogenous set of underlying technologies, and simultaneously allows users to operate in perfect ignorance of these substrata.

Design learners are examining their assumptions around the acceptability and expediency of operating solely in the topmost abstraction layer of computational devices. My research indicates several commonly held perceptions driving this examination. Primary among them:

- The perception of environmental and market pressures to acquire new hard skills.
- The perception of communication dynamics between developers and designers in industry.
- The perception of the expediency of programming tools for prototyping experiences on broadening range of forms in digital devices.
- The perception of code as a potential outlet for creative experiment and play.

Underlying all of this is a simple fact that as design learners, we are naturally curious about the world in which we operate and the systems that make that world work. As one interviewee put it: "we like to know where they are and what's around us."

As indicated in the foreword above, many learners seeking such an understanding lack the ability to articulate or discern what they want to know and how to go about acquiring this knowledge. They are venturing off of the well-defined map of traditional design practice, and lack, to borrow from Ernest Hemingway via Howard Rheingold "internal crap detector[s]." Because of this, they are unable to separate signal from noise and develop sensibilities and understandings necessary for the task they wish to accomplish.

This is precisely why so many of my students came into the classroom with only a vague sense that they should 'learn to code.' My students were operating proactively on an ill-defined set of assumptions about the how to achieve their desired goals, and as a teacher I failed to help them clarify these assumptions a priori. Unfortunately, this mirrored set of failings on the part of learners and teachers has emerged as a consistent pattern in my interviews with design learners.

This thesis is about proposing a corrective to the current situation that allows learners to better understand and articulate their needs and goals with regard to computational systems, and to make informed decisions about how and what to learn.

For some designers, learning one or more programming languages will be a goal. Undeniably, this is a challenge in itself; however, tools and procedures for streamlining and simplifying this process are currently being developed across a number of media types.[1] While I will discuss some of these tools briefly, my focus is on helping creative learners, and interaction designers in particular, gain a computational literacy focused on a plain language understanding of computational systems relevant to their needs and goals more generally.

Rather than focusing on how to designers might better learn to code, this document proposes a system to allow designers to abstract system-level computational understanding while deemphasizing the formal details of one or another programming language. Three interrelated concerns provide both the motivation for, and the structure of this document.

- Understanding exactly what designers want to know about computational systems, and why
- Understanding how current coding education fail to fully meet the needs of designers, and often compromise learning outcomes.
- Proposing a system which fulfills the expressed and latent needs of design learners to provide a richer, more holistic path to computational literacy for designers.

My Thesis Year

Process Summary

Perhaps inevitably, the focus of my research shifted over the course of the past year. In the beginning, I spent several months examining how to help designers better grapple with the challenges of learning to code: problems like gaining better understanding of syntax and structure. Through interviews and directed story telling with students and design professionals, I began to question the assumptions underlying this work. While this phase of my research helped me identify the challenges designers face learning to code, it also suggested a broader set of concerns around vocabulary, goal articulation, wayfinding, and decision making for design learners.

I explored these larger problems in the second half of my thesis research through a number of strategies:

- Interviews with computer scientists and developers to better understand the nature design's professional relationship with these parties and how it might affect my prototypes.
- Posting questions about programming on posters and whiteboards around the CMU School of Design , and inviting participation and feedback from a broader design audience.
- Handing out design probes to assess designer attitudes about designers and developers and about understanding alien concepts.

In the final phase of my design process, I created and testing several low fidelity prototypes with designers. From these, I developed a final prototype for Atlas, an online system to help designers gain a plain-language computational literacy.

What follows consists three sections. First a brief review of some literature relevant to my thesis project, second a discussion of my process, findings and prototype, and finally a set of concluding considerations for further study.

Literature Review

The research this year relied heavily on concepts from the disciplines of design, learning sciences, motivational theory, and computer science. Thus, the research strategies employed might best be described as 'mercenary.' Rather than an exhaustive exploration of any one set of literatures or past studies, this works makes strategic reference to the relevant work of each as a means of synthesizing a rigorous approach to computational literacy for designers.

Writing on two related topics in particular--novice learners and motivation--deeply informed my design synthesis.

Novice Learners

My research necessarily concerns the nature of abstract challenges facing novice learners. Particularly relevant is Mary McCarthy's 4Mat model of learning suggesting that new learners proceed along a trajectory of experience, reflection, conceptualization, action, and finally integration of new knowledge. The learning process emerges from the successful completion of this cycle. Macarthy extends on this with the contention that different types of learners will begin by analyzing their experiences with different questions, specifically: why, what, how and what if.

Complementing this, Susan Ambrose signals that because they lack knowledge, the ability of novice learners to identify salient events or ideas within a given context is highly limited. Further, the sparse knowledge networks of novice learners hinder their ability to expressly articulate needs. One symptom of this lack is an inability to proactively search for appropriate learning guides (Rheingold). Another is difficulty in remembering and reusing concepts, as these functional capabilities are built in parallel with the development of more robust knowledge networks as learners grow more experienced.

To help facilitate this growth, Richhart, Chruch and Morrison suggest relying on routines and practices already intelligible to learners in order to scaffold the learning process and provide a more intuitive basis for the cultivation of new knowledge. For design learners, this routine might utilize the procedure conversation with materials and reflection in action (Schon) as means of gaining an understanding the materiality of new subject matters.

What does this mean?

1. Solutions should embrace rather than deny the lived experience, including difficulites, frustrations and fears, of design learners as a means of cultivating new knowledge.
2. Learners have distinct personal agendas and procedures which must be accounted for.
3. Don’t make learners ask the first question, they won’t be able to formulate it in a way that affords a satisfying answer.
4. Ground new knowledge in existing competencies rather than diving headfirst into completely alien domains.

Goals and Motivation

As Susan Ambrose notes, the subjective value of learning goals is critical to motivating new students to learn. Indeed motivation and goal setting are reflexively linked concerns in my research. In my review of literature, two models of of the nature of motivation gave particular insight into how to frame learning goals and maintain the motivation to learn.

Expectancy Theory

Expectancy Theory is an evaluative model for professional motivation emerging from Victor Vroom's research on practice and management in the 1960s. The theory posits three major components which, taken together, constitute an evaluative metric for gauging motivation. These are:

- **Expectancy:** The belief that a given effort will lead to a the achievement of a given performance level. (values range from 0 to 1).
- **Instrumentality:** The belief that a given performance level will lead to a certain work outcome. (values range from -1 to 1).*
- **Valence:** A coefficient of personal preference for certain results based on individual needs and goals ranging from complete dislike to welcoming. (values range from -1 to 1 with zero indicating apathy).

Vroom provides a useful formula a understanding motivation in terms of these three subjective metrics:

$$\text{Motivation} = \text{Expectancy} \times \text{Instrumentality} \times \text{Valence}$$

If this formula seems simplistic, it nevertheless provides a helpful shorthand for discussing motivation. At a high-level, it theory gives us several key insights about the relationship between motivation and uncertainty salient to my thesis:

- Highly positive motivation toward action strengthens to its potential maximum as certitude about effort, performance and outcomes increase.
- Conversely, an uncertain link between effort and performance or between performance and outcome will lead to amotivation or disinclination.
- While personal preference (valence) for an outcome is a factor in determining motivation, it can be negated entirely by uncertainty surrounding expectancy and instrumentality.
- And, even with perfect Expectancy and Instrumentality, a disinclination toward results negates motivation.

Self Determination Theory

While Vroom's theory is useful, it tells us little about modalities of motivation, and particularly the origins of a given valence. Subsequent work on motivational theory by Porter and Lawler (1968) posits a binary typology of motivation as either extrinsic or intrinsic. The former is characterized by tangible or immediately quantifiable rewards such as financial benefit, good grades, verbal acknowledgement, or the like. The latter type of motivation is characterized by spontaneous enjoyment of an activity for the activity's sake. Examples of this might be playing with a pet, solving a crossword puzzle, or having sex.

Many activities in the real world can be understood a mixture of both intrinsic and extrinsic motivations. A clear example of this is exercising. An individual may find the activity fun on its own, but external motivational forces such the desire to live a long life or to find a mate are also factors.

Gagné and Decipropose Self Determination Theory as a model which leverages Porter and Lawler's work to suggest a typology of extrinsic motivation ranging from the controlled to autonomous. Central to this model is the assertion that methods of regulation found in each mode of extrinsic motivation can be ranked qualitatively as they more closely map to autonomously felt need. The four modes of extrinsic motivation are described as follows:

- **Extrinsic Regulation:** characterized by external control and "contingencies of reward and punishment"
- **Introjected Regulation** characterized by moderate external control and "self-worth contingent on performance, [and] ego-involvement"
- **Identified Regulation**, characterized by moderately autonomy and the "importance of goals, values, and regulations."
- **Integrated Regulation,** characterisized bt autonomous motivation and "coherence among goals, values,and reuglations"

Given this rubric, we might, for example, describe the factors motivating someone to exercise as highly integrated, whereas the factors motivating someone to file tax returns are highly external.

Finally Gagné and Deci suggest factors that facilitate autonomous forms of motivation: "(1) Specific factors in the social context, such as choice and meaningful positive feedback... and (2) the interpersonal ambience, which can be thought of as being analogous to the organizational climate." Critically, these factors are essentially atmospheric and derive their potency from normative experience, belief and environment.

What does this mean?

1. Motivation is tied to clarity about the relation between actions, efforts, and outcomes. In order for learners (and particularly novice learners) to pursue goals, they this relationship must be made explicit from the outset.
2. Motivation is tied to degree of difficulty, for novice learners provide low difficulty learning challenges.
3. Goals should be tied to interest and need explicitly
4. Motivation to pursue a goal can be negated if that goal is not desirable to users.
5. Motivation and goals are bound up with social context, if goals mirror context, learners will find more autonomous motivation.

Process

October 2012: Why Do Designers Want to Learn to Code?

My thesis research began with an exploration of the attitudes and perceptions of designers learning, or attempting to learn how to code. My hope at the outset was to devise a tool to make this process--the acquisition and understanding of the formal syntax of code--more intelligible to design learners. In early interviews, I wanted to understand the rationales of design learners who wanted, or conversely, did not want to learn how to code.

- Many designers expressed their desire to learn in terms of positive, highly integrated goals and cited, for example:
- Code competency as a useful method of communication between designers and developers in industry.
- Code as a form of creative design play. As one student put it: “Look, I know I’m never going to be a developer, but I want to learn how to hack shit together.”
- Coding as a methodology of creating more robust, interactive prototypes for research and presentation.
- Code competency as affording an increased sense ownership over digital artifacts by affording a mental model for how they work.

One rationale however, was both widespread and extremely problematic;

Many design learners expressed their design to learn in terms of fear of professional irrelevance due to a lack of programming knowledge. Interviewees indicated that this fear is animated perceived market pressure and a desire to differentiate themselves with a hard skill in a competitive job market.

Conversely, those designers who answered in the negative provided three rationales:

- A prior learning experience which made them uncomfortable, or ended in perceived failure.
- Professional design goals don't involve coding as part of their desired practice.
- If they did learn to code, they would end up "being asked to do it" in professional practice.

The first affirmative rationale is striking. It suggests that designers feel an ill-defined *external pressure* to gain code competency, and experience this pressure on an epistemic level. As one interviewee put it rather bluntly: "part of me feels like design is bullshit if you're a really good programmer and you get the human then you don't even need a designer." This quote suggests that for some designers, the struggle to learn to code is part of a struggle to justify their own practice and existence. More to the point, it is a *horrible mental picture for students to take with them into code-learning contexts.*

Interestingly, all three negative responses were echoed by learners who responded in the affirmative. Members of both groups expressed experiencing frustration with prior attempts at learning to code, the difference being that members of the former group still viewed learning as a personal goal. More counterintuitively, both groups also expressed the fear that if they learned to code, they would be typecast in industry and have their opportunities inscribed or limited by their coding faculties. This concern suggests that writing code as a hard skill is thought of as somehow external to the design process. That it is, on a fundamental level, something other than a designer's tool.

Taken together with the fear of professional irrelevance, this perception suggests an odd paradox in designer attitudes about code. On the one hand, interviewees suggested that learning to code is part of a struggle to justify design's part in contemporary industry, and on the other, they evinced the belief that coding is different from design.

This is a rather sorry state of affairs for design learners seeking to learn to code:

- Because many have been previously frustrated, they have very low levels of expectancy about the process which decreases their motivation.
- Their motivation is decreased further due to uncertainty about the valence or desirability of programming for their personal goals.
- Yet they feel unease and an external pressure to learn.

October 2012: The Paradox of Code's Externality to Design

The perception of code's externality to design is further confirmed when considered relative to other, more traditional design tools such as Adobe Creative Suite products. While students I interviewed acknowledged the possibility that mastery of these tools might precipitate their use in rote, repetitive tasks (ie wireframing) in professional design practice, they nevertheless expressed a willingness to become more competent users of these systems. They perceived these more traditional competencies as critical for the development and communication of design concepts and solutions.

While learners were might want to learn code for professional reasons, interviews highlighted two key features of code that made them hesitant to engage.

First, traditional tools afford a means of directly modeling, expressing, and manipulating well understood design concepts such as color, form, typography, narrative, etc. Code, on the other hand, lacks this immediacy. Writing code does not afford direct manipulation, and only allows user to model design concepts through the production of a set formal grammatical structures.

Second, while design tools like Creative Suite are bespoke software environments tailor-made for designers, code does not come from design, and its traditional primary uses are anything but designerly. If this somewhat obvious, it nevertheless informs the attitudes of design learners attitudes about code's externality to design practice.

November 2012: How Is Code used in Professional Practice?

In order to better understand how professional designers make use of code and whether learners concerns expectations to reality, I conducted several interviews with designers working at tech companies, design consultancies, startups, and branding agencies. What I found was quite a broad variety of paradigms across professional practice.

What I found is basically this: some designers code, some designers code in professional practice, while other designers have nothing whatsoever to do with code. More importantly, my interviews suggested that coding competency is not, in general, a baseline skill for employment consideration. Rather, the Moreover, in large technology companies and branding agencies in many cases actively segregate design and software development entirely.

- While I found that there is is no single model for how it's done in industry, the strategic use of code for design prototyping is growing more prevalent. The reasons for this are several:
- The proliferation of devices, screen sizes, and service delivery vectors available to end users makes programming a facile solution for rapid prototyping relative to more traditional static mockups.
- The proliferation of relatively high-level interaction-focused libraries such as jQuery, D3, and HTML5 canvas APIs as well as inexpensive physical prototyping systems such as the Arduino and Raspberry PI platforms.
- Within consultancies, a shifting set client expectations about the nature and form of deliverables.
- Within startup culture, a general requirement for an 'all hands on deck' approach to rapid, iterative software development.
- Within software/service firms, a desire to reduce "'cycle time" between designers and developers using coded prototypes as a mutually intelligible method of transcending disciplinary silos.

This list however, is somewhat misleading, because for most professional designers, code competency is primarily seen as a means of *understanding how to design with technology* and not an end itself. Interestingly, non-coding designers made notion of the importance understanding how to design with technology well. For them, this understanding hinged on a building vocabulary for conversational competence and the ability to interpret and understand the nature and structure of computational systems in order to design with a robust accounting of the constraints, behaviors, and possibilities available.

What does this mean?

1. Learning to code, in itself, is not a guarantor of employment in the design industry.
2. The fear that designers may feel about the lack of code capability may be overblown in general.
3. The code that designers do use is targeted toward prototyping.
4. Designers in industry who work with code describe the rationale for the practice in highly integrated terms
5. More important for many designers may be a conversational competency about the nature and structure of computational systems.

November 2012: Why Is Learning To Code Difficult For Designers?

In interviews with design learners, I sought to assess the precise nature of the frustrations felt by designers learning to code. Many users made direct reference to the alienness of code described in the section above as a source of consternation. (code as a formal linguistic structure and code as coming from outside of design). However, while these are factors may underlie learner difficulties, students gave a range of insights into their own frustrations at coming to terms with code. These accountings fall into three primary categories:

Frustration with the lack of immediacy in programming tools: Many interview subjects expressed their frustration as simple exasperation with the formal syntax of programming. As one designer summed it up: "I could never figure my way out of the missed punctuation. Once I messed something up that was it and I couldn't figure out how to go back and retrace my steps." Interestingly, the formality of programmatic abstraction, rather than a difficulty understanding the logic of programming created the biggest stumbling block. Users often followed up such statements of exasperation with the caveat that they "get the logic [behind programming], but don't have the muscle memory" to implement it successfully.

A sense of lostness or disorientation: Learners consistently cited an inability to orient themselves as a locus of frustration. Often, this failure was articulated in geographic terms, as when one interviewee described a feeling of not "knowing where to start or where I'm going." This lostness was often exacerbated by a sense that methods of instruction do not clearly articulate context, and that they "skip explaining the broad sense and dive into these little facets."

A Priori fatalism about failure: This point often revealed itself as as an emergent theme stemming from one or both of the former points. At issue is the concern that code is an extremely alien form of expression for designers, and that because of this . As one learner put it: " It's this fear of not being smart enough to figure it out and to bump up against your brain's limitations. I'll try to access that part of my brain and it comes back 'file not formatted.' Like, it's going to be me feeling so frustrated, crying in front of some tutor, not getting it." Not coincidentally, learners who evinced such fatalism were often the same learners who reported frustration when teachers or more advanced students told them that the simple introductory concepts causing them such grief were easy.

Importantly, the second and third items are not really about coding at all. Item two is really about the much more general problem concerning how novice learners orient themselves when encountering an alien vocabulary while lacking a mental map of the topography she is beginning to explore. Item two, on the other hand is about a fear of failure. While this fear may be exacerbated by items frustrationa nd lostness, it is imported into the situation at hand from without.

November-December 2012: What Does Code-Learning Look Like and What are Its Limits?

While probing designers about their frustrations, I wanted to understand what role particular learning resources played in the learning process. Were some tools better than others at mitigating or eliminating learner frustrations? Did some resources exacerbate learner frustrations?

There are any number of tools and systems available to help designers come to grips with code, from the traditional resources such as books and classroom instruction to the extremely modern websites like Codecademy, Lynda, and StackOverflow. Unsurprisingly, interviewees reported using all of them in their attempts to learn how to code. Some students felt more comfortable with online tutorials while others needed more tailored attention from teachers, but I was unable to identify any clear patterns about the expediency of resource types.

A survey I conducted of students in Computing for the Arts with Processing, an introductory course at Carnegie Mellon targeting artists and designers, reveals this diversity nicely. Students int the class had direct access to six primary learning resources:

- The professor's in-class code examples
- The professor's notes on in class examples handed out as PDFs
- The online API for processing, essentially a reference for all of the functions, and constructs in the language
- A textbook
- Teaching assistant office hours
- Open Processing, an online community-built, showcase of processing code examples.

I asked students in the class to specify how helpful each of these resources were, and their responses seemed confoundingly random. Some students loved the API and hated the book. Other students loved the professor and TA (me!), but did not find the API at all helpful. Still others relied exclusively on Open Processing and its massive archive of user-submitted examples. Further adding to my confusion was the fact that resource preference did not correlate to student performance within the class.

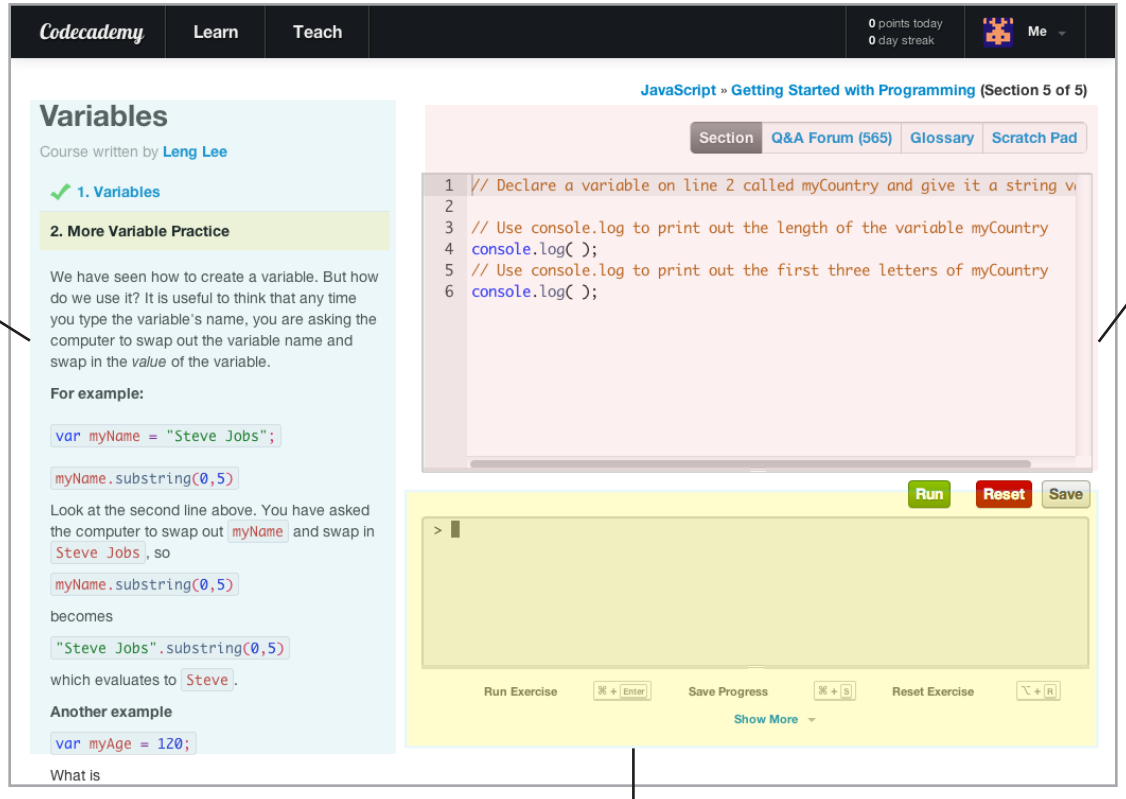
What these results tell us is that *the best tool for the learning to code is the one that works for its user*. If this seems slightly tautological, it nevertheless has meaningful implications for my thesis research. However, it was not until I began to look at one learning tool in particular that I began to understand the significance of this formulation.

In interviews with designers, one online training course stood out for its perceived efficacy. *Codeacademy* is an open web-based code teaching system dedicated to the lofty goal of: "teaching the world to code." For the time being, I will remain mute on the merits of this goal, and simply address the reason it works.

In order to get a better handle on why learners liked Codecademy, I began experimenting with the system. At its essence, Codeacademy is simply a series of online coding tutorials. Its particular genius emerges from its organization. Users select a programming track and complete a series of very small, discreet coding tasks that gradually ramp up in difficulty and complexity as the course progresses. Figure X. shows the layout of one such tutorial page as consisting of three component pieces:

An explanation or overview of the concept being introduced.

A code editor for rehearsing the concept introduced.



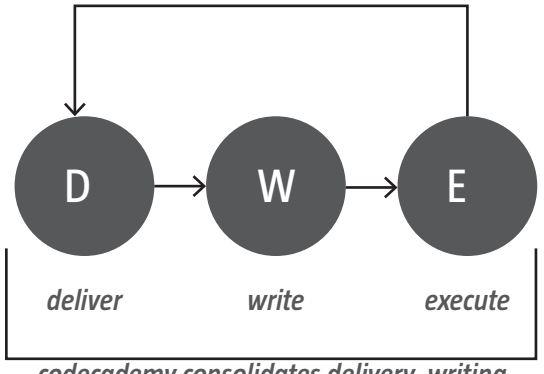
A console for displaying the results.

the layout of tutorial pages on Codeacademy in three parts

1. A code editor for rehearsing the concept introduced.
2. A console for displaying the results.

This system presents a distillation of the code-learning process into an extremely transparent, quantized, and immediate form. Otherwise stated, the three components of the Codeacademy window are essentially a consolidation and abstraction of the three parts inherent in all code-learning systems. These are:

1. Some vector for the presenting concepts/syntax/information. (Delivery)
2. Some code editor for learners to write and experiment with this information. (Write)
3. Some execution context for confirming the results of steps 1 & 2 (Execute)



codecademy consolidates delivery, writing, and code execution into a context

Why do design learners respond to Codeacademy? It consolidates, simplifies, increases the number of cycles learners go through this delivery-write-execute learning loop (Dwell). In so doing greatly enhances the immediacy of the process. By reducing latency between these phases of code-learning, Codeacademy affords a reasonable approximation of the direct-manipulation paradigm of traditional design tools. What emerges from this low-latency, high-speed loop is the mitigation of the frustration with the traditional segregation between delivey and writing of code.

In more general terms, tells us is that different learning resource is simply this. Given the option, learners will make use of whatever delivery vector best helps them manage their frustration with the lack of immediacy of code. If the survey results from the students learning Processing seemed random, it is because each found the delivery vectors best suited to her purposes for reducing frustration in the Dwell.

Pivot: The Limits of Dhwell In Addressing the Needs, Goals and Frustrations of Designers

Once I understood the connective tissue of the Dwell model underpinning code-learning systems, I immediately saw the following gap:

While quality design can attend to the frustration created by code's lack of immediacy, the problems of lostness and a priori fatalism originate before this loop is entered and are therefore unlikely to be resolved within it.

Consider the example provided in the forward regarding students learning the wrong programming language for their needs. Even if I taught masterfully, and liaised my students through the Dwell with the utmost care, they would remain unable to achieve their goals. Alternately, if a learner has an a priori sense of fatalism about coding in general, it is unlikely that she will seriously engage in the Dwell to begin with.

January-February 2013 Interviews about Lostness and Fatalism

At this point in my research I understood that no magic bullet code-learning tool could solve the motivational problems that designers faced, nor could it account for the problems of lostness and fatalism described by design learners. In order to reorient myself I conducted a series of follow up interviews with design learners who I knew had uneasy feelings about learning code to drill down on the nature of these issues.

In this sent of interviews, I changed the way I formulated my questions. Instead of asking design learners if they wanted to learn to code, I asked what they wanted to know about code. This reformulation proved disarming. It lifted the assumption that there was a quantitatively correct answer, and facilitated a much richer set of interviews. In these interviews, three critical new concepts surfaced:

1. Underneath the externalized rationales for learning to code, I found that many of these students simply wanted better understand the ambient context in which their designs lived, and saw learning to code as the only apparent means of learning about this context.
2. Interviewees expressed a desire to simply understand what things mean and how they fit together. This presented it self both on a micro-level as an interest in understanding coding concepts in plain english, and also on a macro-level understanding exotic terms like PHP and AJAX. One interviewee, for example, made reference to a recent conversation in which she felt alienated when the conversation turned to a discussion of an exotic programming language.
3. In these interviews, talking to developers came up again and again as a rationale for wanting to learn to code. Design learners expressed having experienced frustration with not understanding the concerns of developers, and believed that learning to code would be useful in opening a communication path.

What does this mean?

1. Why are designers so obsessed and paranoid about learning to code when developers do not think of this faculty as advantageous to more fluid transmission of understandings?
2. Understand basic implementation patterns might help designers categorize code concepts and move past the exotic terminology and detail.

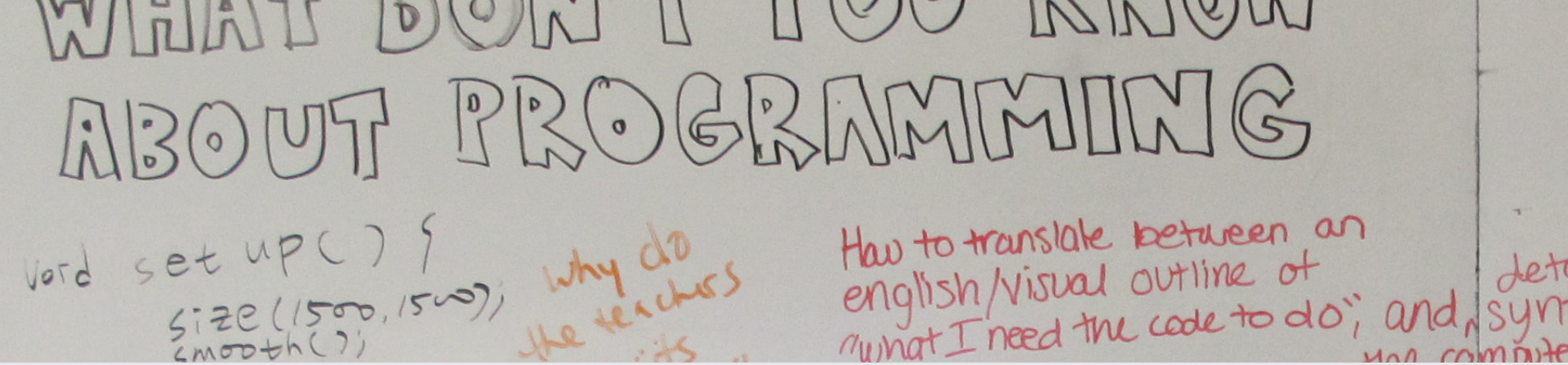
January-February 2013 Guerrilla Research

In order to round out this new formulation of what designers wanted to learn about code, I used whiteboards and posters placed throughout the CMU School of Design with the the following prompt: “what don’t you know about programming?” This was meant as a provocation let designers self-assess their own limits and curiosities.

As a supplement to these provocations, I handed out research packets to help drill down on the perceived differences between designers and developers. These packets focused on how designers viewed distinct though patterns between the two groups.

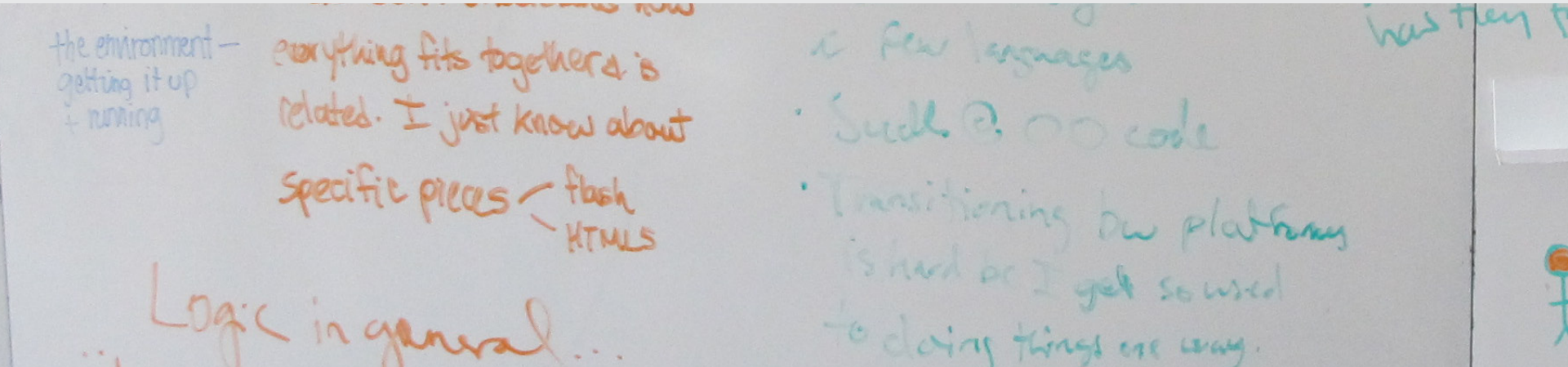
What I found, on the one hand, was that many of designers want to understand macro-level concepts more clearly. While some designers pointed to implementation, a great number of responses indicated a desire to understand things like “how everything relates,” “what language does what” and “which tool is useful for which purpose?” In other words, responses suggesting a desire to better orient to the space.

With regard to the relationship between designers and developers, participants largely framed the divide in terms of binary oppositions such as “questions problem vs. solves problem”, “analytical vs creative”, “front-end vs backend,” and “making vs. thinking.” Where designers saw commonality was in in terms of building products and solving problems. As one participant put it: “We want to make good, useful shit that is easy to use and will change lives for the better.”



What does this mean?

1. Make a tool to focus on orienting designers.
2. Frame the answers to their questions in their language; use plain english and rely on the fact that designers model systems holistically.
3. If designers view themselves as creative, empathetic, and try to "think of themselves as the user", how can this be used as a learning/conversations strategy to establish working relationships with developers about mutual product goals?



Feburary 2013: Interviews with developers

As I began to hear designer concerns about the ability to communicate with and understand developers I decided I ought to go ask a few about their relationship to design and designers. What worked, what didn't, and what would they like designers to understand about their practice.

While some developers I spoke with mirrored the frustrations of designers in communicating concerns relevant to their practice across the divide, others suggested possible pathways forward:

- First of all, developers don't spend time learning the syntax and grammar of code; rather, they they spend their time solving problems with code. This is an important point, because it suggests a reframing from the idea of how to learn to code, to a more active, integrated frame of what do you want to do with code.
- Developers don't care if designers know how to code or not. What they want from designers are robust, complete system models focused on the product or problem at hand.
- The act of development requires a highly cultivated sense of selective ignorance. In practical terms, developers often have to narrow their focus to some particular component and understanding what messages it sends and receives.
- At the same time that developers said that they largely think in terms of system flows and system models and they break down complex systems into a series of
- There are really only a few basic implementation patterns that inform the way developers think about systems (eg Client, Server, Database).

What does this mean?

1. Fatalism is based on the notion that 'I'm too stupid to do this', in fact, it's really based on the ambivalence about perceived motivation.
2. Lostness is being adrift in a sea of words and concepts that I can't name or communicate with.
3. When asked directly, designers want to be able to communicate about code in terms of the systems and structures that use it.
4. Designers who express a desire to learn to code may simply be interested in a richer understanding the nature of the digital systems they design for and with.
5. Designers want to be able to communicate with and understand developers in particular.
6. Designers may simply want to be able to talk about code intelligently, not necessarily talk with code.

Atlas

April 2013: Final Prototype

Atlas is a tool for helping designers gain computational literacy rather than a tool for learning to code. At its heart, it is based on two interrelated principles revealed in my research:

- Designers may or may not want to learn to code. Rather, they want to understand the nature of computational systems to make more informed, articulate decisions about their choices regarding them.
- Designers, like all new learners need help contextualizing and integrating new knowledge, especially as it relates to a topic as broad as coding. My interviews with designers and developers revealed that a focus and simple, communicable computing system concepts greatly reduces confusion about the overwhelming and exotic vocabulary in this spaces.

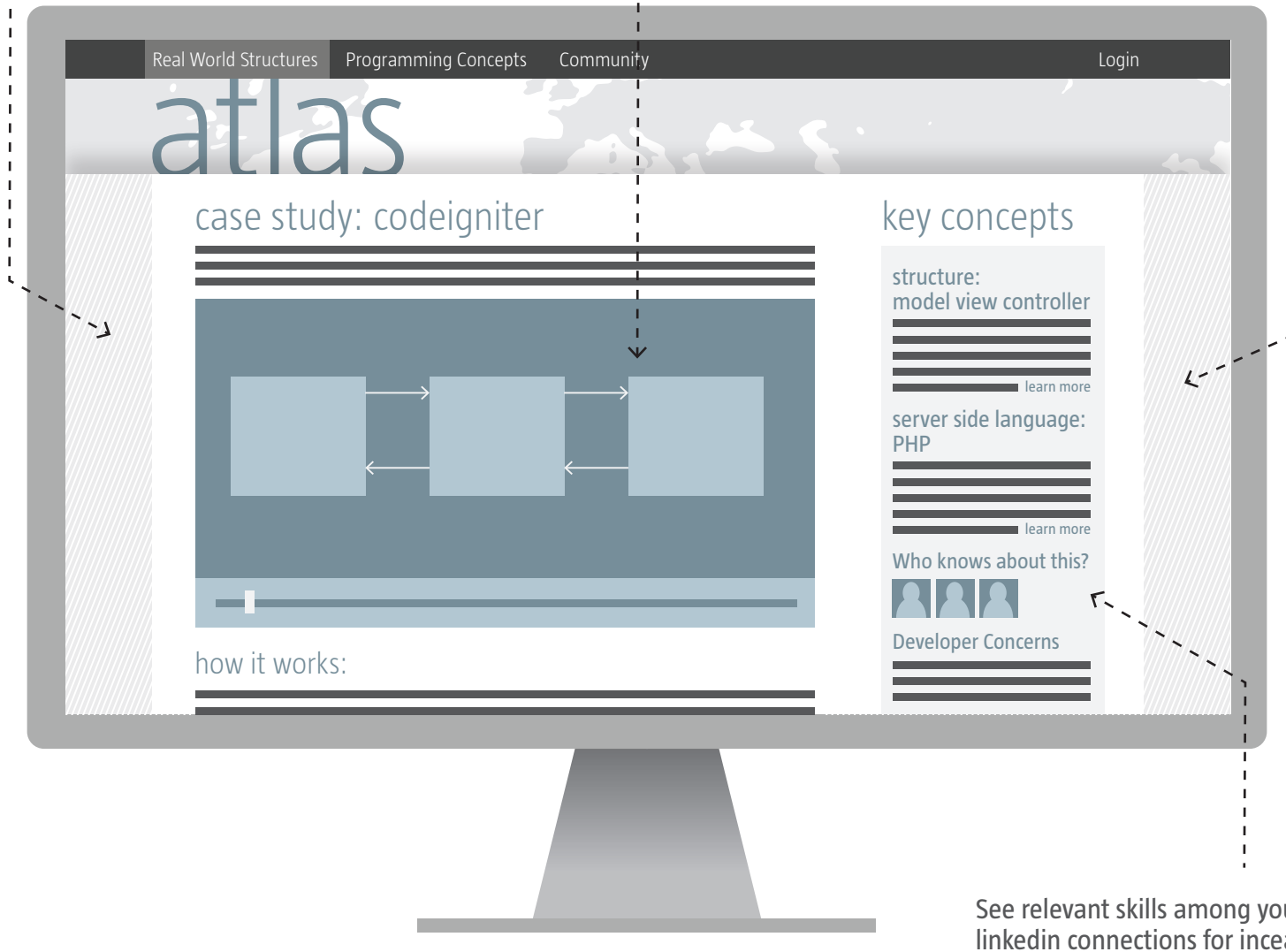
Atlas is an interactive web-based guidebook to help designers meet goals and challenges relating to code and computation. Atlas relies on traditional design values and plain-english communication to consolidate, categorize, and bring computational concepts to designers. While it is designed to expressly to accommodate the needs of design learners, but may also have significant value as a general resource for other non-technical learners The current system provides the following:

- Provides high-level overviews of common programming concepts.
- Makes use of real world examples and metaphors wherever possible to ground macro-level concepts.
- Help contextualizing and integrating learners' current computation knowledge.
- provides a socially connected experience to reenforce environmental and community motivation
- Explicitly communicates problem-solving over syntax as a the common concern of professional designers and developers.

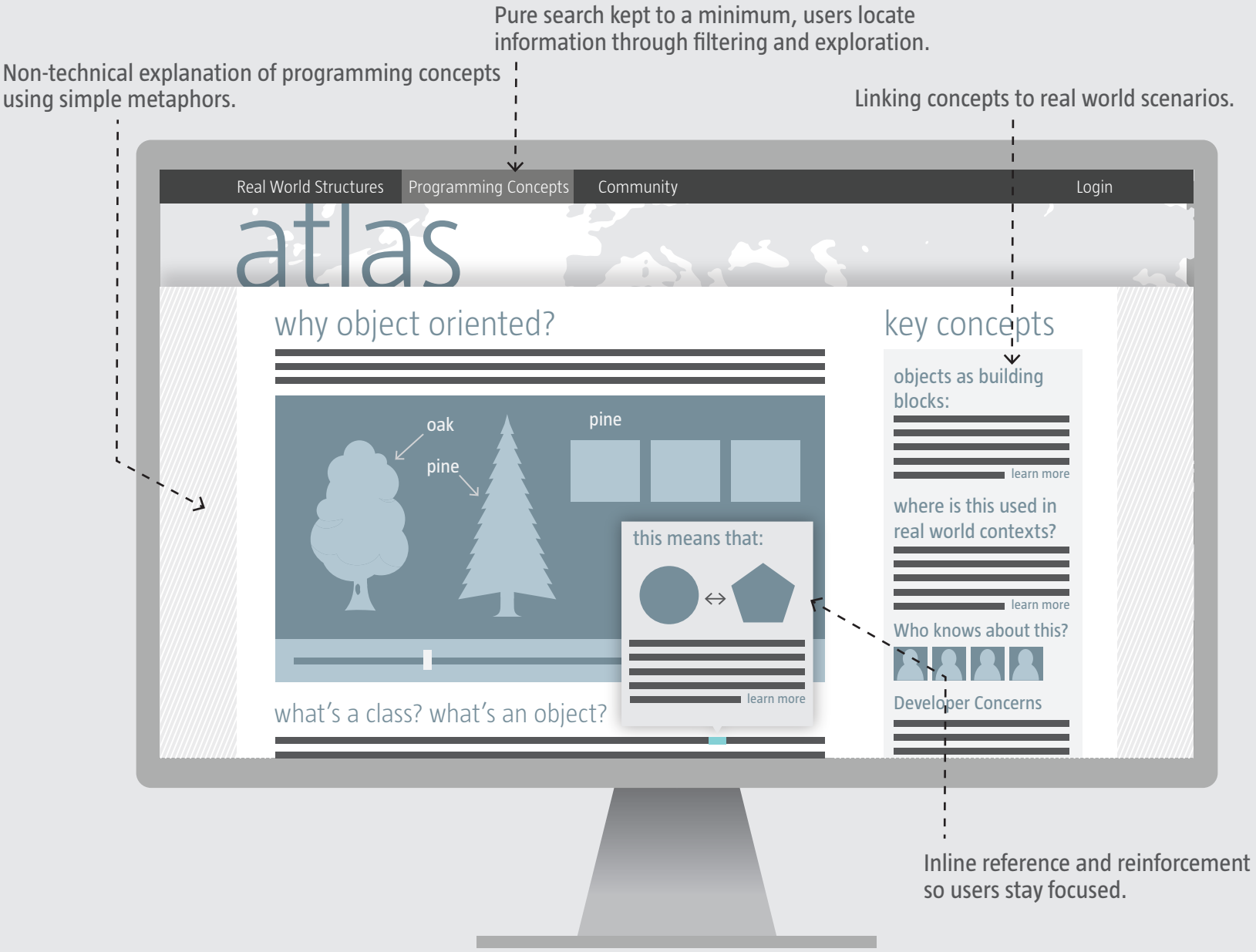
Emphasis on modeling general structures/patterns through real world systems.

Use of video to visualize how systems function

Links structures to specific sets of language tools to provide context for exotic terminology.



See relevant skills among your linkedin connections for inceased social connectivity


Pure search kept to a minimum, users locate information through filtering and exploration.
Non-technical explanation of programming concepts using simple metaphors.
Linking concepts to real world scenarios.
Real World Structures
Programming Concepts
Community
Login
atlas
why object oriented?
oak
pine
pine
this means that:
learn more
what's a class? what's an object?
key concepts
objects as building blocks:
learn more
where is this used in real world contexts?
learn more
Who knows about this?
Developer Concerns
Inline reference and reinforcement so users stay focused.

Community curated resources to help filter the noise of search

Users can rate resources and drive the conversation around what's out there.

Submission requires that users provide a rich context for resources.

Conclusions + Extensions

In the first half of the year, I spent a great deal of time thinking about how to help designers learn to code. Only later did I realize that this goal was myopic and ignored the greater set of designer needs. It is not surprising that for many of the design learners I spoke with, the desire for an articulate, system-level conversational intelligence, collapsed and disappeared fears and worries about learning to code. The problem is that this kind of proscriptive talk creates nervousness, uncertainty and is hopelessly vague.

Fundamentally, code is nothing more than a tool for accomplishing some set of tasks. The problem I've identified in my thesis (and a trap I fell into), is that it's not the only tool for accomplishing tasks. For designers interesting in understanding computation as it relates to them, starting with code is particularly problematic: it is extremely granular in nature, it lacks affordance and reference to known design concepts, and code in and of itself, tells the user nothing about its broader operating context.

Because I spent a great deal of time framing my problem as a problem of code-learning, I came into a big picture understanding of the needs, goals, and fears of designers late in the thesis process. Over the course of the year, I went from a default, and frankly glib set of assumptions that everyone should learn to code, to a more robust understanding how to meet learner goals in the space of computation. Nevertheless, because my road to damascus moment came somewhat late in the process, I feel my prototype still needs refinement and testing before I'm willing to stamp it as 'done.'

Finally, I would really like to implement a system like Atlas in the long term to help designers and other non-technical develop a sensibility about coding technologies. Ironically enough, in order to build such a site, *I will need to use a lot of code.*