Interfacing Iconicity - Addressing Software Divarication Through Diagrammatic Design Principles

George Akhvlediani

Follow this and additional works at: https://scholarcommons.sc.edu/etd

Part of the Computer Sciences Commons

Recommended Citation


This Open Access Thesis is brought to you by Scholar Commons. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Scholar Commons. For more information, please contact dillarda@mailbox.sc.edu.
INTERFACING ICONICITY - ADDRESSING SOFTWARE DIVARICATION THROUGH DIAGRAMMATIC DESIGN PRINCIPLES

by

George Akhvlediani

Bachelor of Science
University of South Carolina, 2016

Bachelor of Arts
University of South Carolina, 2016

Submitted in Partial Fulfillment of the Requirements
For the Degree of Master of Science in
Computer Science
College of Engineering and Computing
University of South Carolina
2018

Accepted by:
Duncan A. Buell, Director of Thesis
Heidi Rae Cooley, Reader
Gregory Gay, Reader
Cheryl L. Addy, Vice Provost and Dean of the Graduate School
ACKNOWLEDGEMENTS

I owe gratitude to the friends and family who so generously offered their encouragement. Among them, I specifically recognize my mother and father, who eagerly and tirelessly support me in all my endeavors. To them I owe most of who I am. The rest certainly deserve acknowledgement, but I abstain from specifics out of respect for their privacy. I ask that they accept my gratitude on these terms.

Among my faculty I must identify both Dr. Duncan A. Buell and Dr. Heidi Rae Cooley, who have invested no small amount of effort in guiding my academic pursuits. I met them in my first undergraduate semester, in separate circumstances. Over the years, they have extended to me an immensely rewarding mentorship. I have no doubt that without them, this would not have come to fruition. I cannot neglect to mention Dr. Gregory Gay, who I came to know only recently, for his integral role as a committee member. To all of them I offer my sincere gratitude.
ABSTRACT

This research examines conflicts accompanying the proliferation of computer technology and, more specifically, constellations of dependency in the always expanding volume of software, platforms, and the firms/individuals using them. We identify a pervasive phenomenon of “divarication” in the growing variety of progressively specialized systems and system roles. As software systems enter new thresholds of sophistication, they effectively aggregate many distinct components and protocols. Consequently, we are confronted with a diverse ecology of stratified and thereby incompatible software systems. Software inherits the limitations and potential flaws of its constituent parts, but unlike physical machinery, it isn’t readily disassembled in instances of failure. The individuals using these systems have no means to dissect and analyze their tools, and thus are necessarily dependent on external assistance.

We assert that divarication is a consequence of interfacing, and particularly in the way computer interfaces operate as the sole point of contact between a user and a software system. These interfaces condense tremendous volumes of computation into outputs that can be read and understood by almost anyone. This is self-evident in the total ubiquity of computing devices across populations and throughout sectors of commerce. However, and unfortunately, merely “using” software doesn’t promise any depth of understanding beyond its most superficial aspects. We argue that this circumstance makes
divarication inevitable. Opaque components accumulate into opaque wholes, and so the magnitude of this problem will likely scale with increasing software sophistication.

Taking Charles S. Peirce’s three types of sign (the icon, index, and symbol) into special consideration, we observe that term “icon” distinguishes a sign that directly resembles its referent. As the thesis title indicates, we bring Peirce’s notion of “iconicity” into accompaniment with “interfacing”, forming an abstract paradigm in response to divarication. We intend to develop a platform that facilitates at least partial disassembly and examination of program operations by incorporating a recurrent “iconicity” protocol. We composed a diagrammatic design scheme; a blueprint for software platforms that might emulate this “interfacing iconicity”. We then developed a prototype platform, implementing this structural logic. This initial prototype is a rudimentary HTML rendering platform, one that articulates the relationship between plain-text code, its Document Object Model (DOM) representation, and the rendered “web page” (as it would appear in a browser). Currently, this prototype is a useful analog for the arguments we present. Since it articulates the relationship between text markup (code) and its interpretation, it may also have educational utility. However, this prototype is not yet a fully realized implementation of our design paradigm, and at this stage a conclusion on whether the latter genuinely addresses divarication would be premature.
# TABLE OF CONTENTS

Acknowledgements.............................................................................................................................. iii

Abstract................................................................................................................................................ iv

List of Figures ......................................................................................................................................... viii

List of Abbreviations ............................................................................................................................ x

Chapter 1: Introduction .......................................................................................................................... 1
    1.1 Project Proposal ........................................................................................................................... 4
    1.2 Chapter Outline ............................................................................................................................ 6

Chapter 2: Analysis .............................................................................................................................. 8
    2.1 Interface ....................................................................................................................................... 8
    2.2 Iconicity ....................................................................................................................................... 13
    2.3 Interfacing Iconicity .................................................................................................................... 18
    2.4 Diagrams ..................................................................................................................................... 20

Chapter 3: The Form-Process Diagram ............................................................................................... 25
    3.1 Verb-Noun-Adjective: The Governing Logic .............................................................................. 27
    3.2 Performative-Enunciative-Determinative - Interior Circuit ......................................................... 28
    3.3 Execute-Compose-Analyze - Exterior Circuit .............................................................................. 34
    3.4 Render-Inscription-Mode - Intermediary Circuit ......................................................................... 36

Chapter 4: Identifying & Addressing Issues of Divarication ............................................................... 41
    4.1 Visual Programming – MIT App Inventor ..................................................................................... 43
    4.2 Professional Development Tools – Adobe Dreamweaver ............................................................ 48
### 4.3 Programming Feedback – Light Table

Chapter 5: Prototype Description

5.1 Intermediary Circuit Analysis

5.2 Exterior Circuit Analysis

5.3 Interior Circuit Analysis

Chapter 6: Prototype Specification

6.1 Libraries used in development

6.2 Graphics engine

6.3 Data processing

6.4 User input/interaction

Chapter 7: Results, Analysis and Consequences

7.1 Development status, short and long-term

7.2 Possible applications

7.3 Project continuation and final remarks

References
LIST OF FIGURES

Figure 2.1 Cube (ink) ........................................................................................................... 9
Figure 2.2 Cube (digital) ..................................................................................................... 10
Figure 2.3 Interior Architecture .......................................................................................... 11
Figure 2.4 Definition “f” of icon in OED ........................................................................ 14
Figure 2.5 A 7-tuple formal specification of a TM .............................................................. 17
Figure 2.6 Transition table corresponding to the 7-tuple TM .............................................. 17
Figure 2.7 Hopcroft & Ullman transition state diagram of a Turing machine ................. 21
Figure 2.8 The Panopticon .............................................................................................. 22
Figure 3.1 Form-Process Diagram .................................................................................... 25
Figure 3.2 Verb -Noun – Adjective triad .......................................................................... 27
Figure 3.3 Interior circuit of form-process diagram ............................................................ 28
Figure 3.4 Exterior circuit of from-process diagram ........................................................... 34
Figure 3.5 Intermediary circuit of form-process diagram .................................................. 37
Figure 4.1 MIT App Inventor “Designer” interface .......................................................... 45
Figure 4.2 MIT App Inventor “Blocks” interface ............................................................... 46
Figure 5.1 Prototype screenshot .......................................................................................... 52
Figure 5.2 Selection in plain text view .............................................................................. 55
Figure 5.3 Selection in DOM tree view .............................................................................. 56
Figure 5.4 Selection in Browser view .................................................................................. 57
Figure 5.5 Google Chrome inspection, elements pane ....................................................... 59
Figure 5.6 Google Chrome inspection, console pane ..............................................61
Figure 5.7 Rendering engine basic flow .................................................................63
Figure 5.8 The HTML document’s determinative effect on the resulting web page ......66
Figure 5.9 Condensed circuit determinative of web page rendering in detail ............67
Figure 6.1 An instance of multiple element selection.............................................78
LIST OF ABBREVIATIONS

CSS ................................................................. Cascading Style sheet
DOM ................................................................. Document Object Model
GUI ................................................................. Graphical User Interface
HTML ............................................................. Hypertext Markup Language
IDE ................................................................. Integrated Development Environment
OS ................................................................. Operating System
UI ................................................................. User Interface
UTF ............................................................... Unicode Transformation Format
XML .............................................................. Extensible Markup Language
CHAPTER 1
INTRODUCTION

In *Cybernetics: or, Control and Communication in the Animal and the Machine* (Wiener 1948, 1961) [1] Norbert Wiener identifies that the partitioned status of emerging scientific fields motivated the formation of “cybernetics” as a distinguished realm of inquiry. He indicates that his colleagues shared common concerns, in that the forefront of scientific progress followed a general pattern of distribution into narrowing territories, becoming confined to the exclusive purview of specialists (Wiener 1948, 1961, p.2) [1]. They observed a tendency for fields of study to fracture as they expanded beyond their initial scope, as topics diverged into subtopics. Wiener articulates the unfortunate consequences of this circumstance as a duality of redundancy and omission (Wiener 1948, 1961, p.2) [1]. Important work was often replicated across disparate fields, and the peripheral regions between these intellectual enclosures that promised the “richest opportunities” were inaccessible. We denote this as a phenomenon of divarication, in order to emphasize increases in both the granularity of and the neglected volume (of opportunities) between these subdivisions.

Consequently, the designation “cybernetics” sought to draw disparate fields of scientific inquiry into collaboration, to form a basis of inter-compatibility between control engineering, physiology, and statistical mechanics (Wiener 1948, 1961, p.11) [1]. As a field of study, cybernetics holistically approached problems of communication,
shedding the idiomatic constraints of distinct disciplines that had each sought answers therein (Wiener 1948, 1961, p.8,11) [1].

The inception of this thesis observes a similar trajectory, and its territory of inquiry is, in part, a direct derivation of cybernetics. Its domain encompasses computation, especially with regards to current deployments that appear to intersect almost every aspect of human activity. This document calls attention to the multiplicity of computing platforms, languages, libraries, and software available in an imminently expanding ecology of discrete use cases. In this context, the fundamental definition of use case is most appropriate, referring to any particular mode of engagement between an actor and a system.

The phenomenon of divarication is observably abundant in the recent and explosive growth of computing use cases. The variety of devices, operating systems, programming languages, drivers, applications, networking protocols, and subsectors therein produces a disparate and rigorously stratified volume of incongruous use cases. In keeping with noted concerns addressed in *Cybernetics*, we observe “an inextricable tangle of exploration, nomenclature, and laws” (Wiener 1948, 1961, p.2) [1].

“Computer use” hosts its own growing pool of sub-specialties; job market solvency has become contingent on an ability to “use” specific hardware, operating environments, software, and so forth. The characteristic effects of divarication become apparent in the moment that engagement with a particular sub-system utterly fails to produce any knowledge beyond the scope of that system’s immediate periphery. In other words, knowing how to “enact” a particular use case does not necessarily imply aptitude
elsewhere, or even within the same general operating environment. Large-scale implementations of computing systems require “Information Technology” (or IT) infrastructure, staffed by mediators who act as translators between persons and machines. Consequently, a significant portion of computer users have a limited understanding of the systems they engage with.

Understandably, the most widely used software systems have the tendency to conceal their internal mechanisms beneath a relatively minimalist “User Interface” (or UI). This becomes especially troublesome when a software fault occurs, as an error message offers only a glimpse of insight (if at all) into an otherwise invisible internal system. When the UI no longer functions as it should, the “user” is left with little recourse, other than to request assistance or to search for a specific solution. Difficulties may emerge in diagnosing an error correctly, determining who to ask (or where to search) for assistance, and what the “user” should do when an apparatus instrumental to their role doesn’t function as intended. The constraints typically imposed upon this peripheral, quarantined role of the “user” preclude internal rectification, and thereby demand external interference. We will consider the possibility of an alternative arrangement, one that offers opportunities for exploration and experimentation (within reason) in a system. With more flexible platform boundaries, it might be possible to avoid the entrapment of specialization. Our inquiry seeks to confirm whether this is viable and feasible, or not. If our solution is either impractical or ineffective, we will seek an explanation in its stead.
1.1 Project Proposal

The comprehensive argument presented by this thesis is to establish a software platform schematic, tuned to address this phenomenon of divarication. This inquiry involves the following:

- A theoretical assessment that identifies the mechanics of divarication and proposes a potential solution.
- An articulation of the proposed solution in a form applicable to software engineering.
- The development of an experimental prototype to evaluate the feasibility, applicability, and ultimate limitations of such a software platform.

In a computer science paper, an immediate invocation of the theoretical may warrant hesitation. However, this a necessary digression. The rapid pace of hardware development and of implementations therein leaves little room for static, instance/implementations-specific treatises. Alfred Aho and Jeffrey Ullman’s first joint publication on the topic of compiler theory, *The Theory of Parsing, Translation, and Compiling: Volume 1* (Aho & Ullman 1972) [2], opens its preface on such an assertion:

> In an area as rapidly changing as Computer Science, sound pedagogy demands that courses emphasize ideas, rather than implementation details. It is our hope that the algorithms and concepts presented in this book will survive the next generation of computers and programming languages, and that at least some of
them will be applicable to fields other than compiler writing.

(Aho & Ullman 1972, p.ix)[2]

This relatively abstract approach is ratified by the volume of citations referencing this text and subsequent publications by Aho and Ullman. The “green dragon book” (named for its cover illustration), *Principles of Compiler Design* (Aho & Ullman 1977), is perhaps their most popular publication concerning this topic. The foundation they provided in the 70’s remains pertinent, as subsequent editions and updates have not addressed the fundamentals of compiler theory so much as they have described implementation in more rigorous detail. For further emphasis, note the similarities between this quotation’s second sentence and the previously cited *Cybernetics* rhetoric.

The specific case of divarication, as per our definition, manifests endemically. This is a consequence of design, as computers can process vast binary sequences at a speed and scale that grossly exceeds human capabilities. Digital computers are valued precisely for this inherent capacity; they conceal tremendous volumes of rapid computation within compact forms. Software systems built upon this mechanical foundation inherit a predisposition for compression, their interior mechanics are by default invisible (or rather, impossible for humans to perceive). Divarication is inevitable in these circumstances without deliberate intervention, given that humans cannot readily disassemble software in the fashion of physical machinery.

Since divarication is an inherent tendency of computer systems, a system-specific implementation cannot completely address it. Instead, we attempt to determine how divarication proliferates in order to design countermeasures (depicted through abstract
schematics) that could proliferate alongside it. The proposed “platform” would serve to demonstrate these software engineering principles in action.

1.2 Chapter Outline

In the course of this project, we attempt to isolate and address a common factor (specifically, divarication) to achieve a more general solution, one that is not instance-specific. We seek a solution that can be applied methodologically, one that achieves desirable outcomes in many different scenarios. This document is divided into chapters for each distinct stage of research.

Chapter 2 encompasses the theoretical foundations of this enterprise, with subsections corresponding to each major concept. The first, “Interface”, asserts that divarication is a product of the convolutional sequences of interfacing that make contemporary computer use possible. The next section, entitled “Iconicity”, examines the qualities particular to Charles Sanders Peirce's theorization of the “icon”. The icon is one sign of a triad, distinguished as possessing an explicit and unambiguous resemblance to its referent. In the section entitled “Interfacing Iconicity”, these notions of interfacing and iconicity are drawn into tension and with each other, establishing the abstract paradigm that is “interfacing iconicity”. In the final section, “Diagram”, we present the Peircean “diagram” as the icon permutation that best suited for visuo-spatial demonstration, and thereby for conveying otherwise invisible interior structures and logic.

In Chapter 3, “A Diagrammatic Model”, we compose the diagram that will instruct our development efforts, for software platforms emulating “interfacing
iconicity”. It aligns three distinct registers of triadic forms, each in accordance with a central governing logic. Section 3.1 introduces the governing logic, the verb-noun-adjective triad. Sections 3.2, 3.3, and 3.4 each address one of the three registers in the order of interior, exterior, and intermediary.

In Chapter 4, we observe an article published in midst of this study. Entitled “The Coming Software Apocalypse”, it reiterates our central concerns and discusses efforts to address them. We consider these endeavors in relation to our own. In subsections 4.1, 4.2, and 4.3, we examine three platforms whose goals are (in some respect) adjacent to our own.

Chapter 5 presents the prototype that accompanies this document, an HTML rendering platform. We discuss the specific features of this prototype platform as well as our reasoning for build it around HTML. We assess this initial platform by the diagrammatic criteria laid forth in Chapter 3. Sections 5.1, 5.2, and 5.3 correspond to the three registers in Chapter 3, in reverse order (5.1 – Intermediary, 5.2 – Exterior, and 5.3 – Interior).

Chapter 6 documents our implementation in technical terms. Section 6.1 lists the libraries used in development. Sections 6.2 – Graphics engine, 6.3 – Data processing, and 6.4 – User input/interaction each address a category of program operations. In Chapter 7 we offer our final remarks on the current and long-term status of this project.
2.1 Interface

The title “Interfacing Iconicity” indicates our analytic point of entry, the interface. Branden Hookway’s *Interface* (Hookway 2014) [3] affords an abstract basis for interpreting the term. The very first statement denotes interface as “a relation within technology, rather than as a technology itself” (Hookway 2014, p.1) [3]. This distinguishes the interface concept from colloquial renditions thereof, shifting emphasis away from the specific front-end suites typically associated with the term. Hookway further stipulates that in forming the common boundary between entities, an “interface” represents the process of distinction between them and the status of relations among them. As such, interfacing predicates the instantiation of the use case. It marks the delineation between actor and system, the moment in which they “enter into an active relation with one another” (Hookway 2014, p.1) [3].

The statement, “the interface brings into effect its own illusory disappearance” (Hookway 2014, p.14) [3], offers an intriguing corollary, that “as a part of an active relation” the interface enacts “the concealing of constituent activities within the production of an overall trajectory” (Hookway 2014, p.14) [3]. A kind of occlusion accompanies the act of displaying, implicitly. The interfacing boundary between two
elements necessarily occludes their zone of intersection as it illuminates certain specific relations traversing this zone. For instance, the computer keyboard presents an implicit threshold between linguistic conceptualization and “digital” renditions thereof; it does so by occluding the implementation details of this translation. We characterize this condition as the occluding facet of interfacing, the outward “facing” representing an interior system of relations. These mentions of “disappearance” and “concealing” invoke a language of optics and visibility. The stem “face” itself implies spatial orientation, an object’s visible side or surface. We can employ geometric representations to illustrate this exchange of presenting and occluding. As a mode of interface, geometric representations typically include a “display” surface and images projected upon that surface.

Figure 2.1: Cube (ink)
Consider Figure 2.1, depicting a cube in 3 dimensions. The cube is defined by its exterior, represented by a set of lines observing a specific orientation. “Interfacing” with this geometric system occurs in the moment that one disregards its constituent elements to effectively visualize the cube as an object, independent of the explicitly planar surface that hosts it. Other attributes along this display surface are strictly irrelevant, insofar as they do not interfere with the interfaced relations (e.g., irregularities in paper texture, line width, pixel alignment, etc.). Figure 2.2 is functionally equivalent to the former, at least in terms of geometric projection.

Figure 2.2: Cube (digital)
However, such a perspective belies significant differences in their methods of composition. One was drawn “by hand”, with lines of ink on paper, and scanned to produce a digital image. The other was contrived “digitally” as a rigid arrangement of pixel values. The specific traits that differentiate these two figures lie outside the scope of geometric representation and are necessarily occluded in an instance of geometric projection. These distinctions are only apparent through juxtaposition; these figures are operationally identical.

Figure 2.3: Interior architecture
The computer interface incorporates many such thresholds. The outward facet of
the computer interface occludes electronic signal processing with symbolic
representations thereof, receiving input signals from peripheral hardware and rendering
specific output signals to audio-visual displays. On the outermost surface, it involves
input hardware and output hardware that respectively send and receive information
to/from a computer system. Figure 2.3 enumerates some of this scheme’s intermediate
thresholds, revealing corresponding components within the computer’s interior
architecture.

With regards to interfacing, each successive computer software system may offer
a set of divergent successors, and each successor system can offer its own set of distinct
possibilities. In the development of a computer program, the choice of hardware platform
may be followed by a choice of operating system, which may be followed by a choice of
corresponding language(s), a choice which may itself be followed by a choice of
corresponding libraries. This permits near innumerable divergent permutations. The
specific way by which a program is composed depends on the target operating system
(OS), the languages supported by that OS, the libraries accompanying those languages
(which must also be supported by the OS), and other contributing factors (such as
peripheral devices, system hardware, drivers, and other software). This provides the
nascent potential for divarication across computer programs, and by extension,
divarication in the ways computers may be used. Given the property of occlusion, we
assert that successive instances of interfacing upon a domain necessarily produce
divarication thereon. The shared properties and common origins of discrete subdomains
become occluded from view as a function of their emanation.
2.2 Iconicity

The second word of the title, “iconicity”, is positioned in response to “interfacing”, and more specifically to its occlusive tendency. “Iconicity” denotes the properties allocated to the icon, as one of the three kinds of sign (Peirce 1885) [4]. We should note that in computer science, acknowledgements of the work of Charles Sanders Peirce are relatively infrequent. This appears to be at odds with the abundant use of certain conceits, at least implicitly. For instance, “type” and “token” are two terms of another Peircean triad (of type, token, and tone) [5] ubiquitous within computer science. To quote *The Theory of Parsing, Compiling, and Translation* once more:

> It is the job of the lexical analyzer to group together certain terminal characters into single syntactic entities, called tokens. What constitutes a token is implied by the specification of a programming language. (Aho & Ullman 1972 p. 60) [2]

Peirce’s influence is certainly present, though unapparent. Although Aho and Ullman do not mention Peirce by name, they do include multiple citations of Noam Chomsky. Chomsky is certainly aware of Pierce, having cited him on multiple occasions. Chomsky may be responsible for the propagation of Peirce’s triadic entities within his region of expertise. Attributions listed in the *Oxford English Dictionary* corroborate this connection by affiliation.
Our deference to Peirce goes well beyond this triad of type, token, and tone. Another triad of icon, index, and symbol denotes three kinds of sign, differentiated by their mode of representation.

Signs are of three kinds,

1st, the icon, which represents its object by virtue of a character which it would equally possess did the object and the interpreting mind not exist;

2nd, the index, which represents its object by virtue of a character which it could not possess did the object not exist, but which it would equally possess did the interpreting mind not operate;

3rd, the symbol, which represents its object by virtue of a character which is conferred upon it by an operation of the mind.

(Peirce 1899-1900) [7]

With these categories of sign in hand, one can immediately observe that computer representations possess an almost pathological compulsion towards symbolic schemes.
Programs are so often designed to be used, rather than understood. Symbolic representations often insulate interior logic (or make it utterly invisible).

Table 2.1: Data size chart for individual UTF-8 characters. [8]

<table>
<thead>
<tr>
<th>Unit</th>
<th>Range</th>
<th>Bits</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit</td>
<td>0 – 1</td>
<td>[0] - [1]</td>
<td></td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>0 – 15</td>
<td>[0000] - [1111]</td>
<td>[0] – [F]</td>
</tr>
<tr>
<td>Byte (8bit)</td>
<td>0 – 255</td>
<td>[00000000] - [11111111]</td>
<td>[00] - [FF]</td>
</tr>
<tr>
<td>UTF-8 Character Codes</td>
<td>0 - 10175</td>
<td>[00000000000000] - [10011110111111]</td>
<td>[0000] – [27BF]</td>
</tr>
</tbody>
</table>

The UTF-8 encoding scheme provides an example. The Unicode Transformation Format (UTF) is a standardized character set meant to accommodate all existing character sets. UTF-8 (typical for email and for web pages) encoding assigns every character a byte-wise value, using between 1 and 4 bytes. Consequently, the Greek capital letter lambda Λ corresponds to 923, or 0000001110011011, in binary. The relationship between Λ and 0000001110011011 is in no way evident, and observing this connection does not yield any particular insight into either entity. This observation provides an explanation for the near impenetrability of divaricated boundaries. It again certifies divarication as an accurate descriptor for a diverse ecology (of software and related use cases) inhibited by strictly limited inter-compatibility.

On the icon, Peirce states:

For a great distinguishing property in the icon is that by the direct observation of it other truths concerning its object can be discovered than
According to Peirce’s definition, the icon denotes a sign whose form mirrors its referent. In contrast, the symbol achieves representation through deliberate, yet arbitrary association with its referent. That said, the icon is not necessarily a guarantor against divarication. A more strictly iconic computer system would presumably exhibit a physical form aligned to its exact operation, but this does not necessarily present a feasible solution.

The Turing Machine elicits one such example; as a mechanical device that physically demonstrates computation (or rather, computability itself). Alan Turing’s paper, “On Computable Numbers, with an Application to the Entscheidungsproblem”, provides the first description of this machine. It articulates “the process of computing a real number”, supplementing Turing’s definition of the set of computable numbers (Turing 1936, p.231) [10]. In terms of mechanical components, it includes:

- A “tape” divided into discrete segments, where each segment can either be blank or bear a single symbol.
- A “head” affixed to the tape that is always fixed over one specific section, that which is currently “in the machine”. The head may write a symbol on a blank section or erase its symbol if it’s not blank. It may also shift the tape once to the left or right, placing itself over the corresponding tape section.
- A finite number of conditions \(q_1, q_2, \ldots, q_n\) denoted as m-configurations, maintained by the machine itself.
The symbol on the tape segment currently “in the machine” is known as the scanned symbol, denoted by $Q(r)$, where $r$ corresponds to the numerical order $[1, ..., n]$ of that tape section. In any given moment, the machine’s current behavior is determined by the pair $(q_n, Q(r))$, the machine’s current m-configuration and its current scanned symbol.

$$M = \{\{q_0, q_1, q_2, q_3, q_4\}, \{0, 1\}, \{0, 1, X, Y, B\}, \delta, q_0, B, \{q_4\}\}$$

Figure 2.5: A 7-tuple formal specification of a TM

A common formal description for the Turing machine (as a 7-Tuple) is presented in the *Introduction to Automata Theory, Languages, and Computation* (2006) by John E. Hopcroft and Jeffrey D. Ullman. Rajeez Motwani contributed to later editions.

<table>
<thead>
<tr>
<th>State</th>
<th>0</th>
<th>1</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$(q_1, X, R)$</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$(q_1, 0, R)$</td>
<td>$(q_2, Y, L)$</td>
<td>–</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$(q_2, 0, L)$</td>
<td>–</td>
<td>$(q_0, X, R)$</td>
</tr>
<tr>
<td>$q_3$</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>$q_4$</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

Figure 8.9: A Turing machine to accept $\{0^n1^n | n \geq 1\}$

Figure 2.6: Transition table corresponding to the 7-tuple TM
Though the Turing Machine depicts a comprehensive analog to the fundamental process of computing, it is limited to the interior mechanics of the computer as a singular, reacting apparatus. It does not address the environment within which computers operate; or rather, it was not designed for that purpose. However, as an abstract schematic, the Turing Machine’s utility has far outpaced Turing’s endeavor to specify computability. Ultimately, its demonstrative capacity is potent, despite the constraints imposed by the physical parameters it abides.

Given that the potential implementations of an icon are multifold and not necessarily pertinent to our ends, this deployment of iconicity demands explicit specification. In Interface (Hookway 2014, p.59) [3], Hookway documents the physical context (specifically, in the field of fluid dynamics) from which interface emerged, and thereby distinguishes his conceptual designation. Similarly, we circumvent the physical (in particular, the optical and material) connotations of Peirce’s specification in order to emphasize a more general condition of iconicity. Thereby, iconicity denotes circumstances in which a particular representation unambiguously corresponds to its referent. As such, sufficiently robust interpretations of iconic entities should yield the means to construct their referents, to recreate that which they represent. That is the rendition of iconicity we employ.

2.3 Interfacing Iconicity

This juxtaposition of “interface” and “icon” reflects a contradiction in their properties as they’ve been articulated thus far. Interfacing functions in part through the
deliberate occlusion of interior relations, while our permutation of iconicity suggests an explicit delineation of that internal structure. Interfacing iconicity delegates a compromise of the latter to the confines of the former, it describes our pursuit of iconicity within some pre-established software platform. It recognizes that a prospective solution to divarication in computer use must also accommodate the contemporary paradigms of computer use. It must make itself available within the software ecosystem and possess the capacity to proliferate therein. As a mode of operation, interfacing iconicity facilitates an interior system of relations through an interface and simultaneously projects an iconic sub-representation of that system through that same interface. In the case of a software platform, this can manifest through an environment that both facilitates the execution of subprograms and displays the mechanics of said execution. This platform achieves a dual status: as a primary suite for both the use and the creation/ modification of its subprograms. It animates the correspondence between “programming” and “the program”, thereby providing a kind of visibility to relations between them.

The ultimate goal of this study is to determine the degree to which disparate computer use cases might be tethered together within such a software platform, one that emulates iconicity throughout its internal ecology of subprograms and the mechanics of their composition, compilation, and execution. To meet this criterion, the platform must, through its form, indicate the relationships between these aspects as they are engaged. In short, an interface for such a platform would express the processes implicit in subprogram activity. Here we state an ambition to pursue a specifically structural iconicity, as opposed to (for example) conventional visual metaphors. Modern GUIs already leverage visually instructive representations, such as of “folders” and “files”,
whose properties deliberately correspond to their material analogues. However, this visual imitation is inexact. It introduces the baseline concept of computer “files” and “folders”, but this representation does not communicate the full extent of their properties. For instance, “files” and “folders” they are not bound by material constraints, files can be as large as the system permits and folder nested can extend far beyond any reasonable physical equivalent.

In effect, this platform would render (or expose) the various thresholds of exchange between written code and the outcome of its interpretation, to the extent that the mechanics involved therein become apparent and unambiguous. An effective implementation would diminish the stratification between computer use cases, by offering users opportunities to observe that which is known by programmers. Following from the precedent of *Cybernetics* (Wiener 1948, 1961) [1], this mandate does not aspire to efface the distinctions between computer use cases, but rather to enhance inter-compatibility across them. It is not attempting to erode barriers so much as it endeavors to magnify their permeability, to form conduits through the volumes nullified by divarication. We do not premise that occlusion (defined here as a consequence of interfacing) can be negated outright, but rather that iconicity, if calibrated properly, might contravene the effect of divarication.

### 2.4 Diagrams

In one description of the term “diagram”, Peirce writes:

…a Diagram is an Icon of a set of rationally related objects. By rationally related, I mean that there is between them, not merely one of those relations which we
know by experience, but know not how to comprehend, but one of those relations which anybody who reasons at all must have an inward acquaintance with. This is not a sufficient definition, but just now I will go no further, except that I will say that the Diagram not only represents the related correlates, but also, and much more definitely represents the relations between them, as so many objects of the Icon. (Stjernfelt 2007, p. 90) [9]

Thusly construed, as a calibrated permutation of the icon, the diagram is an adept paradigm for an iconic portrayal of computation, given that computation is (at least, in many respects) more closely approximated as a system of relations than as a discrete, tangible entity. The ubiquity of diagrams in computer studies can buttress this assertion. State diagrams, used in a variety of ways, are one such example. The following state diagram (Figure 2.7) represents the Turing Machine depicted on pages 11-12 of this document.

![Figure 2.7: Hopcroft & Ullman transition state diagram of a Turing machine (Hopcroft, Ullman, Motwani 2000, p.324) [11]](image)
Through the relations it illustrates, this diagram conveys all the information necessary to compose an equivalent machine. Furthermore, interpreting one instance of a state diagram conveys the method to interpret and compose other such diagrams, and thereby to create new Turing Machine configurations. This Turing Machine depiction is only one of many types of computation-oriented state diagrams.

In particular, diagrams can convey abstract architectural conceits. In Jeremy Bentham’s oft-trodden “Principle of Construction”, the Panopticon demonstrates the tremendous potential of one such diagrammatic model.

Figure 2.8: The Panopticon (Bentham 1787, p. 369-439) [12]

Michel Foucault’s *Discipline & Punish: The Birth of the Prison* offers a rigorous examination of the Panopticon’s diagrammatic properties. In the chapter entitled
“Panopticism”, the following passage ordains a contextual shift from the security protocols of a medieval plague town to the more abstract, “panoptic” model:

The Panopticon, on the other hand, must be understood as a generalizable model of functioning; a way of defining power relations in terms of the everyday life of man. (Foucault 1975, p.205) [13]

The plague town and Panopticon enterprises both instruct mechanisms of control and confinement through an explicit blueprint. Originally, the Panopticon formed the schematic for a prison structure, designed to impose disciplinary order through strategically allocated visibility. However, as a diagrammatic model, the Panopticon transcends the exact physical parameters of this initial proposition. In the conclusion of the previously cited paragraph, Foucault states:

But the Panopticon must not be understood as a dream building: it is the diagram of a mechanism of power reduced to its ideal form; its functioning, abstracted from any obstacle, resistance or fiction, must be represented as a pure architectural and optical system: it is in fact a figure of political technology that may and must be detached from any specific use.

In terms corresponding to this document, the prisoners, observer(s) and their arrangement within the Panopticon form the iconic basis for a diagrammatic codification
of their power relation, wherein the notion of power derives itself from the interfacing of visibility. Foucault parses out this active relation:

He who is subjected to a field of visibility, and who knows it, assumes responsibility for the constraints of power; he makes them play spontaneously upon himself; he inscribes in himself the power relation in which he simultaneously plays both roles; he becomes the principle of his own subjection.

This affirms the Panopticon’s iconicity, indicating that the mere expression of a given panoptic representation (easily accomplished through diagrams) is sufficient to instigate the operation of the active relation it represents.

The example of the Panopticon is particularly compelling for this study, as it demonstrates the sheer potency a diagrammatic model might attain. Foucault further observes that “It could be used as a machine to carry out experiments, to alter behavior, to train correct individuals” (Foucault 1975, p.203) [13]. Thereby, various permutations of the panoptic model might produce new knowledge and achieve new results, well beyond the scope of its initial purview. This perhaps distinguishes the Panopticon from many other diagrammatic entities, such as the Turing Machine. The panoptic relation persists beyond or in spite of any specific context. It regularly manifests in the structure of hospitals, schools, factories, and in the application of continuous video surveillance (and more recently, the electronic surveillance of online activity, facilitated by computers). Each instance asserts and maintains the panoptic paradigm of visibility. We will harness this powerful property with our own diagrammatic model.
CHAPTER 3

THE FORM-PROCESS DIAGRAM

The following figure illustrates the diagrammatic model that this thesis serves and endeavors to certify.

Figure 3.1: Form-process diagram
This diagram attempts to codify a generalized, abstract system of relations, identified as “form-process”. We define “form-process” as the sets of relations between three discrete, yet mutually constitutive aspects delimiting a continuous, cyclic trajectory. Form-process refers to the interchange through which “forms” manifest and “forming” occurs, the methods through which an entity becomes determinable and potentially influences schemes of determination itself. The diagram depicts and defines form-process through its alignment of these triadic relations across conceptual categories.

Within the context of this diagram, a “circuit” is defined as any distinct cyclic path between elements. Each circuit follows the governing logic depicted in Figure 3.2 and detailed in section 3.1. This verb-noun-adjective triad specifies an orientation of three aspects and the characteristics associated with each linguistic form. It establishes the convention for interpreting and engaging this diagram. To employ the diagram, one must begin with an entry circuit and then draw other circuits into correspondence. In this chapter, we will identify and describe three primary circuits. The interior triad of perform, enunciation, and determinative (section 3.2) comprises our initial circuit, a wholly abstract rendition of form-process. In contrast, the exterior triad of execute, composition, and analytic (section 3.3) represents the specific scenario we intend to address, namely computation in practical terms. Its aspects are the elements of computer use that most closely align to the governing logic and to the interior triad’s aspects. The render-inscription-modal circuit (section 3.4) serves a strictly intermediary role between the exterior and interior circuits. It depicts the objective our proposed software platform
should meet, a field intersection that foregrounds interior logic with its exterior manifestation.

3.1 Verb-Noun-Adjective: The Governing Logic

![Diagram of verb-noun-adjective triad]

Figure 3.2: Verb – Noun – Adjective triad.

Our demonstration begins with the persistent triad of verb - noun - adjective. It directs a trajectory across the three linguistic forms. It observes three orientations of influence:

1. verb > noun
2. noun > adjective
3. adjective > verb

The 1st threshold |verb > noun| indicates that nouns are directed by verbs. A noun emerges from a context of being or doing. The 2nd threshold |noun > adjective| indicates that adjectives are directed by nouns. An adjective emerges in reference to a noun. The 3rd threshold |adjective > verb| indicates that verbs are determined by adjectives. This threshold relation is not necessarily as apparent as the first two. Verbs convey either actions or states of being. An action verb is equivalent to the composition
of a state-of-being designator (is, was, be, do) and an adjective describing that state. Thereby, verbs emerge in reference to some adjective.

As an independent element, this verb - noun - adjective conceit is utterly arbitrary. Its orientation is not explicitly warranted; the stipulated directions and positions are not derived by an independent argument. However, as a persistent diagrammatic relation, it defines parameters that inform every other sub-diagram, and the diagram as whole. Contrived as a facet thereof, its significance only becomes evident in instances of correspondence.

3.2 Perform-Enunciation-Determinative: Interior Circuit

![Diagram](image)

Figure 3.3: Interior circuit of form-process diagram.

Again, we observe this relation set in isolation. Independently, this triad represents the interior relations of form-process as a directed exchange between the perform, enunciation, and determinative aspects. The trajectory through these aspects
orients their conjoined constitution. Applying the verb-noun-adjective syntax yields additional distinguishing properties.

**Perform (v.):** The aspect of form-process concerning the enactment of forms. The continuous “happening” through which forms exist and express (such that “being” is equivalent to “performing”). As such, performing is predicated by determining; a form must be determined to “appear” or be known to exist, and in that moment, it is performing.

- **Negation of noun form:** Performing is continuous, indeterminate, and arguably indiscriminate. To describe performance as a singular occurrence is to suggest that individual instances of performing can be isolated from this continuity. Performing should not be delimited thusly, as it pertains to everything that occurs, or occurrence itself.

- **Negation of adjective form:** In this context, attributing the quality of “performative” is utterly redundant, as attribution in and of itself would not be possible on a ‘non-performative’ entity. Explicitly, an entity can’t be “non-performative”; to say that it is performative is to say that it is.

**Enunciation (n.):** The aspect of form-process concerning specific forms that bear the potential to influence, and to change. An enunciation occurs in the moment that a form induces change upon determinative conditions. Enunciations may only emerge from a state of performing; performing elicits the possibility of change, and an enunciation is a realized instance of change.
• **Negation of adjective form**: To apply “enunciative” as a property suggests that a clear distinction can be made between that which is “enunciative” and that which isn’t. This can lead into an unproductive and irrelevant discussion over what makes something “enunciative”. We intend to identify the instance of change in and of itself; how a form became enunciative is not necessarily observable, as enunciations are known only by their effects. The condition of “being enunciative” is a tenuous quality to ascribe, as it implies a persistence that is in no way assured.

• **Negation of verb form**: This negation is similar, “enunciating” as an action shifts emphasis toward defining the conditions for performing enunciations, rather than the enunciations themselves. Specifically, we mean to identify the inductions of change, and to avoid the fruitless task of classifying how they can occur.

**Determinative – determinative (adj.)**: The aspect of form-process concerning form determination, and thereby performing. The determinative refers to that which determines, or the particular ways determining occurs. It is a qualitative designation. Determinative properties are developed, influenced, and adjusted by enunciations. A determinative effect can be portrayed (at least in part) as the aggregate influence of enunciation(s). However, “knowing” a determinative constituency in its entirety would require observing every potentially determinable form, as well as any additional forms made determinable by resultant enunciations. Therefore, we designate determinative as necessarily indefinite.
• **Negation of verb form:** To emphasize determining (v.) is to imply it is distinguishable from performing as another active (verbal) aspect of form-process. However, determining can be considered as equivalent to the enactment of enunciations (specifically, enunciative forms), an operation already implicit to the trajectory.

• **Negation of noun form:** Describing determinative as a singular entity (or a noun), is unnecessary, as it is equivalent to the aggregate influence of enunciations across a continuum. Determination (n.) itself merely refers to the act of determining (v.), which we’ve already shown to be redundant.

This triadic depiction of form-process can only be certified through its application. It must be shown to yield insight into its correlates. The following alignment to the Peircean triad of type, token, and tone establishes an initial context for this interior relation scheme and the abstract basis of form-process.

**Type – Determinative**

Peirce explains:

There will ordinarily be about twenty the’s on a page, and of course they count as twenty words. In another sense of the word “word,” however, there is but one word “the” in the English language; and it is impossible that this word should lie visibly on a page or be heard in any voice, for the reason that it is not a Single thing or Single event. It does not exist; it only determines things that do exist. Such a definitely significant Form, I propose to term a Type.
In this case, Types are explicitly described as determining “things that do exist”. Furthermore, the “Type” itself does not “exist”, it is defined as a determinative characteristic, as opposed to an explicit entity.

**Token - Enunciation**

According to Peirce:

A Single event which happens once and whose identity is limited to that one happening or a Single object or thing which is in some single place at any one instant of time, such event or thing being significant only as occurring just when and where it does, such as this or that word on a single line of a single page of a single copy of a book, I will venture to call a Token. (Peirce 1906, p.506) [5]

A token, in being a single instance, “occurring just when and where it does” is most certainly comparable to an enunciation, in accordance with the verb-noun-adjective conceit.

**Tone – Perform**

Again, Peirce:

An indefinite significant character such as a tone of voice can neither be called a Type nor a Token. I propose to call such a Sign a Tone.

(Peirce 1906, p.506) [5]
For a “possible” Sign I have no better designation than a Tone, though I am considering replacing this by “Mark.” (Peirce 1908) [14]

Peirce’s concept of tone is the most difficult to parse, as it seems to be underdeveloped in comparison to type and token, and it is frequently omitted from the triad in favor of the type/token dichotomy mentioned previously. However, since the condition of possibility to per-form is circumscribed by the determinative, we can state that “perform” designates the possibilities of forms, since forms enter a state of being through per-forming. Moreover, the determinative is certainly significant in the role of outlining possibility, but also necessarily indefinite, as previously discussed. Thereby we arrive at a tone-compatible term, if only marginally.

However, distinctions between these two triads demand acknowledgement. For Peirce, the terms “type”, “token”, and “tone” apparently denote aspects of the sign. In contrast, the determinative-enunciation-perform triad concerns aspects of form-process (as opposed to just the form itself). Form-process deliberately preempts the act of signing, to prescribe the context in which types, tokens, and tones operate. Consequently, while types are certainly determinative, they are only one facet through which determination may occur. Similarly, as specifically recognized instances, all tokens are enunciations within their respective “moments”, but an enunciation is not necessarily a token. Finally, tone is an implicit facet of performing, a possible precedent for enunciations. Through tone, tokens may convey more than what is attributed to their type.
Tonality presents a possibility for enunciation. “Tone” is a mechanism by which enunciations may manifest.

### 3.3 Execute-Composition-Analytic: Exterior Circuit

![Diagram showing execute, compose, and analyze as interconnected processes.]

Figure 3.4: Exterior circuit of form-process diagram.

With form-process laid out in abstract, we shift emphasis to the exterior circuit of execute-compose-analyze. This more tangible triad presents an analog to computation in common practice. Most plainly, compositions are “code” itself. The outcome of
executing a composition is determined through some analytic standard. The verb-noun-adjective conceit facilitates more detailed descriptions of these aspects.

**Execute (v.):** The acting of computing itself. A program exists to be executed. If it does not execute, it is not a program. As a verb, “execute” denotes a continuous state, rather than a discrete interval therein. Furthermore, this verbal term applies to any instance that can be considered computation. The noun form (of execution-s) suggests an enclosure between determinate starting and ending points, but our primary interest is in between, and the possibilities at any moment of computation. The adjective “executive” is even less productive. Like “performative”, it is an arbitrary designation. Anything not “executive” is also unobservable within a computational system.

**Composition (n.):** The sequences, scripts, and code submitted to a computing machine are all considered to be compositions. As a noun, “composition” concerns specifically individual, discrete instances. A composition can be submitted for execution, and the outcome of its execution is determined by some analytic criteria. To compose (v.) is to assemble a sequence of symbols for computation. This refers in part to an external (presumably human) participant’s cognition and occurs largely outside the physical computer. Though we can assume this activity (of composing) occurs implicitly, it doesn’t belong on the path connecting “composition” to “analytic” and “execute”. From the perspective of computing machine, composing is unobservable. At best, it can differentiate between iterations of compositions received. The adjective “composite” is
essentially redundant, as every computation expression can be reduced to a sequence of 0s and 1s, and can be considered composite.

**Analytic (adj.):** The analytic aspect of computation refers to criteria that decide the outcome of execution, that which informs parsing (of code, expressions, etc.). As an adjective, it represents the characteristics of these criteria, rather than their implementation (in terms of a programming language, this concerns its properties, rather than a specific compiler/interpreter). Parsers and compilers are instruments of analytic enforcement. The verb form, “analyze”, occurs implicitly within “execute” (though it may be more accurate to state that “executing” implies analytic processes). An execution necessarily abides analytic preconditions. The noun form, “analysis”, presents this imposition of preconditions as a discrete interval within execution, but this assumption is baseless and potentially misleading.

### 3.4 Render-Inscription-Modal: Intermediary Circuit

The relative orientation of this circuit, as shown in Figure 3.5, indicates its status as an approximation between the interior abstraction and the exterior analog. It magnifies their intersections, delimiting a volume of correspondences between them. As such, it represents the endeavor of interfacing iconicity, an alignment of interior relations to an exterior scope of operation. This triad, unlike the others, is more speculative than it is definitive. The terms “render”, “inscription”, and “mode” are less specific by design. As conduits between interior and exterior circuits, their properties are contingent upon those endpoints. We will examine each aspect in greater detail to derive an implementation
compatible with a software platform. These aspects are identified by conjoint terms, produced through synthesis of their interior and exterior correspondents.

Figure 3.5: Intermediary circuit of form-process diagram (highlighted).

Render (v.): The verbal aspect draws “perform” and “execute” into association. The term “render” (v.) offers definitions that evoke both. Performing aligns with the fourth definition category offered by the *Oxford English Dictionary*: “To bring into a specified condition” (OED, IV.) [15]. The definition reflects the enactment of forms through determinative conditions and criteria. “Execution” aligns with the definition “To express or represent”, and especially with the variant, “To reproduce or express in another language; to translate” (OED, I., 2.a.). Through execution, compositions are
translated into machine instructions. “Render” also carries specific meaning with respect to computer graphics, denoting the conversion of image data into a visual format (OED, IV.22.a).

As a conjoint term, “render” tethers the otherwise abstract “perform” to “execute”, such that it concerns performing specifically in relation to executing. To reiterate, “execute” is to run, operate, or otherwise process a composition through a computing machine. To render is to provide a perspective that conveys the interior mechanisms of execution in tandem with its own continuous operation.

**Inscription (n.):** “Composition” and “enunciation” adjoin as “inscription”. “Inscription” can refer to “The action of inscribing; the action of writing upon or in something” and to “that which is inscribed” (OED, 1., 2.) [16]. In this context, the first definition suits composition, and the second enunciation. Consequently, composition is an action that elicits enunciation. Inscription represents the link between composition and enunciation; or rather, it exhibits a composition together with the effect it incurs upon computation (an analytic process). This term can refer to a composition in its whole (such as an entire script), or with regards to individual lines and/or language tokens within the code. In any case, inscription pairs expressions with their specific (or rather, determinative) effects.

**Modal (adj.):** “Modal” denotes the analytic, along with its determinative influence. “Analytic” identifies the traits and characteristics of a programming language, the way expressions are interpreted. “Determinative” designates an indefinite
contingency of formal possibility. The term “modal” is perhaps most immediately associated with to modal logic, indicating a qualitative proposition; that is, a proposition “involving the affirmation of negation of possibility, impossibility, necessity, or contingency; that contains an adverb or adverbial phrase, or in which the predicate is affirmed or denied of the subject with a qualification”. (OED A. 1.) [17] A computing-specific definition recognizes “modal” as “Designating a program, system, or user interaction which requires the user to switch between different modes of operation to perform different types of action.” (OED A. 8.). While both definitions are pertinent, they are not consistent with one another. The first (referring to modal logic), in designating qualitative propositions and more generally qualitative assignment, conforms with determinative. In contrast, the computation-oriented definition anticipates practical application, indicating explicit delineation between “modes” of operation.

“Modal” adjoins the analytic aspect of computation with the interior logic it implicitly expresses. The modal aspect fixates on the interactions between language and expression, moments when the operative characteristics of a language are most apparent. It is reactive by necessity (“render” and “inscription” are comparatively proactive), as it conveys analytic properties only when they are invoked. Modal representation pronounces the association between analytic validation (of a subprogram) and the attendant determinative invocations. In implementation, the modal aspect shows how parsing processes inform a subprogram’s execution.

Our diagrammatic explication concludes with this last circuit, which provides our point of departure from speculation to experimentation. We contend that an emulation of this intermediary circuit approximates interfaced iconicity, and it should thereby produce
software inoculated with divarication-resistant features. To design a software platform in accordance with this scheme, one must choose an initial circuit to implement, and then from that instantiation draw forth additional circuits as they become available. A rudimentary articulation of a single “sub-program” circuit forms our initial development basis. A complete implementation of such a circuit should exhibit the three intermediary aspects and express the relationship between them. In doing so, it should invoke both the interior and exterior form-process circuits.
CHAPTER 4
IDENTIFYING & ADDRESSING ISSUES OF DIVARICATION

As software permeates through every dimension of industry, governance, media, and social engagement, every successive gap in system comprehension magnifies both the cost of and potential for systemic failure. In “The Coming Software Apocalypse”, James Somers suggests that contemporary programmers inadvertently produce code of impenetrable complexity.

“The problem is that software engineers don’t understand the problem they’re trying to solve, and don’t care to,” says Leveson, the MIT software-safety expert. The reason is that they’re too wrapped up in getting their code to work. “Software engineers like to provide all kinds of tools and stuff for coding errors,” she says, referring to IDEs. “The serious problems that have happened with software have to do with requirements, not coding errors.” When you’re writing code that controls a car’s throttle, for instance, what’s important is the rules about when and how and by how much to open it. But these systems have become so complicated that hardly anyone can keep them straight in their head. “There’s 100 million lines of code in cars now,” Leveson says. “You just cannot anticipate all these things.” (Somers 2017) [18]

The article cites recent disasters incurred by faulty, incomprehensible code. It warns that more are sure to follow, especially as software approaches more sensitive tasks, such as
self-driving cars. The article draws attention to industry-wide impact left by Bret Victor’s “Inventing on Principle” talk, given in 2012.

By the time he gave the talk that made his name, the one that Resig and Granger saw in early 2012, Victor had finally landed upon the principle that seemed to thread through all of his work. (He actually called the talk “Inventing on Principle.”) The principle was this: “Creators need an immediate connection to what they’re creating.” The problem with programming was that it violated the principle. That’s why software systems were so hard to think about, and so rife with bugs: The programmer, staring at a page of text, was abstracted from whatever it was they were actually making. [18]

Victor, a former Apple UI designer (currently a researcher for Dynamicland), drew the attention of several prominent developers, including Chris Granger (who worked as lead developer for Microsoft’s Visual Studio IDE) and John Resig (lead developer of jQuery). Many attempts to realize Victor’s idealized state of programming have been made in recent years. These include independent projects, such as Granger’s Light Table [19], and efforts from massive industry firms (such Apple’s attempt to incorporate similar features into the Swift programming language [20]).

While Victor apparently recognized endemic issues in the field of programming, the approach he advocates (at least as addressed in this article) seems just as likely to perpetuate divarication as it is to assuage these effects. Victor compels programmers to build toolsets that don’t necessarily rely on manual text input. To demonstrate this concept, the article describes programming interfaces that employ more tactile
interaction, such as the code-free digital image editing made possible Adobe’s Photoshop. These environments allow for more intuitive development, but by deliberately occluding system properties. Photoshop employs visually instructive “icons”, similar to the “files” and “folders” example briefly discussed in section 2.3 (albeit more sophisticated). These tools are limited to the extent of the features they accommodate, and they do not necessarily prepare one for development without them. Later in this chapter, we’ll offer Adobe Dreamweaver (Photoshop’s sibling product) as an example [21]. As a web development suite (and unlike Photoshop), Dreamweaver is intended to produce interactive and comparatively dynamic objects, and therefore a more suitable subject for this discussion.

We have argued that divarication is a primary factor of these widely-recognized issues in software use and programming, and that divarication is perpetuated by constant expansion and sub-specification. We then claimed that the abstract paradigm of interfacing iconicity could counteract the effects of divarication, and we have endeavored to produce design schematics for such a platform, one that emulates the condition of interfacing iconicity, through diagrammatic principles. In the following sections, we assess other software engineering techniques by the standards and principles established in this document.

4.1 Visual Programming – MIT’s App Inventor

The MIT App Inventor project is introduced as “…an intuitive, visual programming environment that allows everyone – even children – to build fully functional apps for smartphones and tablets” [22]. It offers the opportunity to create
android-compliant applications (for mobile hardware) through a visual programming interface. In this environment, “coding” is accomplished by arranging structural elements (representing variables, operators, constants, and functions) into expressions as if they were interlocking puzzle pieces. App Inventor provides two primary development interfaces. The “Designer” interface (Figure 4.1) facilitates user interface design and multimedia asset management (image, sound, and so on). The “Blocks” interface contains the visual programming workspace, where expressions are assembled and assigned out of color-coded “blocks”.

This platform turns programming into a conceptually simple and aesthetically intuitive task. It only requires selecting and fitting the appropriate blocks together. This activity implicitly distinguishes operator and operand; operator “notches” are fit with operand “wedges”. The platform introduces programming fundamentals such as loops and conditional statements through its own imitative analogs (see Figure 4.2).

The App Inventor does indeed streamline android development tremendously. Projects can be exported directly to mobile hardware with the “MIT App Inventor Companion” app. Once the app is installed on a compatible device, it can retrieve projects by scanning a QR code generated by the programming interface [23]. The interface design tools are configured specifically with mobile screen dimensions and hardware in mind. In this respect, it offers a hands-on approach to mobile development, however limited.
Figure 4.1: MIT App Inventor – “Designer” interface. [22]
Figure 4.2: MIT App Inventor – “Blocks” interface. [22]
The App Inventor platform demonstrates the effects of basic programming elements, but not necessarily how to use them outside of this carefully constructed space. It can show how loops and conditional statements work, but by automatically “snapping” the code blocks together, it denies the significance of precise syntax. The platform’s distinct characteristics impede the cultivation of externally applicable programming proficiency. As a specific example, the “segment” block produces a segment of a “text” string input, starting at the position designated by the “start” integer input, with the second integer input specifying length. The initial index value accepted by this block is 1 (1 refers to the first character of the text string) in contrast with the convention of 0 used throughout most programming languages. This subtle inconsistency bears significant implications, as it isn’t a requisite compromise for a visual programming platform. It indicates a design approach at odds with any instructive intentions. The App Inventor appears to inhibit its own instructional efficacy. Simply, it may not be of much assistance for one’s acclimation to programming.

In keeping with our arguments concerning divarication, we contend that sufficiently specialized development environments provoke the phenomenon of divarication; they delimit distinct modes of operation and restrict compatibility across them. In this case, App Inventor specializes in concealing the unappealing rigors of programming beneath a colorful and soft-edged veneer. In doing so, it erects a divergent boundary within the field of programming. It enables rapid application development and deployment, which is certainly impressive, but it relies on a very limited degree of engagement. It attempts to preclude errors entirely by enforcing strict rules for assembly
of the programming “blocks”, leaving developers completely unprepared for unexpected errors.

4.2 Professional Development Tools – Adobe Dreamweaver

With obvious differences set aside, Dreamweaver and App Inventor have similar general concepts. Both platforms facilitate creation in a previously established ecosystem. However, Dreamweaver is sold as a professional tool, and it is not explicitly educational. Instead, Dreamweaver is itself something “to be learned”, and has spawned training courses for this exact purpose. Dreamweaver is suite of tools built upon others, HTML (Hypertext Markup Language), CSS (Cascading Style Sheets), JavaScript, and more. It assembles a comprehensive web development kit by aggregating these components within its interface. This interface is the “product”, one whose potency evidently outweighs its imperfection. Though Dreamweaver itself must be “learned”, the many technical nuances under its supervision can be wholly ignored. It achieves success by hiding these internal systems beneath menus, buttons, and widgets.

Dreamweaver is an ideal sample case to consider in terms of divarication. It induces a specialization in “Dreamweaver use”. This is a distinct category of proficiency, and it is not necessarily transferrable to any other web development platform. Dreamweaver offers an intuitive venue for web development but is also a key contributor to divarication. A user of Dreamweaver still depends on a dense mesh of overlapping systems and components, even if they are well hidden. Furthermore, the software derives value from this very quality, the simplicity of control over complex systems. We do not mean to argue that Dreamweaver is built “incorrectly”, but rather that software and code
become impenetrable in the pursuit of desirable outcomes. Consequently, we assert that divarication must be addressed by explicit and persistent countermeasures.

4.3 Programming Feedback – Light Table

As mentioned earlier in this chapter, Light Table is a programming environment inspired by the “Inventing on Principle” talk. It distinguishes itself with features for extracting the flow of data through program code, which can even accompany execution simultaneously. The video currently featured on the project’s landing page [19] shows a workspace divided evenly between a web browser with real-time animation and a text editor containing JavaScript animation code. As variables are modified in the text editor, the animation in the browser adjusts accordingly. Light Table contains sophisticated tools for code evaluation, eschewing the need for manually written print statements that are usually a programmer’s only means to verify values at various stages of execution.

Light Table is open source and widely configurable, apparently making every effort to meet programmer’s needs. It supplements existing frameworks, instead of building itself on top of them. It is better described as a set of tools for experienced programmers. Though it’s not a fully constructed software platform, it provides a rare point of reference on real-time code feedback.
CHAPTER 5

PROTOTYPE DESCRIPTION

This chapter introduces the prototype implementing our diagrammatic design scheme (Chapter 3). We intend in this chapter to convince the reader that our software is a legitimate realization of the theory-driven arguments made in previous chapters (primarily 1-3). With constraints in both development time and resources to consider, we chose to develop this initial prototype as an HTML platform. This offered several advantages. HTML is ubiquitous and relatively approachable for those who lack programming experience; it can be introduced with very basic XML (Extensible Markup Language) scripts, totally excluding any CSS (Cascading Style Sheets) settings or JavaScript code. Once the fundamental syntax is understood, additional elements can be gradually introduced. Our development approach reflects this iterative quality of HTML, we have ignored CSS and JavaScript to focus instead on foundation for this platform’s distinguishing features.

In the absence of JavaScript (and unlike many other programming languages), HTML is static; the code cannot change/react in response to user interaction or perform operations in real-time. As such, these factors (user interaction, time elapsed, instances of operations) don’t need to be traced and presented alongside execution, at least not in the initial state of our prototype. A platform for static markup processing can be achieved far more quickly than one that relies on dynamic code, and with HTML we can develop a
rudimentary foundation and incorporate more dynamic elements in later iterations. Executing HTML elicits a (potentially interactive) visual artifact in the form of a “page”. Since HTML is visually demonstrative in execution, it significantly reduces the amount of abstraction required to portray a cohesive environment (assuming the exclusion of full JavaScript support, for the time being). We can instead focus on a foundation for the platform and concerns ourselves with data flow later on.

Following the diagrammatic explication of Chapter 3, our initial software prototype began with the identification of an entry circuit. In this case we began with the intermediary circuit (section 3.4). The intermediary (‘render – inscription – modal; see Figure 3.5) circuit emerges as a field of intersection between the interior (‘perform - enunciation - determinative”; see Figure 3.3) and exterior (“execute - composition - analytic”; see Figure 3.4) diagram circuits. The former (the interior) comprises a totally abstract depiction of form-process, while the latter (the exterior) explicitly represents computation in a compatible circuitous arrangement. Consequently, the intermediary circuit attempts a synthesis between these disparate scopes. It presents a potential foreground within which abstract comprehension accompanies and emerges through ordinary engagement. According to the arguments made in Chapters 1-4, that is the condition this prototype must fulfill to counteract divarication. The sections of this chapter address each of the three major triadic circuits featured in the form-process diagram (Figure 3.1) in relation to our prototype.
Figure 5.1: Prototype screenshot.
5.1 Intermediary Circuit Analysis

To implement the intermediary circuit (of “render-inscription-modal”), we first sought three viable “facets” that maintained the triad’s relational logic. In particular, we must replicate the “verb-noun-adjective” syntax detailed in 3.1. HTML conveniently offers two discrete (yet corresponding) aspects, the source “script” and the rendered “page”. These readily align with our established triadic scheme. The HTML script serves as the noun-form “inscription”, a discrete instance distinguished by the effect it exerts. The rendered HTML page is a continuous (and thereby, verbal) expression of the static inscription (this is more apparent with animation or interactive elements). It aligns with the “render” aspect of this triad.

To feature the full triad, it was necessary to contrive a “modal” facet corresponding appropriately to the render and inscription aspects. We describe the “modal” as a structuring/filtering effect, in this case, pertaining to the conditions that determine what construes HTML expressions, or how those expressions manifest. In noun form, one might be predisposed to emphasize an HTML parser/renderer in and of itself. In verb form, the emphasis is drawn towards processes of parsing/rendering. In the adjective form, we instead concern ourselves with the properties or characteristics of the language that are engaged within the circuit. Instead of exhaustively tabulating parsing/rendering protocols or producing an animated simulation of this process, we must show how HTML script is interpreted and rendered. This is, admittedly, a vague stipulation - but one we intend to qualify. Our modal facet displays a Document Object Model (DOM) representation of the HTML script. This virtual structure makes the syntactic hierarchy of HTML visible. When displayed in tandem with the HTML script
and the rendered HTML, it shows how HTML elements are identified and organized from a script and how they are rendered in a browser.

According to previously established specifications, this platform must exhibit the relationship between HTML “scripts” (inscriptions), their rendering, and the “modality” through which this occurs (in this case, the HTML grammar and syntax). We attempt to emulate this circuit with these three display facets:

- **Inscription**: A “source” script, displayed in plain text. In other words, what HTML looks like in a text editor. We will call this the “plain text view”.
- **Modal**: A virtual depiction of the page, as a Document Object Model (DOM) “tree” structure. We will call this the “DOM view”.
- **Render**: The rendered HTML, the “page” as it would appear in a browser. We will call this the “browser view”.

In its current state, the prototype reads an HTML file and produces three display fields. The first display shows the file as it would appear in a text editor. The second shows a DOM tree interpretation of this file. The third display imitates a web browser, showing the rendered HTML file. Thus, we have our triad. Once an HTML file is loaded in, it is displayed in the first facet (the inscription) as plain text. The modal properties invoked by this inscription manifest in the virtual DOM tree display. Finally, the last display facet emulates a standard web browser, revealing the HTML “render”. Thus, the triadic representation is complete. However, without a depiction of the directional paths featured in the diagram, this circuit is necessarily incomplete.
The circuit’s structure demands the expression of correspondence across these three display fields. Our program displays these relations in response to user input. Selections are made by clicking on a line of the script, a node of the DOM tree, or an element in the browser view. When a mouse click is detected, the program first determines whether it falls within the one of the three display fields. If so, that field is designated as the current selection and its border color changes. Content within that display may now be shifted vertically with the “up” and “down” arrow keys.

Figure 5.2: Selection in plain text view.

If the click location falls upon a specific line of text (plain text view), node (DOM view), or element (browser view), a border and background highlight appear over that item. The program then generates lines connecting the item to its correspondents within
the other two fields. Selecting a line of code in the plain text view (Figure 5.2) draws lines to any DOM nodes initiated on that line, which are then themselves connected to their browser view counterparts (when manifest).

Selecting a node in the DOM view (Figure 5.3) draws a line backwards to the appropriate text line. If it is a visible HTML element, a line is also drawn forward into the browser view to its location therein.

Figure 5.3: Selection in DOM tree view.
Finally, selecting an element within the browser view (Figure 5.4) draws a line backwards to that element’s location within the DOM view, and from there a second line backwards upon the line of code that generated that element.

![Diagram showing selection in browser view](image)

Figure 5.4: Selection in browser view.

These connecting lines superficially resemble the circuitous path through the aspects inscription, modal, and render. They direct the eye towards a precise subset of the on-screen information, allowing for more immediate isolation and inspection. This
concludes our entry circuit and the first iteration of our software prototype, at least with respect to the Render-Inscription-Mode triad.

### 5.2 Exterior Circuit Analysis

The exterior circuit, of execute-composition-analytic (3.3), identifies concrete aspects of computation and aligns them with form-process in abstract. The aspect “execute” refers to the process of execution, of “running” a program and anything occurring therein. “Composition” designates code, scripts, and other forms of strings that are to be computed. The “analytic” descriptor identifies the characteristics of a programming environment, or the specific parameters execution must abide. In this section we employ this triad to examine the technical capacity of our prototype in greater detail. We’ll discern the extent to which each aspect is portrayed within the prototype, and we’ll make practical comparisons between this prototype and contemporary software. This exercise will also establish development priorities for continuation beyond this iteration.

In this iteration, HTML execution is absolutely minimal. The “browser view” only renders standard HTML elements, ignoring CSS and JavaScript. The program recognizes the default display attributes for some standard elements. Many of these absent features (particularly CSS) are within immediate reach. They can be implemented in our engine as-is. However, other features, such as fractional font scaling and JavaScript in general, are beyond the capacity of our software engine. In several cases, partial implementation is much more feasible, such as with hyperlinking. Relative links (links that point towards local file locations) can be implemented in our software engine...
by simply loading a linked file, but URL links would require incorporating network functionality. Currently, clicking on the browser view selects the element at that click location. We will need to include an option to toggle between this element examination and ordinary browser interaction (once the latter is implemented).

Functionally, these components are a near imitation (however limited) to the “Inspect” feature of the Google Chrome browser (and its various equivalents). In Chrome’s case, this extends directly into a suite of web development tools, called Chrome DevTools [24]. With Chrome, one may “inspect” the currently loaded page, which opens an interface overlay (Figure 5.5). This overlay offers several panels to alternate between, though we will only emphasize those pertinent to our concerns - Elements, Console, and Sources.

![Figure 5.5: Google Chrome inspection, elements pane.](image)
The “Elements” panel identifies and collapses HTML elements, maintaining the XML hierarchy. Element selections made within this panel highlight rendered correspondents whenever appropriate. Our prototype performs similarly. However, in observation of our design principles, it disentangles the HTML hierarchy structure from the script itself, enabling independent examination of each (and of the otherwise implicit relationship between a script input and its DOM equivalent). Furthermore, it accommodates element selection in both rendered and pre-rendered views; it traverses HTML in both directions. Though our application of these features is not very different from this aspect of Chrome DevTools in function, it is distinguished by intent (and desired outcome). In our case, development-applicable features are means to an end, rather than the end itself. We emulate development tools in pursuit of a comprehensive environment for the sub-program, one that conveys processes implicit in its operation. Though unveiling these interior mechanics, we hope to enhance one’s ability to engage with and understand software systems, and to thereby elide specialization.

According to the Chrome DevTools documentation, the “Console” panel has two main functions [25], “Viewing diagnostic information about the page” and “Running JavaScript.” At the moment, we are concerned only with the first function, as it specifically provides another perspective on HTML execution. It identifies the source of JavaScript runtime errors (Figure 5.6). Upon achieving JavaScript implementation in our software, we could replicate this feature and augment it with drawn-line connections to the sources of error. This would necessarily involve an additional display field - our own “console” to provide runtime feedback.
Figure 5.6: Google Chrome inspection, console panel.

The “Sources” panel provides a file browsing interface, showing the server directory of a web page. It invokes the aspect of composition (rather than execution), and accordingly we shift our focus to that aspect, concluding our examination of Chrome DevTools. Like the console feature, a file directory display could supplement the plain text view, with additional lines pointing from linked files (such as images and relative page links). While the purview of the execution aspect is indefinitely expansive, composition (in this case) refers almost exclusively to the HTML source files loaded into this software scheme. The plain text view preserves white space so that source files appear exactly as they would in a text editor. Features such a file directory and text formatting are supplementary, and not of immediate concern.

The primary purpose of the plain text view is to showcase the composition, and to identify components therein. The HTML parsing routine we use is limited to line-by-line
specificity. It can only locate DOM nodes according to the first line they appear in (this technical limitation is detailed in section 6.3). Ultimately, we intend to enable the isolation of nodes within and across lines of code, but this will likely require a custom HTML parsing routine. Though the plain text view presents itself in the fashion of a text editor, it doesn’t currently function as one. Developing a text editor within our time frame was not feasible, and it isn’t necessary for demonstration. For prototype development, we can alternate between loaded files and perform changes to them externally. In a “finished product” state, this program would necessarily include a fully furnished text editor, incorporated within the plain text view.

The last computation aspect to consider is the “analytic”, denoting criteria applied in the parsing/interpretation of code. In our approximation of the “execute” and “composition” aspects, comparisons with existing software were readily available. Within the HTML platform scope, web browser imitation unambiguously evokes “execute”, reflecting exactly how HTML is typically engaged with. Our rendition of “composition” in the plain text view (and eventually a text editor) is imprecise, portraying only one among many mediums that produce HTML code (one need only consider Squarespace, Dreamweaver, and the many other varieties of software-assisted web development.) However, this fundamental state of composition is sufficient to demonstrate the relationship across the three aspects of “execute - composition - analytic”, as every other permutation of HTML composition follows the same analytic precepts. But unlike “execute” and “composition”, the “analytic” aspect does not lend itself to such a direct comparison. The plain text and browser views adopt the front-end features of ubiquitous and frequently used software. In this context, “analytic” denotes
the properties and characteristics of the HTML language. It identifies the back-end processing that renders the browser view, as well as the formal standards that compositions (displayed in the plain text view) presumably abide. It emphasizes the region of intersection between these two overlapping but inequivalent scopes, indicating our most immediate development priorities (at least with respect to this aspect).

“How Browsers Work: Behind the scenes of modern web browsers” (Garsiel & Irish, 2011) [26] is a web article published through Google’s “HTML5 Rocks” (now known instead as Web Fundamentals). It describes in detail the functional properties of contemporary web browsers, namely “Chrome, Internet explorer, Firefox, Safari and Opera,” as well as their mobile counterparts. The article identifies the “basic flow” of an HTML rendering engine (after acquiring the document from the networking layer), which consists of four phases (Figure 5.7). The rendering engine receives an HTML document and produces a rendered web page through this process.

![Figure 5.7: Rendering engine basic flow](image)

Our foremost concern (with regards to the analytic aspect) lies with the first phase, “Parsing HTML to construct the DOM tree,” as it involves the application of HTML rules to the document (or rather, the composition). The other three phases involve the graphical manifestation of parsed data and are more specifically relevant to execution.
than to composition. Similarly, the processes related to document submission are not as important as the parsing routine itself. Hence, the DOM view presents an intermediate state of the back-end process connecting the plain text view to the browser view. It is composed from the very same data structure (a virtual DOM tree) used to build the browser view’s render tree.

This DOM view, like the browser view, offers a minimal interpretation of this aspect. It identifies and distinguishes DOM nodes within the submitted document following HTML standards. In doing so, this view reveals properties and characteristics of HTML. Internally, this view can be enhanced with more robust tree navigation (using collapsible hierarchy), and with more detailed node data (such as by showing style attributes, default or otherwise). Externally, we could employ the previously mentioned console to expound upon markup standards in the manner of an HTML validator (a tool specifically intended to verify markup standards). While validation would certainly enhance our software’s utility, it is strictly supplementary (and therefore low priority), as a browser will usually execute HTML regardless of its validity.

5.3 Interior Circuit Analysis

We return now to the abstract basis of form-process, the interior circuit. It traverses the “perform - enunciation - determinative” triad. The exterior circuit (discussed just prior in 5.2) compelled direct comparisons to computation, identifying software features pertinent to each aspect and priorities for continued development. Here we contend with the deliberately nebulous notion of form-process, represented in this triadic
cycle (detailed in section 3.2). In this section, we consider whether our prototype replicates its blueprint; to determine if it conveys the diagram informing its construction.

In section 3.2, we define form-process in terms of three mutually constitutive aspects, perform(v.), enunciation(n.), and determinative(adj.). Performing is the enactment or realization of forms, enunciations are forms that instigate determinative effects, and “determinative” delimits conditions and properties of performing. However, we must emphasize that determinative effects can manifest in ways not specified by the aggregation of observable enunciations, and that enunciations may impact their own determinative state as well as others. Due to these properties, form-process enables an indefinite series of overlapping, concurrent, and intersecting circuits. Our prototype provides an analog to identify some of these inter and intra-circuit transitions.

Consider the enunciation represented within the plain text view. The determinative effect produced by this HTML document manifests primarily in the browser view. It does not exert an effect upon its own determinative properties. Instead, by being performed (in this instance, rendered), it generates an entirely new circuit in the form of a web page. On the web page, the possibilities for performing are influenced in part by the determinative effect of the source script. In other words, a web page acts (or is engaged) within parameters determined by its source script. The diagrams in Figures 5.8 and 5.9 trace this trajectory, following the form-process triadic model. The HTML document (B) (an enunciation) emerges as an instance of text encoding (A). The document’s compatibility with HTML standards (C) determines the outcome of page rendering (G). On figure 5.8, we condense an intermediary circuit (depicted in figure 5.8), as these aspects are not explicitly exhibited by the prototype, despite their implicit
role in the transition from C to G. The circuit through page rendering (G), page interaction (H), and determinative properties of the page (D-F) depicts form-process, to the extent it is observable within a web browser. Since the enunciation specifies page interaction, we exclude integrated development tools from consideration.

Figure 5.8: The HTML document’s (B) determinative effect on the resulting web page (D-F, G, H).

*Two caveats regarding this aspect:

First, this ought to be an adjective, but ‘Web page possible’ seems too awkward a neology. It must still be emphasized that this refers to the condition of possibility (with respect to the web page), rather than some set of “possibilities”. The former definition
does not compel one to confirm what may or may not constitute a “possibility”, and instead it’s applied as a general descriptor for anything that could happen on a web page. We are not interested in an itemized list of “possibilities”, or anything of the sort.

Second, as indicated by the designation D-F, this aspect is a condensed representation of another triadic circuit, one lying between C and G. As shown subsequently by Figure 5.8, the directional verb-noun-flow is maintained despite the truncation.

Figure 5.9: Condensed circuit determinative of web page rendering in detail (D-F. in figure 5.8)
The prototype enables one to explore a sub-program (web page) in conjunction with underlying processes that inform its function. Various supplemental features (like those discussed in section 5.2) can articulate implicit and/or adjacent circuits, such as the circuit shown in Figure 5.9. However, without in-circuit enunciation (with respect to sub-program composition and execution), this design is incomplete. In the finalized state of this platform, modifying the properties of the sub-program’s language must be an option, in addition to the editing and execution of sub-programs. In-circuit enunciations would modify determinative properties that directly influence subprogram performance, potentially changing how every other subprogram executes thereafter. In terms of computation, this means including the ability to submit compositions affecting the programming language, rather than the program itself. In practice, such a feature may not see much use, but it would provide potentially unprecedented opportunities to explore and experiment with the language itself. One could observe the effect of a parser modification throughout each of the various aspect exhibits, down to the program’s runtime.

This final stage of form-process implementation is currently well beyond our means to develop, at least in a specified time frame. However, a partial fulfillment for demonstration could be achieved far more quickly. The ability to create new HTML tag types with custom behaviors, while still a significant undertaking (requiring mappings to rendering engine operations), can be accommodated as an iterative upgrade to the current prototype.
CHAPTER 6

PROTOTYPE SPECIFICATION

This prototype program engine is written in C++ and relies primarily on the Open
Graphics Library (OpenGL) in conjunction with Simple DirectMedia Layer (SDL).
Interdependence would absolutely cripple a software project of this scope, and so we
have made every effort to construct functionally distinct, modular components.
Otherwise, experimenting with and incorporating new features would always carry the
risk of compromising the entire code base. Consequently, we treat these components as
general build tools. They should be viable as discrete APIs and replaceable in the event
better solutions are discovered.

The program is driven by a main module, whose primary responsibility is to
invoke other modules. Most calls to other modules occur in the program’s execution
loop, in response to user input. All the data used to generate the on-screen display is
stored and maintained in the render_manifest module. At the end of each pass
through the execution loop, the manifest is sent to the renderer module, whose sole
function is to filter through manifest data and submit attributes to the OpenGL rendering
context, creating the visual objects (text, rectangles, etc.) that appear on-screen. The
modification of manifest data is performed with the renderOps module, a class
consisting exclusively of static methods that either modify manifest objects or provide
information about them. The parsing and processing of the HTML file that will appear in the program interface is delegated to “data processing” modules.

This chapter is divided into sections corresponding roughly to the major module categories. Section 6.1 covers the libraries we used in this development. Section 6.2 details the graphics engine, and specifically the various modules associated with OpenGL rendering. Section 6.3 covers the data processing modules responsible for producing the different display modes of the input HTML file. In section 6.4 we describe the management of user input and interaction along with other ancillary components. At this time, the components discussed in 6.4 do not constitute discrete modules. We will isolate these into modules at a later time, one we’ve determined the full extent of functions and dependencies required to perform these tasks.

6.1 Libraries used in development

The prototype is written in C++11 and compiled on an Ubuntu system with the G++ compiler. The build configuration is managed by a standard Makefile. We made use of the following libraries.

**Simple DirectMedia Layer (SDL) - 2.0.2**

“Simple DirectMedia Layer is a cross-platform development library designed to provide low level access to audio, keyboard, mouse, joystick, and graphics hardware via OpenGL and Direct3D.” [27]

SDL is a versatile development library. In this case, it was used to generate the rendering context of OpenGL and to interpret mouse and keyboard input. It was chosen to ensure cross-platform viability for the foreseeable future.
Open Graphics Library (OpenGL) - 3.0 Mesa 10.1.3

“OpenGL® is the most widely adopted 2D and 3D graphics API in the industry, bringing thousands of applications to a wide variety of computer platforms. It is window-system and operating-system independent as well as network-transparent.” [28]

OpenGL provides an expansive graphics toolset that suit a variety of applications. It allows for nearly unlimited experimentation in graphical presentation.

Freetype - 11.0.5

“FreeType is a software font engine that is designed to be small, efficient, highly customizable, and portable while capable of producing high-quality output (glyph images). It can be used in graphics libraries, display servers, font conversion tools, text image generation tools, and many other products as well.” [29]

We used FreeType to gain access to the font libraries we need to be able to render text natively. It supports a variety of font formats, though for prototyping purposes we

TinyXml2 - 6.2.0

“TinyXML-2 is a simple, small, efficient, C++ XML parser that can be easily integrating into other programs.” [30]

TinyXML served as an interim XML parser, allowing us to reach a demonstrable prototype in a reasonable interval of time. With it, we could build our engine around a parsing routine that was certifiably complete. This gives us an unambiguous benchmark for producing our own parsing implementation in the future.
6.2 Graphics engine

This program’s graphics engine is facilitated by the renderer module. This module is initialized at the start of execution. During initialization, it creates the program window and attaches it to an OpenGL/SDL rendering context. This window is set to a fixed resolution of 1280x720. If this is successful, it then initializes two sub-modules, the text_renderer (for characters of alphanumeric text) and the rect2D_renderer (for rectangles and other simple geometry). In initialization, these sub-modules compile and link their respective shader programs. Generally, the renderer module makes all the joint OpenGL/SDL function calls, and the sub-modules (text_renderer and rect2D_renderer) make direct calls to OpenGL functions. If any stage of initialization fails, the program submits an error message to the console and terminates immediately.

The primary renderer module’s display() method is called at the end of every iteration of the main execution loop. This method receives the render manifest, which contains a mapping of visual objects (text, rectangles, and lines), as well the current screen dimensions and the font textures needed to render text. Each visual object is assigned to a “subRegion”, designating the rectangular area on-screen where those objects will appear. Objects positioned outside their subRegion boundary will be culled during rendering (using glScissor). The program uses 5 distinct subRegions, one for each of the three display fields outlined in 5.1 (see Figure 5.1 – plain text view, DOM tree view, browser view). These three areas are constrained by explicit boundaries; any visual objects outside the white area will not be rendered. The fourth subRegion contains overlay items, such as the connecting lines (which extend across the boundaries of all
three display fields). The fifth subRegion contains a “debugging” display (not included for demonstration) that is toggled on/off. The subRegions are rendered in the order of overlapping (so the two overlays are rendered last). The display method iterates through the data for each visual object and forwards it to the appropriate renderer (either for text or rectangles).

The renderOps module is the other major component of the graphics engine. It is a class consisting entirely of static methods. These methods are called whenever the render manifest needs to be modified, updated, or accessed. Some methods insert new visual objects into a designated subRegion, and others update attributes of those objects. This module writes to the render manifest, and the renderer module reads that manifest. There is no direct link or dependency between these two modules; their only connection is through the manifest.

In section 5.2 we mentioned that this build does not support alternative font faces, beyond free formats we can readily use. The FreeType library supports TrueType fonts (TTF) and several other options, but an investigation into font licensing at this stage of development seems premature. For the time being, this inhibits compliance with HTML font face standards. With HTML, one can modify text scale by fractional increments, and fractional font scales are even the default for certain element types (such as headings) [31]. This presents another text rendering predicament. Currently, our implementation produces a single texture for each continuous “block” of text, assembled from glyphs. These glyphs, or individual character textures, are drawn from an “atlas” containing a rasterized alphabet of glyphs. Specifically, the data to assemble a block’s texture consists of atlas coordinate sets, where each coordinate set points to a glyph’s location within the
corresponding atlas. Rasterized textures distort considerably with scaling, so we generate multiple atlases at different font sizes in advance. This is not a viable solution for fractional font scaling, as it would require the program to generate hundreds (if not more) of atlases to accommodate these size settings. We can either scale font textures within sufficiently narrow thresholds (for minimal texture distortion), or we can employ a method that temporarily generates appropriately scaled atlases. If these solutions prove insufficient, we will investigate the font-scaling implementations used by standard web browsers. We will and adapt our graphics engine accordingly.

6.3 Data processing

The modules that process an html file are invoked by the main module’s loadHTML() method. This method receives a filename input and returns a Boolean value indicating either success or failure. Once the file string is extracted, it is sent directly to the subRegion of the “plain text” view (in the render manifest) as a new text “block”. Since the plain text display represents the file as it would appear in a text editor, the line positions and whitespaces are maintained.

Afterwards, the input file is submitted to the domTree module. To save time, we employed the TinyXml2 parser to extract node data. This allowed us to reach a complete iteration very quickly, though it came at a cost. As mentioned in section 5.2, TinyXml2 can identify a node’s line number, but not its specific location within the source text. As such, the program can only point back towards the line in which a node appears. Precise
in-text node selection requires a more robust parsing solution, one that indexes each identifiable token (DOM node) and stores this information.

If the DOM tree creation method succeeds, a second domTree method generates the visual objects that represent the nodes of this tree. This method takes the render manifest and a font size setting (an integer value) as parameters. It navigates the virtual DOM tree and makes renderOps calls to insert new visual objects into the DOM view subRegion (of the manifest). The nodes are arranged in the DOM view in accordance with the DOM hierarchy. Currently, element nodes appear with white text on a solid blue background; text and comment nodes are shown with black text on a pale-yellow.

The renderTree module performs the last stage of data processing. Using the DOM tree data structure from domTree, it constructs an HTML “render tree” (as described in section 5.2). This structure is effectively a “sub tree”, containing only HTML element nodes. The tree construction method traverses the DOM tree and attaches a new renderNode whenever an element is encountered. The renderNode is a struct with the following fields: styleProperties (all the HTML style settings for the element), children (any elements contained within), content (a string of the element’s contents, other than its children), and an index that will be used to match the renderNode with its DOM node counterpart. As the DOM tree is navigated, HTML elements are identified and assigned default style properties accordingly. For example, “h1” elements have a specific default font size, and elements such as “head” and “title” have a “display” value of “none” [31]. In this iteration, we ignore in-line style settings and other attributes.
As with domTree, a second renderTree method is called to create the visual display once the tree data structure is complete. This method navigates the tree and uses the renderNode style settings to generate the visual object for each element. It uses renderOps calls to submit these to the browser view subRegion. Naturally, elements with a “display” value of “none” are ignored. Each object is attached to a map in the subRegion with its DOM tree index serving as the key. We use these keys to match elements in the browser view with their positions in the DOM tree view.

6.4 User input/interaction

The operations involving user input and system interaction are currently managed by methods in main. These features are largely experimental and may change radically in subsequent versions. We will develop them into discrete modules once we can narrow down and specify their scopes of function.

In the execution loop, the program retrieves user input with SDL_PollEvent(). If a mouse click occurs, it retrieves the mouse pointer location and determines whether it falls within the subRegion of either plain text, DOM tree, or browser views. If the mouse click occurs in one of these subRegions, that subRegion is set as the current selection, the border color of that region is changed, and the mouse position coordinates are submitted to a “selected” method for that subRegion. If the “up” or “down” arrow keys are pressed, the program adjusts vertical displacement of the currently selected subRegion, shifting its contents accordingly.

The three “selected” methods compare the submitted position coordinates against the render manifest data of subRegion in question (the secondary role of renderOps is
to retrieve the position data of manifest objects). If the click location falls upon a line of code, a DOM node, or an HTML element, “highlighting” rectangles are activated and assigned to rectangular dimensions around the selected object. The methods then search for corresponding objects in the other subRegions.

If the “selected” method for the plain text subRegion matches the click location to a line of code, it invokes the domTree to search for any nodes matching the line number. If a matching DOM node is found, it is also highlighted. The method then uses the indices of the matched DOM nodes to search for any matching element nodes in the render tree. These will also be highlighted. In the DOM tree selection method, if the mouse position matches a tree node, that node’s line number is used to find the segment of code it originates from, and the same highlighting occurs. The search for a render tree node and resultant behavior is identical. The browser view’s selection method iterates through the map of elements (of the browser view’s subRegion) to determine whether the mouse lands on one. If an element in the browser view matches the mouse position, its key is used to retrieve the DOM tree node, whose line number is then used to identify the line of code it originates from.

While these three “selected” methods are mostly similar, there is an asymmetry that prevents full standardization. Due to the limitations of our XML parser, selections made in the plain text view must be line by line at minimum. Otherwise there is no way to match DOM nodes with the code segments they originate from. However, a line of code will frequently contain multiple DOM nodes. Figure 6.1 demonstrates one such instance of “multiple selection”.

77
The “selected” methods are also responsible for drawing the lines that connect each selection to its correspondents in the other two displays. Since each display (the plain text, DOM tree, and browser views) can have its own independent vertical displacement (if the user elects to scroll any or all of them vertically), the coordinates for the connecting lines must be adjusted accordingly. This condition requires us to maintain an externally accessible displacement variable for each subRegion. The render_manifest and rect2d_render modules had to be modified to incorporate this external variable as a parameter. For lines connecting plain text code to DOM nodes,
the first endpoint is set to the right edge of the code segment’s bounding rectangle, with
the second falling on the left edge of DOM node. Lines from DOM node to browser
elements connect that same position (left edge of DOM node) to the middle of the top
edge of the browser element’s own bounding rectangle.
CHAPTER 7

RESULTS, ANALYSIS & CONSEQUENCES

The culmination of this thesis project elicits several key questions to consider.

Our concluding discussions are divided into the following sections:

• 7.1 - In terms of software development, what has been tangibly accomplished thus far? What additional functionality can we incorporate in the short term? What else would be required of a complete product?

• 7.2 - What are the potential applications of this software, taking the possibilities of continued development into consideration?

• 7.3 - Does this implementation of our design principles approach a compelling and feasible response to the divarication of computer software? What criteria can we apply to determine this? Is continued development warranted, or is reassessment in order?

7.1 Development status, short and long-term

The prototype we present along here offers a modest, “proof-of-concept” circuit. It displays and articulates connections between an HTML script in plain text form, a DOM tree representation of that script, and the rendered page resulting from that same script. These are all contained in independent displays that appear simultaneously in the program’s on-screen interface. Users may select items in any of these three displays (either lines of plain text code, DOM tree nodes, or HTML elements). The program
highlights the selected item along with its correspondents in the other two displays (the two that aren’t currently selected) and draw lines connecting them. With this platform, an HTML file can be navigated in three different, complementary facets.

We designate two general feature categories for this prototype: the overall program engine/interface, and HTML support. As is likely evident by observation, we have invested far more development effort in the former category. We endeavored to produce a robust program engine and sought for demonstrative purposes only the minimum degree of HTML support. This approach was somewhat necessary for a sufficiently malleable interface (to showcase the program’s primary justification, its distinct interface features), but it was also deliberate. A more robust engine will leave us better equipped to continue the project in any circumstance that compels us to modify or outright abandon the HTML implementation. We partition our discussion of what remains to be done (in both the short and long-term future of development) according to these two categories.

Much can be done to improve HTML support in the short term. We can expand the set of recognized tags to include all standard HTML elements, along with their default style properties (as long our program engine can render them). We can also extract element attributes (using TinyXML) and display them in the DOM tree representation (or if space becomes a concern, they can be listed in an external dialog window). We can also implement certain additional features that don’t rely on network functions or JavaScript, such as relative hyperlinking and displaying images. Once support for the standard HTML elements and default styles is complete, we can implement CSS support (though a full implementation of either would probably extend
well beyond a short-term development span). In the long term there is JavaScript to consider, which will require a JavaScript engine and an entirely new set of interface features to articulate its behavior. It is too early for us to speculate further on a JavaScript implementation. The program will also need a custom HTML parsing routine (instead of TinyXML) that records the in-line starting and ending positions of tags, allowing more precise code-to-node matching.

In the short term, the code base needs refinement and reorganization. This appears to be a regularly recurring necessity, as we have updated program modules many times over with the incorporation of new features. Specifically, we will probably need a distinct “driver” module. This module will be responsible for receiving/processing user input, pushing display data to the “render manifest”, and calling the “renderer” module to render the manifest data (and/or delegating these tasks to submodules). This should eliminate platform-specific code from the “main” module, and significantly improve code legibility. A console can be added to the bottom-left of the program interface (though this isn’t imperative for static HTML demonstrations). There are several more quality-of-life features we could add in the short term, such as an ability to “suspend” selections (so that clicking elsewhere doesn’t erase the selection). For the three display facets (plain text view, DOM view, browser view), we can add better scrolling/panning options, zooming, in-frame reorientation, and an ability to scale/reorient them. These improvements don’t demand any significant changes to the engine. Potential improvements for the line-drawn connections could include changing the color of an individual line segment on mouse over, and drawing arrows pointing towards out-of-frame endpoints (since ordinary lines can’t be drawn if an endpoint is out-of-frame).
Long term development towards a complete product must integrate several additional engine and interface features. The “plain text view” will need to be replaced with a fully realized text editor (along with an ability to either update the other displays with recent changes, or to revert to the previously loaded state). This view should also offer multiple selection options (allowing code selection by tag, line, etc.) Each of the display views will need additional interaction modes (other than selection) that users can toggle: an “edit” mode for the plain text view, a “page interaction” mode for the browser view (functioning as an ordinary web browser), and alternative structures (other than the tree hierarchy) for the DOM view. Various interface customization options need to be made available (screen resolution, window proportions, display facet proportions, etc.), along with complementary updates to the pertinent program functions.

### 7.2 Possible applications

The prototype described here is only in the earliest stages of development, but as a proof of concept it demonstrates many of the qualities we discussed in our pursuit of interfacing iconicity. In Chapter 5 (specifically sections 5.2 and 5.3) we were able to rely on this program as a point of reference, to ground the abstractions of Chapter 3 with a relatively concrete analog. In the previous section, we noted several engine improvements that would augment this demonstrative capacity in the short-term. Interface improvements to allow zooming/panning in the individual display facets (the plain text, DOM, and browser views) would enhance the platform’s ability to display large-scale scripts and facilitate their analysis.
With improvements to HTML support, this software could become an effective educational tool, one that disambiguates HTML and makes syntax errors easier to find and address. However, pursuing this possibility would compel a significant shift in development priorities. Software design would be heavily influenced by the needs and expectations of its prospective user base. We would focus on producing an iteration that could be tested by these prospective users for feedback. An emphasis on educational utility would significantly narrow our project goals and thereby expedite development significantly. We would temporarily abandon the full context for this platform, producing a more immediately “useful” program instead.

When (and if) the long-term software goals are complete, it will come into its own as an HTML development platform. With the inclusion of JavaScript support, the platform engine should be sophisticated enough to support other programming languages (presumably, a new “parser” module would be the only major requirement). The software would essentially resemble an IDE, but one built with a distinct design approach. At this state, it would be appropriate to consider online functionality. We could enable multiple users to edit a document and to observe its editing history, and thereby to articulate another dimension of program composition.

7.3 Project continuation and final remarks

After contemplating potential outcomes of this software project, we must now confirm whether those outcomes are worth pursuing. Ultimately, we mean to establish principles of software design in opposition to the effects of divarication. We defined divarication in computer software as a recurring divergence into discrete modes of
operation. We argued that this divarication results from successive interface thresholds, that as each system interface provides a condensed point of access, it necessarily conceals the internal mechanics of that system.

A presumptive solution for divarication must exhibit an ability to persist through multiple interface iterations; otherwise, it will very quickly cease to be relevant. Any insight our method conveys must persist beyond the specific environments it occupies, otherwise it is no different than any other specialized software utility. Hence, we’ve attempted to imbue design principles in an iconic rendition, so that third parties might adopt them of their own volition. Our software platform doesn’t necessarily need to reach a degree of completion beyond the capacity to communicate and spread these principles (depicted diagrammatically in Chapter 3). We cannot guarantee that we will reach this threshold, since it relies on circumstances beyond our control. However, the odds of success will likely to improve as the platform achieves greater degrees of sophistication.

While proliferation relies at least partially on the platform’s quality (as a comprehensive piece of software), it is an effort in futility if our design principles are not explicitly expressed through the platform’s operations. In this respect, the prototype remains insufficient. It doesn’t differ significantly in function from a typical integrated development environment (IDE). It articulates the processual link between an HTML script, the DOM interpretation of that script, and its rendered HTML form. However, the prototype offers only one venue for composition, the HTML script. In section 5.3, we discussed the absence of in-circuit compositions. A full circuit traversal must overlap with the aspect of origin. The prototype’s circuit originates with a composition in the “plain text view” and concludes with executing in the “browser view”. It touches all three
aspects, but only once each. In this case, in-circuit composition would involve interactions with the HTML language. To make this equivalent to our established facet for compositions (the plain text view), we would require an additional display, featuring an HTML parser in text form that can also be edited, with those edits directly effecting how the platform parses HTML. Our concept for implementing this is subject to change drastically. We can consider many alternatives (custom tags, editing default tag attributes, etc.), or we may find that the HTML platform isn’t suitable for this. We want to build a platform that articulates the translation from code to program, (as is currently done with HTML), as well as the translation from language to code (by providing a venue to modify a language and to observe the consequences). If we cannot extend this out of our current prototype, we will still make use of the engine code we accumulate along the way. In either case, we won’t know which direction to take until the project advances further.

We have discussed two requisite conditions for a valid solution: persistence across iterations and a facet for in-circuit composition (to demonstrate a full circuit “cycle” in a succession of compositions). A third condition lies in the following question: “Does this platform accurately reflect the internal mechanics that comprise it?” If we assume that our design scheme is iconic with respect to self-transmission (such that observing it is sufficient to recreate it, so that it can proliferate), we must yet confirm that this conveyance induces the desired effect. In other words, does our representative platform convey software mechanisms to an extent that facilitates their replication? As instigators, this is not a question we can answer for ourselves, not entirely. We have assumed that user-adjacent alternatives to conventional software interaction can disrupt repetitive instances of stratification. Our software system is designed with flexible role
boundaries, in hopes that it won’t form its own specialized constraints. Without testing, we can’t be certain of any claims. However, we can assume that if the platform isn’t effective in conveying the properties of HTML, it can’t possibly achieve our greater goals. If access to in-circuit composition doesn’t augment its instructional capacity, we can make the same assumption. We may test several different implementations before reaching an answer, though each test should indicate which techniques are likely or unlikely to succeed.
REFERENCES


