Deconstructing code that works

Peter Wildman

A thesis in fulfilment of the requirements for the degree of
Masters of Fine Arts
UNSW Art and Design
Faculty of Fine Art

Supervisor Dr Tim Gregory

26th of February 2018

PLEASE TYPE

THE UNIVERSITY OF NEW SOUTH WALES
Thesis/Dissertation Sheet

Surname or Family name: Wildman

First name: Peter Other name/s: Charles

Abbreviation for degree as given in the University calendar: MFA

School: Art and Design Faculty: Fine Art

Title: Deconstructing code that works.

Abstract 350 words maximum: (PLEASE TYPE)

'Deconstructing Code that Works' is a practice led research project that frames theoretical critique as practice in a gallery setting. The work uses a deconstructive methodology derived from Jacques Derrida's practice of 'sous rature' to perform critique upon a particular moment in the historical formation of the field of 'codework'. The term codework was established in 2001 and attempted to describe literary works that were developed from or included elements of computer code. The taxonomy of this field, formalised by Alan Sondheim, was contested by John Cayley on the basis that non-executable work should not be included into the field as code or executable text. By bringing the thesis of this research into the gallery space the performer uses the theoretical methodology as a practical methodology to produce critical artefacts. The thesis is placed under erasure within a system that produces computational exceptions or non-executables as work. These exceptional texts are caught and handled within the gallery as a mode of production and are transformed into physical objects to be thrown into the space. The resulting exceptional texts are developed from this codework divide yet they can no longer be read along these terms

FOR OFFICE USE ONLY Date of completion of requirements for Award:

ORIGINALITY STATEMENT

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.'

Signed ……..

Date ……..

Acknowledgements

As the author of this thesis I would like to acknowledge that it has been done with the support and input of many others.

I acknowledge and pay respect to the traditional owners of the land, past, present and future, on which this thesis has been written; the Gadigal people of the Darug Nation.

I acknowledge the extensive support of my supervisor Dr Tim Gregory as it is difficult to determine the elements of this thesis that were not intertwined with his input as we discussed it throughout the creative process.

I acknowledge the support of others from the University of NSW who have engaged with me in feedback on the campus and also provided assistance and resources in developing this thesis.

I acknowledge the support of those outside of this university system who also contributed by providing conversations that undoubtedly guided the ideas and arguments present in this thesis.

Table of Contents

Introduction and Methodology

0.1	Introduction	4
0.2	Methodology	7

Chapter 1

1.1	Delimiting a field of codework	12

Chapter 2

2.1	Making exceptions	18
2.2	How exceptional texts work	23
2.3	Throwing exceptional messages	24
2.4	Maintaining coded structures	32

Conclusion

3.1	Conclusion	40

Chapter 3

3.1	Documentation of exhibition	46

INTRODUCTION

The purpose of this thesis is to return to the historical context of Alan Sondheim's formalisation of a general field called codework and to trace a subsequent binary discourse through John Cayley, Rita Raley and N. Katherine Hayles that addresses the separation of the field of codework between executable/non-executable forms. This thesis is a practice based research project that deconstructs theoretical critique of an historical discourse and positions this critique as a contemporary art practice by implementing a methodology based upon Jacques Derrida's post-structural practice of sous rature as it has been translated by Gayatri Chakravorty Spivak as placing under erasure. This methodology will be further described in the following section of this thesis. Introducing the topics of this thesis is nuanced and complex, as the method of deconstructing theory/practice has an effect of erasing a discernible origin by collapsing the reading/writing of this thesis. Whilst reading this thesis you are situated within an art gallery setting. These words are being read upon the walls of this gallery. These words have been previously written outside of this gallery, using a method of deconstruction, then printed on paper, to be further deconstructed in the gallery through a system developed from the conclusions within the written words. Whilst reading this thesis you are reading chapter three of this work, which is offered as a deconstruction of these previously written words in chapters one and two. Chapter three has been previously enacted within this gallery space leaving these artefacts as a trace. Upon entering the gallery you saw this deconstructive system being performed in a video documentation. This introductory video documentation shows the process of this deconstructive system being applied to the abstract of this thesis. This documentation is offered as another trace of the deconstructive system that was previously enacted to produce the creative outcome of this research.

The following is an attempt to describe chapters one, two and three in chronological order. Chapter one of this paper is an introduction to the aforementioned discourse of codework and asks what power dynamics are at play when a general field is used to

redefine works that already exist under their authors own terms. I claim that a general field is one that attempts to gather a multitude of works into its definition and in doing so places works into the field despite differences of terms that do not fit or are not addressed in the field's taxonomy. To establish this line of inquiry I present Alan Sondheim's formation of the codework taxonomy, as a theoretical act of reauthoring works to form codework as a general field. I claim that this is a process of reauthoring because the subsequent discourse of these included works as codework focuses on the definition and validity of these works as codework, rather than addressing the terms of reference offered by the authors whose work were being discussed. Sondheim's formation of the field in this thesis is followed by John Cayley's alternate terms of reference for codework, based upon the binary distinction of executable/non-executable code, which divided and attempted to exclude works that could not compile that were included within Sondheim's original taxonomy. Rita Raley's inclusion in the discourse covers her attempt to move beyond the binary discourse offered by Cayley as a means to unify and expand the field. The republishing of N. Katherine Hayles' survey of the field of electronic literature is also addressed. In this survey Hayles applies a colonial context to the codework discourse by referring to executable codework as a pure machine language and non-executable codework as a broken creole. The republication of Hayles' survey in 2017 offers this paper the impetus to readdress the executable/non-executable terms that were historically formed in this field. It acts as an important element, situating this historical formalisation of codework and this research as contemporary issues for codework and electronic literature more generally.

Chapter two mentions some examples of programming languages and concepts to show the multitude of approaches that can be taken in the formation and use of computer code as a programming language structure. This chapter begins by taking the position that all programming languages are particularly constructed and hence there is no one structure that can describe their formation. The executable/non-executable binary discourse is then deconstructed by taking a particular programming language called Processing and discussing the process that takes place when non-executable text is written and executed

within this programming environment. What occurs here is a process of exception handling, which is an executed process for handling exceptional text that is not written according to the Processing language programming structure. The processes in place for handling these exceptional texts are then deconstructed in this thesis to show how non-executable or exceptional text is executed as a corrective message. This corrective message is used to maintain the executability of the language structure. This paper concludes that the non-executable in the Processing language is always executed, as the maintenance of the language structure depends on the programmer understanding the executable properties of the language.

Chapter three is situated within this gallery setting and framed as a creative theoretical practice. The artefacts of this theoretical practice are produced by procedurally placing the words of this paper under erasure. This procedure forms this as a deconstructive practice and as a means of throwing exceptional messages. Conclusions drawn from theoretical critique in chapter two inform the materials and methods that are used to create this deconstructive system. These materials are presented as a particular, idiosyncratic deconstructive critique of the general field of codework. The artefacts that form this research are presented as creative outcomes that work as critique of the binary divide. This practice acts upon the field of codework by deconstructing the terms used to form it, and places these terms sous rature, offering the practice as a creative work of critical difference that is from this field but can no longer be read from the binary division from where it has been deconstructed.

METHODOLOGY

The following section is the establishment of a methodology to be applied in this practice based research project. This methodology is derived from the post-structural deconstructive practice of sous rature developed by Jacques Derrida as he used it in philosophical discourse. More specifically this deconstructive practice will be an interpretation and implementation of Gayatri Chakravorty Spivak's reading of sous rature as under erasure found in Spivak's preface to her translation of Derrida's *Of Grammatology 40th Anniversary Edition*. For the purpose of this practice based research, the method of sous rature will be used to deconstruct the discourse outlined in the introduction to this thesis, which is further described in chapter one. This deconstruction will take place as chapter two of this thesis, where a particular programming language called Processing is used as a case study for this method. Chapter two also introduces other approaches to programming languages and concepts that support the findings of this case study. Chapter three is presented as a further deconstruction of chapters one and two. Chapter three places chapters one and two under erasure materially and within a gallery setting, where all three chapters are read as an idiosyncratic, practice based research outcome. The process of deconstruction is framed as a performative critical act, as a practice based theory, yet this act can only be seen through the artefacts that are left as the trace from this performance.

Spivak notes that Derrida collapses binary constructions because he reads them as "a violent hierarchy" of control where one term "holds the superior position." Derrida poses that "… one could reconsider all the pairs of opposites… not in order to see opposition erase itself but to see the announcement of a necessity such that one of the terms appears as the differance of the other, as the other different and differed in the economy of the same…" (Derrida 2006, xlviii) The executable/non-executable definition of computer code is focused on in this thesis. As will be discussed further in chapter one, the position of the executable was offered by John Cayley as the dominant position that he believed should be

applied to the codework taxonomy and further discourse was either aimed at reinforcing the importance of the non-executable within the field, as can be seen through Sondheim and Raley, or to further describe the qualities of the executable as opposed to the non-executable, as is seen through Hayles. The aim of placing codework's executable/non-executable binary position under erasure in this thesis is not to reinforce or maintain either of these binary constructions, but rather to announce the construction of these binaries as a system that maintains the hierarchy of the coded structure. As Spivak quotes Derrida "… the issue is… to deconstruct the metaphysical and rhetorical schema which are at work in [a text], not in order to reject and discard them, but to reinscribe them otherwise." (Derrida 2006, xcviii) This thesis is offered as an artwork that reinscribes the historical discourse of executable/non-executable codework as a contemporary reading that can no longer be read with these binary terms.

The following is an attempt to outline the deconstructive method applied in the particular case of this thesis. At first, Spivak's preface to Derrida's methodology has been taken as the primary source informing this thesis' particular method. Spivak quotes Derrida at length in her preface and the following quote was the starting point for this thesis.

"If in the process of deciphering a text in the traditional way we come across a word that seems to harbor an unresolvable contradiction, and by virtue of being *one* word is made sometimes to work in one way and sometimes in another and thus is made to point away from the absence of a unified meaning, we shall catch at that word. If a metaphor seems to suppress its implications, we shall catch at that metaphor. We shall follow its adventures through the text and see the text coming undone as a structure of concealment, revealing its self-transgression, its undecidability… I am not speaking simply of locating a moment of ambiguity or irony ultimately incorporated into the text's system of unified meaning but rather a moment that genuinely threatens to collapse that system." (Derrida 2006, xcviii)

Chapter one of this thesis is my own reading of the discourse of the executable/non-executable divide of codework as it was constructed by Sondheim's formalisation, through taxonomy, and further debated through Cayley, Raley and Hayles. This chapter is the text, through which, the adventures of deconstruction have been followed. Chapter two uses a point from chapter one that seems to threaten to collapse the executable/non-executable unification in the codework discourse and unravels this to place it under erasure. This particular point is discussed in more detail in chapter two, but it can be named simply as the exception or exception handling in the Processing language structure. The exceptional text, or non-executable, is then interrogated and revealed as a construct of the executable structure. What will be shown is how the system of the executable, in the particular language called Processing, designs the non-executable as an exception that can be handled, caught and thrown back to the author of the exceptional text as an exceptional message. It is here that the seemingly separated binary of the executable/non-executable reveals itself as a constructed metaphor that conceals the structure as one always maintaining its own executability.

The conclusions from this deconstruction in chapter two are then taken as a point of departure for further deconstruction. This further deconstruction can be read as a material system of deconstruction that is chapter three. In an interview on the *Culture Machine Live* podcast series Federica Frabetti argues for a material understanding of deconstructive methodologies that do not only comprise of what can typically be read as literal written text. She states "we can deploy a deconstructive methodology and at the same time firmly believe in the materiality of matter, in the materiality of the word, of the economy, of technology and of course the materiality of software." (Frabetti 2015) The system that produces the work, the trace of deconstruction in this gallery, is one that embraces the multifaceted material readings of computer programming language structures. In describing the deconstructive methodology that she applied to her book titled *Software Theory: A cultural and Philosophical Study* Frabetti states she attempts to "…wrench deconstruction out of the linguistic turn in the humanities and to bring it back to its roots, which I believe

take into consideration materiality, starting with the materiality of language and what Derrida calls the trace." (Frabetti 2015) The deconstructive system to be described below is an attempt to wrench the executable/non-executable binary discourse out of the codework discourse through a material deconstruction and place it back into a discourse of multiplicity and difference.

Chapter three is made through a system that places the material of this thesis under erasure. This system resembles the theoretical system of deconstruction that was also used in chapter two. Spivak reads sous rature as placing under erasure and states that it "is to write a word, cross it out, and then print both word and deletion. (Since the word is inaccurate, it is crossed out. Since it is necessary, it remains legible.)" (Derrida 2006, xxxii) This crossing out and leaving legible is applied in a number of ways and is present in this gallery space to be read accordingly. The point of origin of this system is difficult to find, but for the purpose of providing a concise way of describing this system, I will begin with the more literal crossing out that can be read directly here with these words upon this gallery wall.

Each word of this thesis has been crossed out with a red marker. This crossing out has occurred over time and according to a piece of code that has been written in pencil on a number of pieces of paper. This code is a piece of executable code poetry, with the executed outcome being a random number generated between the numbers five and seven. The number is used to cross out a sequence of words in each iteration of this deconstructive practice. This code that appears to be executable is never executed within the computer as executable code. Rather it is read aloud as a poem as it is erased with the red part of an eraser that is shaped in the form of a tool. The erasings are handled like executable code, carefully carried and thrown onto a black rectangle shaped piece of fabric placed upon the floor. The piece of paper is then placed into the printer. The written thesis is then crossed through with a red marker according to the number generated by the code poem. The words

that are crossed out are typed into the Processing Development Environment (PDE) and are executed. The written words generate an exceptional message that is thrown to the error console of the PDE. This message is written into another window of the PDE and both the words from this thesis and the exceptional message are edited to form two lines of approximately equal length. The words of the thesis and the exceptional message are then digitally cut out from the PDE and are sent to the printer via another computer program that is written in order to compile these lines in a particular way. This compilation is one line printed on top of the other. This compilation is printed upon the piece of paper that was previously placed in the printer. This printed piece of paper is then taken from the printer, scrunched into a ball and thrown against two walls in the corner of the gallery. This process is then repeated until all of the words of this thesis have been deconstructed in the gallery. Each page of the written thesis is pinned to the gallery wall once all words on that page have been crossed out. This deconstructive process, as it has been applied to the abstract of this thesis, is placed outside of the gallery, along with a video documentation of this process, as an introduction to the deconstructive system. The performance of this deconstructive system as it was applied to the introduction, methodology, chapter one, chapter two and the conclusion remains only accessible through the artefacts in this gallery that are presented as a trace, a creative outcome of this research methodology.

CHAPTER 1

Delimiting a Field of Codework

In 2001 Alan Sondheim published an article in the American Book Review which sought to delimit a field he termed codework. In this introduction to codework he proposed a taxonomy placing authors and their works into three separate categories to unify a field within the limits of one terminology. These categories describe an "interplay" between a "surface language" and a "submerged code" where work is created either by making submerged code "emergent", referring to submerged code on the surface, or by creating a combination of both code on the surface and language in the depth. (Sondheim 2001) In a discussion that took place on the *Nettime Mailing* List in February 2004 Sondheim stated that codework for Sondheim would tend "toward inclusivity," an approach that could be seen to benefit a collective and encourage an emergent community. (Sondheim 2004, par 53) Sondheim's taxonomy uses works that had already existed as a way of constructing the terms of this field and he lists the creators of these works in the taxonomy's defining characteristics. This is troublesome as these characteristics were already present in the included works, but were termed by the creators themselves before Sondheim decided to form his taxonomy. One example of this is the inclusion of m[ez]ang.elle which is work termed and created by Mez Breeze. Breeze had terms for m[ez]ang.elle and through Sondheim's taxonomy/authority found its place within the field under his terms. (Breeze 2001) Another example of this was the work of Talan Memmott who termed rich.lit and this, like m[ez]ang.elle, found its place within these new terms of codework. (Raley 2002) Whilst I believe it is reasonable to accept that work can be viewed from differing perspectives and a person can relate qualities between different works, it should be maintained that these critiques are from one author. Sondheim attempted to formalise a field that reauthored work rather than offering a critique of these works according to Sondheim's terms of understanding, obfuscating an idiosyncratic point of view and

taxonomic system of reference by assigning a perspective to the term codework that was to become a general field. Discussions about the included works of codework were then centred around a taxonomy that belonged to the term codework and was no longer viewed as an idiosyncratic critique of Alan Sondheim. This became troublesome as debates about codework in relation to the included works were had without the inclusion of the authors on the terms of reference they had created before Sondheim implemented his terms and taxonomy upon them. In turn, Sondheim positioned himself as an authority on these works as codework and these works were then being spoken about according to their classification within the field and toward Sondheim's understanding.

One example of this is seen in discourse between Alan Sondheim and John Cayley that focused on the inclusion of m[ez]ang.elle into the field of codework. John Cayley, an electronic literature author whose work termed programmable poetry prior to it being termed as codework, did not agree that work which Cayley believed was not functionally executable within the computer as a computer program should be included into the field. Cayley sought to critique Sondheim's categorisation and inclusion of work that "references… computer language and engagement" (Sondheim 2001, par 5) as codework. It was not enough for a work to reference computer language and engagement, for Cayley codework must instead act in and engage the computer as code, and address the machine in this way - it must be executable. For Cayley non-executable works were considered interface text, or pseudo code, and in an article Cayley introduces as an "argument against the collapse of categories" (Cayley 2002, par 1) titled *The Code is not the Text (unless it is the Text)* he states that "we would not try to compile the code in the interface texts of Memmott, Mez or Sondheim." (Cayley 2002, par 25) Here Cayley refers to the process of compilation that takes place within the computer when one code translates into another code and so on until the binary structure of machine code is reached. One recognised early conceiver and developer of this process, Grace Hopper, defined this as an interpretation process between the "main program" and a "subroutine" that is written in a "pseudo-code" to allow the execution of the pseudo code within the main program of the computer.

(Hopper 1953, 2) Although Cayley states while he would not compile these interface texts into the computer as code it is my claim that he does compile/interpret the non-executable works in his critique as a means to exclude them from the field of codework. It is not my intention here to place a computational framework of computer programming onto the field of literary critique but rather to introduce a question of interpretation of computational critique that is used in literary critique. Cayley's perspective that the works of these authors would not compile is a literary critique of computational compilation theory and one that I disagree with. It is my opinion that literal text that does not compile as it is written within the computer is most often compiled or executed as computer code, as exceptional texts. This position will be further addressed in chapter two of this paper. It is my intention to show that executability of computer code within a field of literary critique is a topic that can have multiple perspectives. Cayley takes one perspective to judge the compilability of these works by stating that he would not attempt to compile them and then subsequently (depending on this perspective of non-executable code) interprets these works or compiles and handles them as exceptions to be excluded from the field.

Alan Sondheim disagreed with Cayley's reading of the terms of code and compilation in regard to the executable/non-executable divide he placed upon codework. In a discussion between Sondheim and Cayley on the *Nettime Mailing List* Cayley raised questions on the "properties and methods of code" to be considered in codework and Sondheim replied by highlighting that these questions were Cayley's approach and not Sondheim's. (Cayley 2004, par 17) This response shows that the terms of code in reference to the field of codework were being considered idiosyncratically. It is my critique that this comment reveals the importance of framing codework, not as a general field, but rather as the perspective of an individual who attempts to gather other works into a discourse based upon the terms defined by them. By pivoting on this critique I argue that both Sondheim and Cayley were working toward the formation of a general field of understanding that separates and conceals their subjective positions through the general term codework and their disagreements on the terms of reference were therefore about whose terms of

reference the field will be formed upon. A field that was named by Sondheim (according to his terms of reference) to bring together a multitude of already established and differing terms on behalf of the creators of these works was now split into two fields based upon the executable/non-executable binary position of Cayley. In lieu of this contention, the discourse was aimed at resolving and unifying these binary positions within the field of codework.

In an article published in 2002 titled *Interferences: [Net.Writing] and the Practice of Codework* Rita Raley also contributed to this debate. Raley framed this executable/non-executable point of contention between Sondheim and Cayley's terms of reference as a first wave of codework and suggested that other questions needed to be raised as the field approached a "second wave of critical discourse". (Raley 2002, par 8) Raley suggested a resituating of the codework discourse to move on from the executable/non-executable divide that can be seen at this point in time as one of the issues destabilising the foundations of a unified general field. She noted that the "descriptive foundation still needs to be articulated as we move on to consider some of the more important questions and issues raised by the practice." (Raley 2002, par 8) One of these more important issues that she lists are the "relations between natural and programming languages," (Raley 2002, par 8) a move that I would argue here is related to the occupation of the executable/non-executable binary and one that can be seen at play in N. Katherine Hayles' survey of codework as it later became termed within the field of electronic literature. But Raley's position on the necessity for codework to compile as code according to Cayley's perspective of executability and compilation was to disagree with the necessity of Cayley's terms and in turn place works like m[ez]ang.elle back within the field of codework. Raley attempted to reunify the field, or more precisely Sondheim and Cayley's divided terms of reference, by incorporating the binary divide of executable/non-executable into the qualities of the field itself. She states that "codework tries on the whole to move beyond this schism - the code and its 'work' or operation - to make something new. It relies on this schism in order to produce its effects, but then there is a mixing, an interfusion, and

something other emerges." (Raley 2002, par 31) Yet, this reunification of codework has not taken hold as we can still see this schism being referenced fourteen years on.

In 2016 Routledge Press published a book titled *Doing Digital Humanities: Practice, Training, Research*, in which the editors republished an article titled *Electronic Literature: What is it?* written by N. Katherine Hayles. Version 1.0 of Hayles' article published in 2007 was written as "the first systematic attempt to survey and summarise the fast-changing field of electronic literature." (Hayles 2007, par 1) In this article Hayles segments the field of electronic literature into separate genres, one being the field of codework, which she describes as "a linguistic practice in which English (or some other natural language) is hybridized with programming expressions to create a creole evocative for human readers… in its purest form is machine-readable and executable… more typical are creoles using "broken code," code that cannot actually be executed but that uses programming punctuation and expressions to evoke connotations appropriate to the linguistic signifiers." (Hayles 2007, par 30) Hayles includes a note at the end of this description that points to John Cayley's 2002 article *The Code is not the Text (unless it is the Text)*. In this article Cayley critiques Hayles' use of the terms creole and pidgin as a "significant misdirection" (Cayley 2002, par 26) because the "complexities of address should not be bracketed within a would-be creolized language of the new media utopia." (Cayley 2002, par 8) The new media utopia that Cayley is referencing here draws back to his position that executable code is addressing or executing computational processes, while codework that does not do this, non-executable computer code, is based in a context where the code is situated in a utopian setting where for example an "invasion of an empire of machinic colonizers whose demands of trade and interaction require the creation of a pidgin by economically and linguistically disempowered human users." (Cayley 2002, par 26) Whilst Cayley is critiquing Hayles' misdirection of the terms of non-executable codework within colonial terms of language creation and use, he is also reasserting his position on the importance of the computational executability of code within the field of codework.

Cayley uses the utopian setting to state that code which is not addressed to the compilation and execution processes of the machine are left for the human only, in other words natural readings of language and hence should not be considered as code within the computer according to a computational theory of compilation. This position, albeit shown in a different context in response to Hayles, was exceedingly similar to the one held in the formation of Sondheim's field of codework. What can be seen here, despite Hayles' reframing of codework into a colonial context, is the continuation of the separation of codework as two fields divided in the discourse between Sondheim and Cayley in 2001. Hayles' contribution to this discourse in 2007 was to take this executable/non-executable divide, place a colonial rhetoric upon it and in doing so reassert an executable/non-executable binary dominance and division of power that Raley had previously attempted to resolve. Furthermore, Hayles frames the non-executable, or the creole text, as a broken form. This colonial context and perspective of the executable/pure non-executable/broken will be further addressed in more depth in chapter two. And it is from this position of the executable/non-executable divide that this thesis will attempt to deconstruct the field of codework to reveal the arbitrary constructions of the binary as a means to control. What will be produced from this process will be situated outside of these binary terms as these terms will have been placed under erasure and thrown into the gallery as exceptional text.

CHAPTER 2

In the following chapter I will be using the deconstructive methodology previously outlined to place the executable/non-executable binary discourse of codework under erasure. What this section seeks to show is how the executable/non-executable distinction is a constructed principle of a computer programming language used to maintain an idiosyncratic language structure. It is my claim that there is no general structure that defines a code, but rather there is the ability to form structures called codes and ways of adapting and maintaining these structures. Rather than accepting code as a general construction and arguing for the inclusion/exclusion of codework based upon it, I seek to claim that the non-executable is designed into the structure of a language and is furthermore used as a means of maintaining a language's authorship. By deconstructing the executable/non-executable binary position using a computer programming language environment I hope to offer a new way of considering these environments to produce codework artefacts that cannot fit within authoritative or correctional executable/non-executable binary terms. These new approaches will take form within the gallery setting as chapter three of this thesis and be presented as a production of contemporary codework. These artefacts will be made and presented as both/neither executable/non-executable exceptionally coded texts.

Making Exceptions

Within John Cayley's critique of Alan Sondheim's field of codework he raises questions on the terms of reference used to define code in the codework taxonomy. There are a number of points of critique in Cayley's argument against Sondheim's taxonomy for codework, including questions of whether code addresses a machine or human; whether a code can be compiled; whether a text is surface language as opposed to a programming language; or if the text is being read as natural language or programmable code. There are too many binary positions of critique in Cayley's discourse to deconstruct at once in this

thesis, so the following chapter will focus on one, namely the question of executable/non-executable code. Rita Raley reads the executable/non-executable binary critique into Cayley's argument and she addresses this in her 2002 article when she states that "for all of the differences among particular instances or events of codework, they all incorporate elements of code, whether executable or not." (Raley 2002, par 7) Both Raley and Sondheim address Cayley's suggestion that non-executable codework should be excluded from the field by acknowledging the divide, and in doing so suggest that non-executable work should be included. N. Katherine Hayles also acknowledges this executable/non-executable divide in her survey of codework as electronic literature. Hayles frames the non-executable as a broken creole and the executable as a pure form of codework and attributes the critique to Cayley at this point of her survey.

Within his article titled *The Code is not the Text (unless it is the Text)* Cayley refers to the issues of reading what he terms interface text as code. One argument Cayley makes throughout this article is that the consideration of works that are non-executable as examples of codework is a utopian scenario. He states that "the utopia of codework recognises that the symbols of the surface language (what you read from your screen) are the 'same' (and are all ultimately reducible to digital encoding) as the symbols of the programming languages which store, manipulate, and display the text you read" (Cayley 2002, par 5) and for Cayley this "simplifies the intrinsically complex address of writing in programmable media." (Cayley 2002, par 8) But rather than simplifying the complexity of writing in programmable media, it is my claim that considering all readable textual elements as digitally encodable reveals the way codes are constructed and how the executable/non-executable codes in programming languages are authored or designed. A computer programer writing with literal textual computer code can do so according to a syntax that defines the ways in which the programmer's code will function according to the design of language being used. The programmer composes literal text, often through a keyboard, into a text editor. The typed code will be sent, either through the editor or separately, to a compiler or interpreter, which is a computer program written by the

designer of the language. This compilation or interpretation process can continue through different languages to translate the textual code into electronic functions of the computer. If the code is valid according to the syntax of a language, or a number of connected languages, then the code will eventually be passed to the central processing unit (CPU) of the computer. The computer's compilation process that transforms languages between structures and contexts is one reason for the multitude of coding languages that are available. Some examples of the diversity of language structures include ones based on Arnold Schwarzenegger (ArnoldC), lolcat lolspeak (LOLCODE) or non-ascii character sets like the Arabic programming language قلب (Alb).

Literal text as computer code is flexible and interpretable to the point where you could assign programming properties to every character in an alphabet transforming all natural language elements into a computer programmable syntax. A computer programming language that assigned all literal text elements to functions of the CPU within its syntax would be considered a valid language in the field of computer science. Such a language could be considered verbose and lack expressive power yet it would not be excluded. Verbosity, terseness and expressive power are qualities that may be considered by a programmer before they decide to use a language, but even if a language was not used because of its inappropriateness for the programmer, it would still be valid. Although Sondheim attempts to frame Cayley's critique as being based on a subjective definition of code, for Cayley, he is arguing for general definition. To deny in the validity of a programming language that encoded all compilable marks and symbols as functions of the computer would overlook an entire group of programmable codes called esoteric programming languages. Daniel Temkin is an artist who has written programming languages like *Entropy*, a language that decays the data of the program as it is run; *Folders*, a language where a program is written using empty file folders; and *Light Pattern* a programming language that is written with photographs. Temkin also writes a blog called esoteric.codes in which he presents, as the title to his website states, "programming languages as experiments, jokes, and experimental art." (Temkin, 2015) In a post published

in 2015 titled *Executing code off the machine, or: non-deterministic processes don't have to give you a heart attack*, Temkin suggests that computer programming languages do not even have to be executed on a computer at all, for example he suggests that an esolang could be "a person scratching symbols in the sand." (Temkin 2015, par 2) Rather than simplifying the complexities of executability and code, a computer programming language that could execute a function within the computer by encoding all literal text reveals the complexities of how programming languages are designed and function. Furthermore if such a language were constructed the executable/non-executable critique of Cayley would no longer be relevant, or would shift, as there would no longer be any executable/non-executable codework written in literal encodable text. Works of Breeze, Memmott and Sondheim would be executable just as this paper would be executable. The existence of such a language compiler would make all readable literary works potentially executable, whether written online, in a book or in the sand.

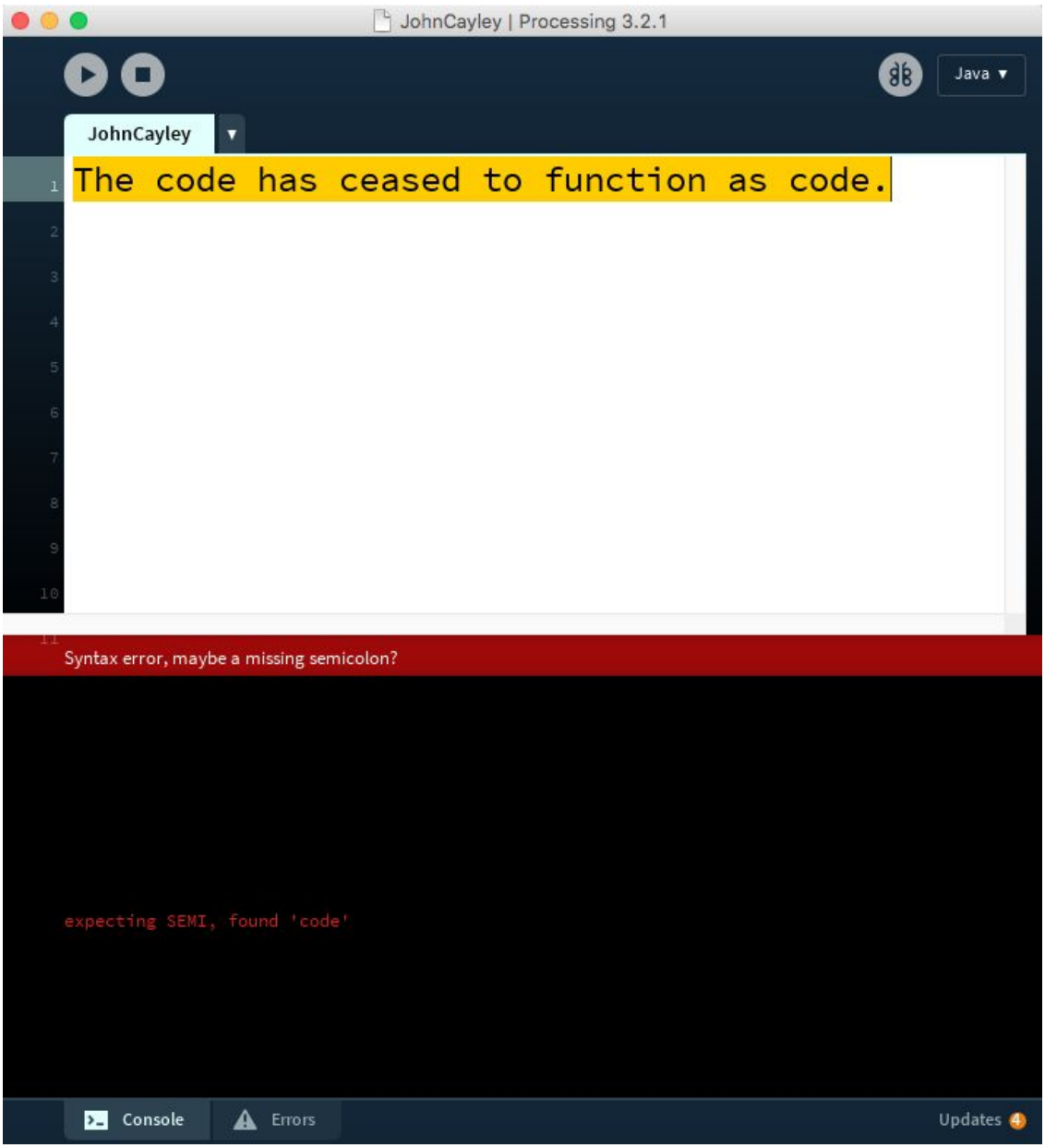


Figure. 1: screen shot of text written into the Processing Development Environment

How Exceptional Texts Work

Rather than rely upon a hypothetical esoteric programming language to deconstruct the executable/non-executable binary we can take an existing programming language and compile some literal text to further investigate John Cayley's critique. The language that will be used to attempt this is called Processing and it started in 2001 by Ben Fry and Casey Reas. The above image is a screen shot of an attempt to compile some natural or surface language in the Processing Development Environment (PDE). In this example I have taken a sentence from Cayley's critique of non-executable code, typed it within the PDE and attempted to compile the statement as programmable code. In this example the PDE has presented a number of outcomes that are returned once the program is made to play or run. These outcomes include the line of text written being highlighted in yellow; some words in the written text being underlined red; a phrase being presented in white text upon a red background; and a phrase being presented in red text upon a black background. I will refer to these three outcomes as a series of exceptional messages and the process that creates these messages will be referred to as throwing exceptional messages. When Cayley states that "the code has ceased to function as code" in his critique of codework he seeks to place non-executable text, which is not written according to any programming language syntax, outside of the field of codework. (Cayley 2002, par 26) Throughout the codework discourse, through Sondheim, Raley, Cayley and Hayles, these texts are referred to as interface, surface, broken, natural language texts and set against programmable, narrow or pure codes. Despite being included/excluded within the field of codework, all of the theory posed by these academics above asserts that this binary position of executable/non-executable literal text exists and can be differentiated in these general terms. But any literal text placed within most programming language development environments will function or create a response within the computer and therefore cannot be represented in this way. The code that has ceased to function as code in the example above has functioned as code by design. In this example, the Processing language is designed so it executes this text as exceptional text and in doing so an exceptional message

is thrown. This message reads: *Syntax error, maybe missing a semicolon?* and *expecting SEMI, found 'code'*.

Throwing Exceptional Messages

When a computer programmer writes text into the development environment of their chosen language they are expected to do so according to the structure, or syntax, of that language. Once a programmer has written some text, for example in the PDE, they compile and run it. Compilation begins with a screening process, a parsing of the text according to the syntax found in the pre-processor of the programming language. If the text is written according to the syntax of the language it is then compiled and run, or executed. The Processing language is derived from another language called Java and according to the Java language documentation "when a program violates the semantic constraints of the Java programming language, the Java Virtual Machine [JVM] signals this error to the program as an exception." (Oracle 2011, par 1) A Java tutorial on the language designer's website goes on to state that an exception is defined as "an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions." (Oracle 2015, par 1) In regard to the PDE the compilation and execution of the program happens with one action, as the programmer selects the play button or run option on the PDE. The written text will be run if there are no checked exceptions at compile-time checking but if the program does run and there are run-time exceptions they will either be handled within the code or stop the program entirely. (Oracle 2011) In either example there is a distinct process that the Java language carries out through the JVM. When the JVM identifies that there is an exception in the syntax of the text written by the programmer it creates an object called an exception object, moves this object to the runtime-system, a process called throwing an exception, and searches for code called an exception handler that can manage the text. (Oracle 2015) "The exception handler chosen is said to *catch the exception*. If the runtime system exhaustively searches… without finding an appropriate exception handler… the runtime system (and, consequently, the program) terminates."

(Oracle 2015, par 5) And when a Processing application terminates or even fails to run, due to exceptional text being present, the JVM handles the exception by throwing an exceptional message to the programmer. Exceptional text, created as an exceptional object, caught, handled and thrown is executed as a non-executable exceptional message. The execution of the non-executable exceptional text is a process of object construction and the exceptional object it produces differs between programming languages.

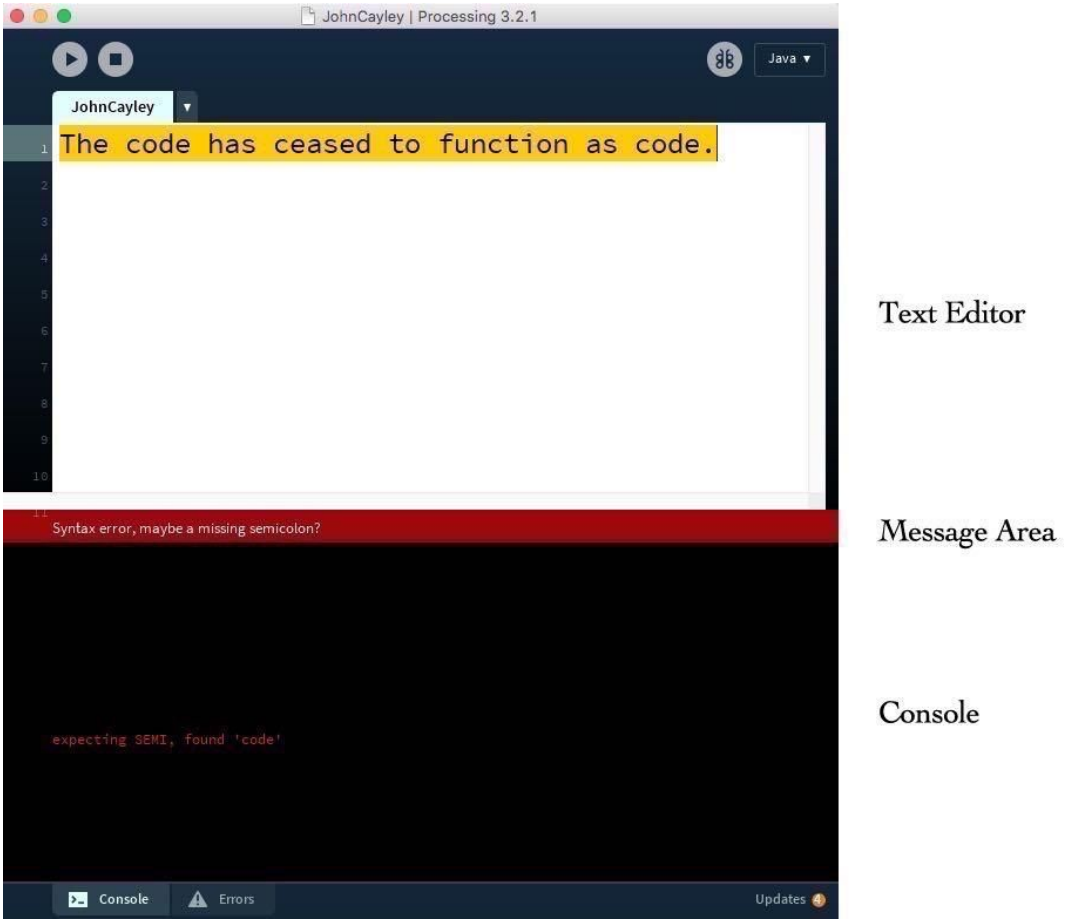


Figure. 2: screen shot of text written into the Processing Development Environment with area labels

The Processing language throws exceptional messages to the PDE in a number of different ways and these can be seen in different places within the PDE. Three of these areas are the text editor, message area and console. The Processing website describes the text area as the place where programs are written; the message area as a place that displays

errors; and the console as a place that displays complete errors. (Processing 2017, Environment) As Processing is a programming language built from the Java language we can understand how these exceptional messages are thrown by referring to the Java documentation page titled *How to Throw Exceptions*. This documentation begins by stating, "before you can catch an exception, some code somewhere must throw one. Any code can throw an exception: your code, code from a package written by someone else such as the packages that come with the Java platform, or the Java runtime environment. Regardless of what throws the exception, it's always thrown with the throw statement." (Oracle 2015, par 1) The code, which is the throw statement that threw the exceptional message in the example above can be seen in the diagram below, which has been taken from the PDE source code. This code is written by the developers of the Processing language and it functions within the PDE in a number of ways. Code within computer programming languages such as Processing can catch exceptional text and handle it by throwing it in particular ways according to the authorship or design of the language developers. As can be seen from the diagram above there are a number of exceptional messages being thrown in the PDE when exceptional text is written. I claim these messages are the executed functions of non-executable text as computer code in the PDE. The so called non-executable critique of John Cayley has executed or functioned as code; it has highlighted itself in the text editor, thrown a particular message in the message area that is determined by the particular structure of the text written and thrown a particular message to the console that includes a part of the text. These functions are particular as each written word and its context within the text editor determines the messages executed by the throw statement. These messages are a functional outcome of the text written that can be accessed when the syntax of the designed language is not followed.

```
if (msg.indexOf("expecting SEMI") != -1) {
  System.err.println(msg);
  throw new SketchException("Syntax error, " +
                            "maybe a missing semicolon?",
                            errorFile, errorLine, re.getColumn(), false);
}
```

Figure. 3: screen shot of code that is stored on the Processing Github code repository (Github 2016)

Exceptional messages are a pedagogical tool to correct the programmer and tell them that what they have entered as input does not fit the structure of the language. In Java there are a number of built in exceptional messages that are sent to the programmer when they compile and run their code. Java also allows programmers to modify the messages that are sent so the designer of a language or program can make exceptional messages of their own. The designer of a language also designs the messages sent to correct the programmer using that language and one motivation for this is to improve exceptional message readability. Java system exceptional messages can be considered difficult to read as is documented on a website offering tutorials in the Java programming language. A tutorial written by a senior software engineer, Chaitanya Singh, describes Java system exceptional messages as "not user friendly" that the user would find hard to comprehend. (Singh 2014, par 5) In the SketchException.java file of the Processing language, which throws exceptional messages, a programmer contributing to the design of the Processing language has written a comment that states, "nix the java.lang crap out of an exception message because it scares the children." (Github 2015, line 134) And the children should not be scared of the exceptional messages thrown by the Processing language because these messages are used as a way to correct their written text. Processing was initially designed as "a first programming language" (Processing 2017, par 4) that "promoted software literacy." (Processing 2017, par 1) So, for this language in particular, exceptional messages are thrown to someone who may be encountering programming language syntax for the first time. Designers of programming languages often depend on transparent and comprehendible documentation of

their language's syntax so people are able to learn and use their language. If a programmer learning a language cannot amend their mistake, according to the syntax, then they will not be competent in that language. It is my claim that the reason why exceptional texts written into a programming environment are thrown as exceptional messages is to correct the programmer.

Any deviation from the structure of the Processing language is designed to go back into the structure as a corrective, pedagogical, exceptional message that points the programmer to the language structure itself. One feature of Processing that can be enabled from the system preferences has the PDE "continuously check for errors and show warnings." (Processing 2017, par 75) As the programmer types each letter of their text the PDE is checking to see if what is being written is an error or exception. In the figure bellow we can see that when this preference is enabled the exceptional text is underlined in red and the exceptional message is shown in the message area. It is important to make the point that in this example the programmer has not attempted to run their code. There is anticipation that occurs when the PDE is checking the text as it is being written that does not occur when the continuous preference is off. When the continuous checking is on, the PDE throws error and warning messages even when the programmer is writing their text according to the language structure, but has just not completed the text they are writing. This anticipation of what the programmer is writing is an interruption or intervention into the writing process. When the programmer is able to complete their text, without interruption and attempt to compile their text in the PDE, they are able to do so in their own time without their text being thrown back at them as an exceptional, correctional message, whilst they are writing. Now, with continuous checking for these exceptions, the programmer is constantly made aware of the structure, even when they are writing according to it, and are being corrected before they have had a chance to complete their text

whether it is or is not according to the structure. It is my claim that this process of continuous exception checking, one based on continuous anticipation and interruption, alters the way that the programmer is framed by the design of the checking system. The programmer is now assumed wrong as soon as they make a mark by way of typing a letter or symbol and they are constantly suggested what is wrong with their attempt. It is in my opinion that this form of exception checking is not handling the code, checking for and correcting exceptions in the code, but rather is handling the programmer. The programmer, with a hand on their shoulder as they write each piece of text, is never allowed to make a mistake as the program is never completed before being corrected.

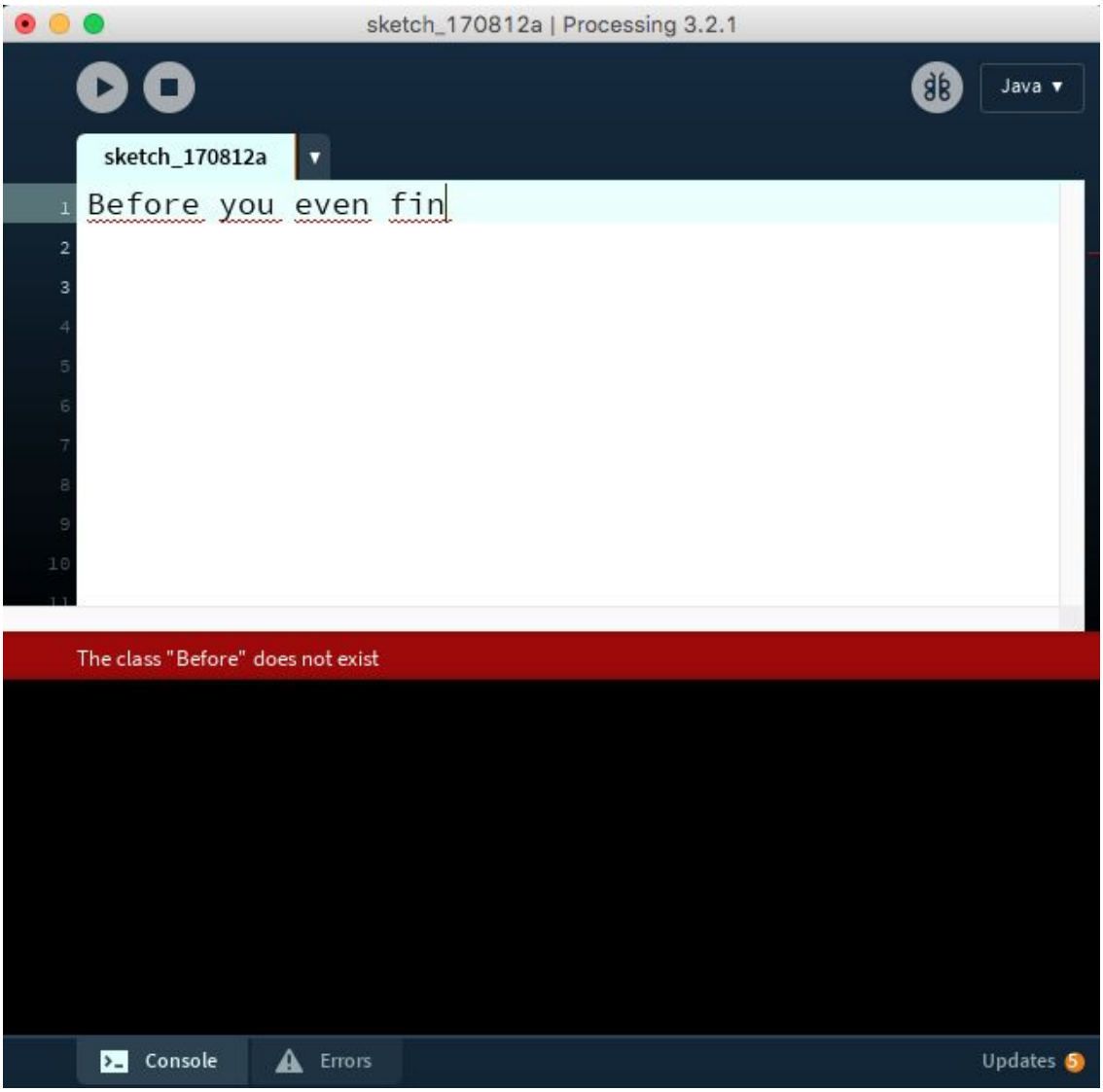


Figure. 4: screen shot of text written into the Processing Development Environment with continuous checking enabled

It is possible for exceptions to be handled in other ways that do not utilise them as corrective opportunities. *FuckItJs* is a Javascript Library written by web developer and composer Matt Diamond. It is stated on the github description page where the library can be downloaded from that "through a process known as Eval-Rinse-Reload-And-Repeat, FuckItJS repeatedly compiles your code, detecting errors and slicing those lines out of the script." (Diamond 2012, par 2) Javascript is typically used for and compiled inside of a web browser. This library acts as an intermediary between the text and the browser's

compilation process. When this library detects that a line of code has an error or exception it handles it by removing that line entirely from the code. It continues this process, line by line, until all lines that have exceptions in them are deleted. Another Javascript library that is designed to handle exceptional text before reaching the Javascript compiler of the browser is *FatFingerJS* developed by Daniel Temkin. FatFingerJS is described as a "library expanding JS [Javascript] to allow typos and misspellings." (Temkin 2017, par 1) Temkin asks, "why bother with clean, well-formatted code when you can write this and FatFinger will guess at your intentions?" (Temkin 2017, par 1) FatFingerJS changes the programmer's code so it fits the Javascript structure and then the changed code is passed to the Javascript compiler within the browser. The text that contains exceptions in these two libraries is still being corrected, but the corrections are being made by the designer of the library, rather than by the programmer via the instruction of an exceptional message. The designers of Processing show that they are bothered with the programmer knowing clean, well-formatted code by having the programmer correct their own code, as opposed to removing, or assuming what is meant by the programmer's exceptions to the structure. It is my claim that the approach that the designers of Processing take is once again a means of maintaining and controlling the Processing language structure by correcting the programmer and especially a programmer in the process of learning.

Maintaining and controlling the relevance and use of a programming language depends on exceptional texts functioning as exceptional messages. The exceptional or non-executable text cannot be left functionless because it becomes integral for the persistence of the language. If the non-executable exceptional text was not made executable as a message then upon attempting to execute the text, the programmer, wanting to make a program, might be met with something unexpected or unintended. Potentially nothing would be executed and the programmer would have to return to the written text and find where it did not fit the structure. This may be a lengthy and uncertain process for a programmer who is still learning the language. Potentially the code could be corrected by the structure itself to execute an approximation of what the structure can assume was the

intended meaning of the text, but these assumptions may lead to the programmer being ambiguous about the language, or the language designers being ambiguous about the programmer's knowledge and use of their language. But no matter what the designer of the language decides to implement for the exceptional text we can see how the exceptional text can be handled as in the case of exceptional messages where they are thrown to correct the programmer. The binary terms of executable/non-executable are placed under erasure when considering the use of the exceptional message in the Processing language structure. The executable elements of this language depend on the executability of those named as non-executable to maintain the knowledge of its syntax. The executable/non-executable distinction can no longer exist in general terms when we consider the exceptional text as executed code within the PDE. The deconstruction of the executable/non-executable binary shows that any written literal text that can be encoded by the computer is executable. The non-executable is executable and we see how it is executed as an exceptional message that frames the text as a mistake, error or exception to the structure. When there is a division between executable/non-executable encodable text, or if an exceptional message is used as a corrective process upon some exceptional text we can read this as the maintenance of a particularly coded structure. The text itself is not non-executable but rather has only been named that way by the designers of the structure. The non-executable is named that way so it can execute the correction of the programmer. The non-executable is structured as a means for the designers of a language to reassert the language structure back upon the programmer and disallow any deviation from it.

Maintaining Coded Structures

The so called non-executable or exceptional text is made to be considered a lesser element of a coded system through its framing as an error or non-functioning contribution, despite its utilised functionality, as a way of maintaining the dominance of a particular use of a coded structure. We can see how binary assertions create positions of dominance in the codework and electronic literature discourse through the inclusion/exclusion of codework

on the basis of the executable/non-executable argument in the discourse. But another troubling point of comparison in this discourse which reflects on the violent context of language structuring was made by N. Katherine Hayles when she compared the work of Mez Breeze and Talan Memmott's to the colonial context of creolization and furthermore went on to use this context to contribute to the executable/non-executable codework debate. In an article published in 2003 titled *Deeper into the Machine: The Future of Electronic Literature* Hayles wrote that "Talan Memmott's '*Trans*lucidity' and MEZ's 'mezangelled' productions push toward the creation of a creole comprised of English and code." (Hayles 2003, par 3) Hayles goes on to ask readers to question what kinds of "subjects" and "subjectivities" will be formed by this "…interplay of human language and machine code." (Hayles 2003, par 13) It can be understood, as it was by Cayley, that Hayles is asking us to envisage a future scenario for languages like Breeze and Memmott's. It is important to note here that Cayley referred to the colonial context of the term creole in response, yet he did so to refute the importance of the use of executable/non-executable languages in a future scenario. Cayley introduces the colonial context in this debate by ridiculing the assertion that one day there could be an "invasion of an empire of machinic colonizers" that could prompt the use of these languages. (Cayley 2002, par 26) But at no point in Hayles' article or in the codework discourse have we been asked to reflect on the historical context of creolization beyond this utopian statement of Cayley's or question the implications of the use of the term creole in describing a broken non-executable as opposed to a pure executable codework. (Hayles 2007)

Non-executable or exceptional texts are not broken but are made to work to maintain the designed structure of the language. Not only does Hayles frame the exceptional text as broken but also describes a set of languages, referred to as creole, as broken or not able to be executed. Furthermore, Hayles decides to place the broken creole language into a binary position against the executable or pure language that as we have seen earlier in this thesis was used to position the pure executable in a dominant position over the broken non-executable. I would like to interrogate this framing further and believe

it is important to do so through a definition of creolization offered by Édouard Glissant. In his book *Poetics of Relation* Glissant states that "…the word *creolization*, approximates the idea of Relation for us as nearly as possible. It is not merely an encounter… a *métissage*… if we posit *métissage* as, generally speaking, the meeting point of two differences, creolization seems to be a limitless *métissage*, its elements diffracted and its consequences unforeseeable." (Glissant 1997, 34) If we consider the meeting points of what Hayles frames as the English language and machine code in these terms of a process of creolization we can deconstruct Hayles' binary use of the terminology. We overlook a multiplicity of materialities and histories when considering machine code set against English as two pure or singular forms coming together to create a binary understanding. When writing computer programming languages we are engaging with a multitude of materialities that make the computer. The keyboard, mouse, circuitry, to name a few, can all be seen as elements diffracted and these elements, along with others, collide together to make computer programming possible. By seeing machine code as one pure formation that has a general totality is to overlook the many different formations of the computer and how it is used to create code that we can interact with. Overlooking the possibly limitless collisions that can happen to form different instances of computer code also risks overlooking the different forms that computer codes have taken historically. In an article titled *On Software, or the Persistence of Visual Knowledge* Wendy Chun states that "software and hardware (like genes and DNA) cannot be physically separated" and reminds us that computer programming once "comprised the human task of making connections, setting switches, and inputting values ("direct programming"), as well as the human and machine task of coordinating the various parts of the computer." (Chun 2004, 28) When we consider machine code as a multitude, we can bring into question what Hayles sees as pure, which I claim can also be read as a singular, ultimate, or correct form of language in the context of the codework discourse. We can reframe this purest structure as a set of elements diffracted, which have come together over time as machine codes, in relation to many other elements.

Similarly the English language could be brought into question considering Glissant's definition of creolization. We could turn to its historical formation from a number of differing languages to remove the limited totality that is given by framing it as a pure language. Furthermore, English can also be seen as a limitless set of elements if we were to consider each utterance of the English language as a unique English that is different than another utterance. *In My Own Words* is a documentary film written and directed by Erica Glynn that "…follows the journey of adult Aboriginal students and their teachers as they discover the transformative power of reading and writing for the first time in their lives." (Blackfella Films 2017) There is a moment in the film where one of the teachers, a Cuban man named Chala LeBlanch, addresses a group of young Aboriginal men as he introduces a DVD that they will be using for their lesson. LeBlanch describes it as "…a DVD with explanation about the program in general. Those people you will see are from Grenada, a small island in the Caribbean. So they are black people like us because they were conquered by the British but they speak a different English. So there will be three different English here, you, me and them. But we get along very well." (LeBlanch 2017) When considering the use of language in this way it is possible to refute the construction of a singular or pure form of a language. Rather, what is being constructed is a mechanism to execute a particular use of a language as broken, so those who have authority over the language can attempt to correct the users of that language and attempt to disallow the language's adaptation. Aboriginal English and Australian English are two dialects that have developed side by side since the colonial settlement of the British began in what is commonly considered today as a nation called Australia. Emeritus Professor Ian Malcolm spoke about the developments of the English language in a radio broadcast called *Awaye* where he stated, "from the time of settlement two different Englishes progressively developed in Australia, one among the settler community, and the other among the Indigenous community. The settler varieties were eventually levelled out into Australian English and Standard Australian English but, meanwhile, the Indigenous people had a different path towards English." (Malcolm 2011, par 2) He goes on to state "there are many, many different Englishes because people in different parts of the world have taken

ownership of English." (Malcolm 2011, par 14) But these Englishes can face resistance as he notes has been the case for Aboriginal English in Australia, due to the opinion that there is only one English. This view has lead to our current environment, where people who speak other Englishes are being corrected in educational systems, told their English is not legitimate and then left at the bottom of standardised educational indicators. (Malcolm 2011)

The notion of a pure or singular language structure known as English or even a coded structure known as machine code is constructed. For Glissant "creolization carries along then into the adventure of multilingualism and into the incredible explosion of cultures. But the explosion of cultures does not mean they are scattered or mutually diluted. It is the violent sign of their consensual, not imposed, sharing." (Glissant 1997, 34) The violence that Glissant speaks to here is different from the violence that Derrida speaks to in the formation of binaries. The violence in a binary comparison is one of dominance, the formation of one over another, which in the case of the exceptional message is constructed to correct and control. The violence of creolization is in the collision that creates a dispersion of new elements at multiple meeting points and these diffractions are propelled to create further collisions. An important distinction here is in the consensual not imposed sharing that makes creolization as defined by Glissant a violence of mutual materialisation and not of control. The Processing language is developed through a system of open source code where anyone can come and contribute to the code through a Github web repository of the language. The PDE has been changed over 12,000 times with over 400 versions released since it has been stored on Github. The colonial reference of Hayles and how she frames the non-executable as a broken creole are general statements of executability made without the consideration of cultural power and design. The design of programming languages are not general or free from culture, but are culturally designed to work in particular ways for particular people. Ramsey Nasser designed the Arabic programming language قلب (Alb) as a "conceptual art piece" to ask "could you really build a language that didn't use the latin alphabet?" (Nasser 2013) His work not only questions the historical

development of programming languages but it also brings into question the accessibility of other languages within computers. Nasser explains that "قلب is built entirely on Arabic and everything broke. Every text editor has no idea what to do, the terminal is useless, all of the tools that I use to be creative using code fall apart." (Nasser 2013)

Instead of considering a piece of literal text as being broken because it does not work within the computer according to a particular syntax we can begin to question the executing system instead. How has the structure been designed to break when it meets Ramsey Nasser's language? Rather than excluding it we can consider what can come from the meeting of the Arabic language and the computer programming language structure. Nasser continued the development of his language and found that there are particularities of the Arabic language that offer new ways of computer programming.

"Arabic language has some very interesting properties that lend itself towards code. Arabic is a join language where certain letters join with the letter that follows them using a line so that words form almost like solid forms. What you can do in Arabic is you can stretch out the length of that join so you can align things perfectly in Alb just by stretching all the words out and make it look visually beautiful in a way that you couldn't do with English code and that's just entirely the result of just using Arabic as a text." (Nasser 2013)

Nasser revealed new opportunities for computer programming languages by breaking through the limitations of the paradigm that was offered to him and created a new consequence from this collision. Angie Abdilla is the founder and CEO of Old Ways, New, a company of Indigenous Consultants and Technologists who describe themselves on their website as people who "tap into Indigenous Knowledge Systems by drawing upon tens of thousands of years of culture, research, iterative design and innovation of technology." (Old Ways New) In an interview on ABC's Radio National, Angie Abdilla, along with company Elder Mukgrrngal Wayne Armytage and Director Dr. Robert Fitch spoke about the possibilities of approaching robotics from the knowledge system of indigenous peoples,

considering ways of mapping and protocols that draw from indigenous ways of relating and knowing. In the interview Abdilla spoke about the approach the company took whilst putting together a robotics workshop for indigenous students at the Centre for Indigenous Excellence. She says, “we were interested in how we could introduce emerging technologies that could tie us back to the language of code… we thought about robotics and looked at the different ways in which we could introduce robotics… to look at, how do we connect indigenous knowledge and an indigenous way of doing things to robotics?” (Abdilla 2017) Rather than taking the established and widely used Western approach to coding robotics the workshop took an approach that was particular to Aboriginal culture. Abdilla states that to do so “first and foremost we have to come back to protocol, and understanding how indigenous knowledge is different to the Western idea of knowledge, is fundamental to first of all even understanding your place within it.” (Abdilla 2017)

For new knowledge to be recognised, the protocols that we have in code must first diffract, explode and break the dominant paradigm. When Ramsey Nasser created قلب he witnessed the breaking of protocols, which lead him to redesign all of the elements that he needed to give the Arabic language structure access to the computer. As Abdilla says

“…when thinking about how we can move forward, we could potentially see code coming together in a different way if we take the understanding of pattern thinking, which is an ability to see a relationship between all things and understanding that in its truest sense, is the embodiment of that. Then for indigenous peoples to be able to look at and create code in a different way. Then I wonder if there’s an opportunity to create different types of technologies that have a connection back to country, a connection to a kinship system that has a more highly nuanced relationship.” (Abdilla 2017)

The ability to see a relationship between all things is something that has been missing from the discourse and analysis of the executable/non-executable binary

construction in the field of codework. This discourse has been caught handling two-sided arguments, approaching code as a means of defining and controlling a field. Having limitless consequences is possible in the field of codework if this discourse is able to cease from imposing its taxonomy and reducing coded languages to the binary paradigm. This would be the consensual violence and explosive potentiality of creolization as Glissant has shown us through his definition. Rather than framing what works or does not work we should encourage codeworkers to break down the standardised, dominant paradigm as a means of identifying and dismantling controlling and imposed violent discourse.

CONCLUSION

Through deconstruction of the codework discourse, this thesis has raised two particular issues that I would like to note in this conclusion as areas that are in need of further critique, which is not in the scope of this thesis. Firstly, the issue of the exceptional message as a corrective message in the PDE, which is used as a means of maintaining Processing's particular language structure by structuring the language use of the computer programmer. This is a pedagogic act of correction. This act raises issues in regard to pedagogical methodologies when determining the accepted, standard, dominant or general use of a language structure in particular environments. Secondly, the issue of addressing the use of the term creole in N. Katherine Hayles' contribution to the codework discourse led to a discussion of other examples that encompassed creolization such as that of Aboriginal English. When considering the further trajectory of this research I would suggest that a starting point be that of aforementioned academic Ian Malcolm, who along with Ellen Grote, Louella Eggington and Farzad Sharifian with the support of the Australian Institute of Aboriginal and Torres Strait Islander Studies published what they consider as the first study of its kind titled *The Representation of Aboriginal English in School Literacy Materials*. (Malcolm et al. 2002, i) This thesis raises questions about the colonial context of creolization and its presence within the dominant pedagogy of computer programming structures in the Australian educational context. These questions would need to follow and address work already being done by Old Ways New founded by CEO Angie Abdilla, Ramsey Nasser, Ian Malcolm et al. and Édouard Glissant to name a few. Questions could be asked of what Aboriginal and Torres Strait Islander codes and structures exist in the Australian education system when considering the pedagogy of computer programming languages and computational structures? What can computer code as a literary practice offer pedagogical material in Australia when it is considered from the perspective of the codes that are present in Aboriginal and Torres Strait Islander cultures? It is my intention to note that this thesis has raised these questions and due to the scope of the research cannot attempt to answer them here.

Codework is particular to those who make it and they structure it from their own process of knowledge creation. Rather than considering particular examples as broken, we can consider the *creolization* that Glissant offers when considering a consensual space where cultures and languages diffract both internally and externally, between and within, across and down, historically and speculatively. The Processing language is one particularly designed language that can be used according to its structure to produce computer programs. Within this language syntax are functions that maintain the syntax by throwing messages to the programmer when the text they have written into the PDE does not conform to the particular syntax of this language. If a programmer chooses to write text that is not within the syntax they can expect to be thrown exceptional messages. These exceptional messages can therefore also be considered as a part of the Processing language syntax. All literal text that is written within the PDE is therefore executable. Some of this text is caught and thrown as exceptional messages that correct and support the language itself. This is particular to the Processing language and it has been structured to correct and support the language itself. Codework can be written in many languages, some will be unforeseeable because they will be created at meeting points of difference. If codework is a place where computer encodable texts meet literature then we should consider how these works are read on their own terms.

When Alan Sondheim set the taxonomy for the formalised field of codework in 2001 he structured it in a way to describe the qualities of code and literature that come together to create this field. Since then Sondheim has released an article titled *Code, work* published in 2016. This article does not open with a formalised purpose but it does speak to codework in the manner of a descriptive list. It describes what codework is in each point but often in a way that is not as didactic as Sondheim's article and taxonomy in 2001. One point of this list describes codework as "always already _a failure_ or collapse; it swirls and transforms blankspace into crashed subjectivity; it gnaws itself. It does nothing and to the extent that it's a _style,_ it negativizes language past the point of redemption, to useless

hipsterisms. But then this failure is _of interest." (Sondheim 2016, par 9) What this thesis has aimed to do is deconstruct Sondheim's formalisation of the general field of codework and the subsequent debates over the executable/non-executable. The method of placing the executable/non-executable part of the discourse sous rature or under erasure is used to collapse the general definition, to produce a particular critique that shows the power dynamics at play when including/excluding works of difference in/out of a field. This collapse is aimed at objective definitions to set them crashing into a context of subjective critique of works that already have their own defining principles that are always different to the general field of codework. Code is not a blank space that works according to a pure paradigm. Code that works is made to work according to a particular set of rules, and these rules are subjective and a part of particular cultures. The failure of code within coded structures reveals the structure of failure and how failure is framed. In the case of the executability of code that does or does not work we have come to see this binary construct as a correctional act upon texts of difference. The non-executable is the exception to the rule and is made to execute, to highlight its faults and be changed so that it can be read according to the design. Code that is made to work in this way and the programmers who write it experience the authoritative, pedagogical correction that maintains a structure.

The purpose of this thesis is to return to the historical context that framed the formalisation of a general field called codework and trace a discourse through time to this contemporary moment. Both Hayles' recent republication and Sondheim's recent publishing of *Code, work* point to the relevance of this discourse to this day. This thesis is not situated as a historical analysis, but rather through the use of the deconstructive methodology it has been placed alongside this contemporary moment to create a critical artefact that offers a new perspective. The third chapter of this paper is the physical deconstruction and execution of this methodology as a performative act within a gallery space. The paper that you are reading is situated within this space where the artefacts can be read as exceptional messages resulting from the systematic programming of this thesis within the PDE. This system of programming was developed as a piece of codework that

comes from but cannot be read within the executable/non-executable divide. It is left here in this space for you to observe as a trace that can be followed back to the formation of the field of codework, the subsequent debates of the executable/non-executable divide, and back again to this contemporary moment where what we read is a new paradigm of code that works as exceptional text.

Works Cited

Abdilla, Angie, interview by Joe Gelonesi. "Robotics, AI and the power of slow." *The Philosopher's Zone*. ABC Radio National. Accessed August 16, 2017. http://www.abc.net.au/radionational/programs/philosopherszone/robotics,-ai,-and-the-power-of-slow/8224076.

Blackfella Films. 2017. "In My Own Words Synopsis." Accessed August 16, 2017. http://blackfellafilms.com.au/project/in-my-own-words/.

Breeze, Mez. 2001. "THE DATA[H]BLEEDING TEXTS" *netwurkerz website*. Accessed August 16, 2017. http://netwurkerz.de/mez/datableed/complete/index.htm.

Cayley, John. 2004. Discussion. *Nettime Mailing List*, February 2004. Accessed August 16, 2017. http://www.shadoof.net/in/whitecubebluesky/alsoexchange.html.

Cayley, John. 2002. "The Code is not the Text (unless it is the Text)." *Electronic Book Review*. Accessed August 16, 2017. http://www.electronicbookreview.com/thread/electropoetics/literal.

Chun, Wendy. 2004. "On Software, or the Persistence of Visual Knowledge." *Grey Room 18* 28: 26–51.

Derrida, Jacques. 2006. *Of Grammatology*. Translated by Gayatri Chakravorty Spivak. [place]

Diamond, Matt. 2012. "FuckItJS." *FuckItJS Github Repository Description.* Accessed August 16, 2017. https://github.com/mattdiamond/fuckitjs.

Frabetti, Federica, interview by Janneke Adema, "Software Theory – Federica Frabetti." *Culture Machine Live*. Podcast audio, February 25, 2015. Accessed 16th August, 2017. http://culturemachinepodcasts.podbean.com/e/software-theory-federica-frabetti/.

Glissant, Édouard. 1997. *Poetics of Relation.* Translated by Betsy Wing. Michigan: The University of Michigan Press.

Github. 2015. “SketchException.java.” Last modified August 14. Accessed August 16, 2017. https://github.com/processing/processing/blob/0abee5af6ad3b11cf2b73bb794b8a97c157c4762/app/src/processing/app/SketchException.java.

Github. 2016. “JavaBuild.java.” Last modified November 10. Accessed August 16, 2017. https://github.com/processing/processing/blob/c6c433ff57c7e60faf5f477591a49691ec70ed6c/java/src/processing/mode/java/JavaBuild.java.

Hayles, N. Katherine. 2003. “Deeper into the Machine: The Future of Electronic Literature.” *Culture Machine* 5. Accessed August 16, 2017. https://www.culturemachine.net/index.php/cm/rt/printerFriendly/245/241#Note%205.

Hayles, N. Katherine. 2007. “Electronic Literature: What is it?” *The Electronic Literature Organization Website*, v1.0 January 2. Accessed August 16, 2017. http://eliterature.org/pad/elp.html

Hopper, Grace. 1953. “Compiling Routines.” *Computers and Automation* 2:1-5.

LeBlanch, Chala. 2017. “In My Own Words.” Directed by Erica Glynn. Sydney: Blackfella Films Pty Ltd.

Malcolm, Ian, interview by Maria Zijlstra, “Aborigibnal English.” *Awaye*, ABC Radio National, December 3, 2011. Transcript accessed August 16, 2017. http://www.abc.net.au/radionational/programs/linguafranca/aboriginal-english/3709226#transcript.

Malcolm, Ian., Ellen Grote, Louella Eggington and Farzad Sharifian. (2002). “The representation of Aboriginal English in school literacy materials.” *Edith Cowan University Research* Online. Mount Lawley, Australia: Centre for Applied Language and Literacy Research, Edith Cowan University. Accessed August 16, 2017. http://ro.ecu.edu.au/cgi/viewcontent.cgi?article=7838&context=ecuworks

Nasser, Ramsey. “Arabic Programming Language at Eyebeam.” YouTube video. 3:00. Accessed August 16, 2017. https://www.youtube.com/watch?v=77KAHPZUR8g.

Old Ways New. “About”. Accessed August 16, 2017. http://www.oldwaysnew.com/purpose/#about.

Oracle. 2011. “Chapter 11. Exceptions.” Accessed August 16, 2017. https://docs.oracle.com/javase/specs/jls/se7/html/jls-11.html.

Oracle. 2015. “How to Throw Exceptions.” Accessed August 16, 2017. https://docs.oracle.com/javase/tutorial/essential/exceptions/throwing.html.

Oracle. 2015. “What is an Exception?” Accessed August 16, 2017. https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html.

Processing. “Environment (IDE). Overview.” Accessed August 16, 2017. https://processing.org/reference/environment/.

Processing. “Overview.” Accessed August 16, 2017. https://processing.org/overview/.

Raley, Rita. 2002. “Interferences: [Net.Writing] and the Practice of Codework.” *Electronic Book Review*. Accessed August 16, 2017. http://www.electronicbookreview.com/thread/electropoetics/net.writing.

Singh, Chaitanya. 2014. “Exception Handling in Java with Example Programs.” *Beginners Book*. Accessed August 16, 2017. https://beginnersbook.com/2013/04/java-exception-handling/.

Sondheim, Alan. 2001. “Introduction to Codework.” *American Book Review* 22, no. 6. Accessed August 16, 2017. http://litline.org/ABR/issues/Volume22/Issue6/sondheim.pdf.

Sondheim, Alan. 2004. Discussion. *Nettime Mailing List*, February 2004. Accessed August 16, 2017. http://www.shadoof.net/in/whitecubebluesky/alsoexchange.html.

Sondheim, Alan. 2016. “Code, work.” *Utsanga #9*. Accessed August 16, 2017. http://www.utsanga.it/sondheim-code-work/.

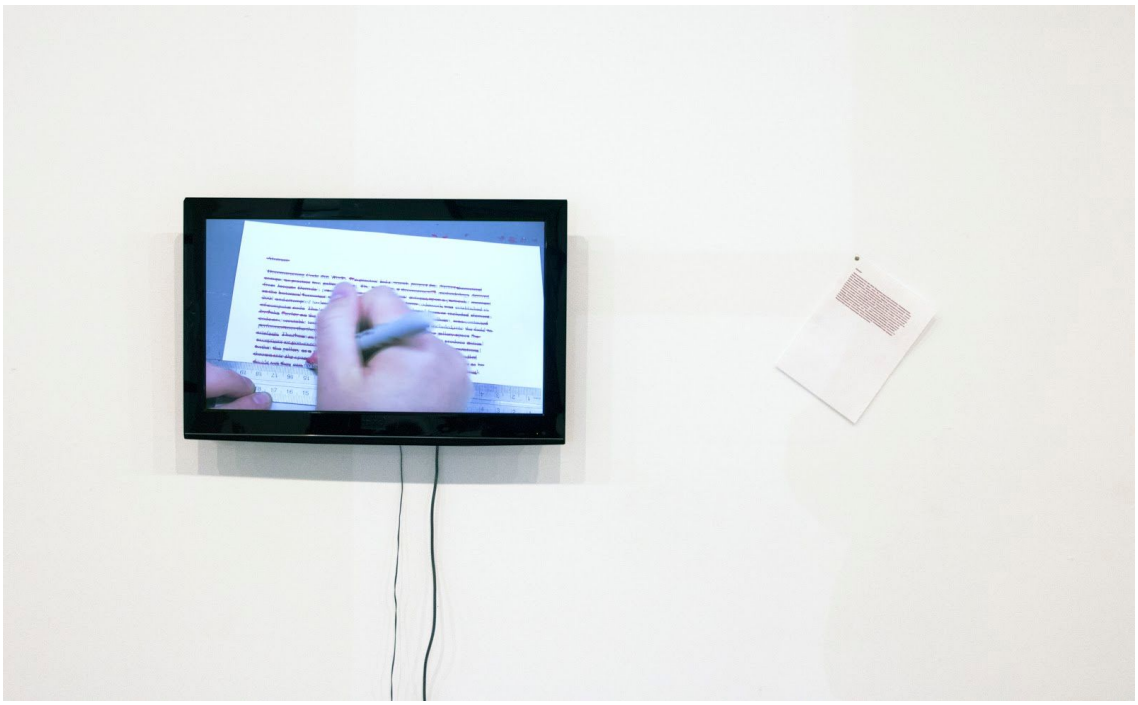
Temkin, Daniel. 2015. “Executing Code off the Machine, or: Non-Deterministic Processes Don’t Have to Give You a Heart Attack.” *Esoteric.Codes Blog*, September 29. Accessed August 16, 2017. http://www.electronicbookreview.com/thread/electropoetics/net.writing.

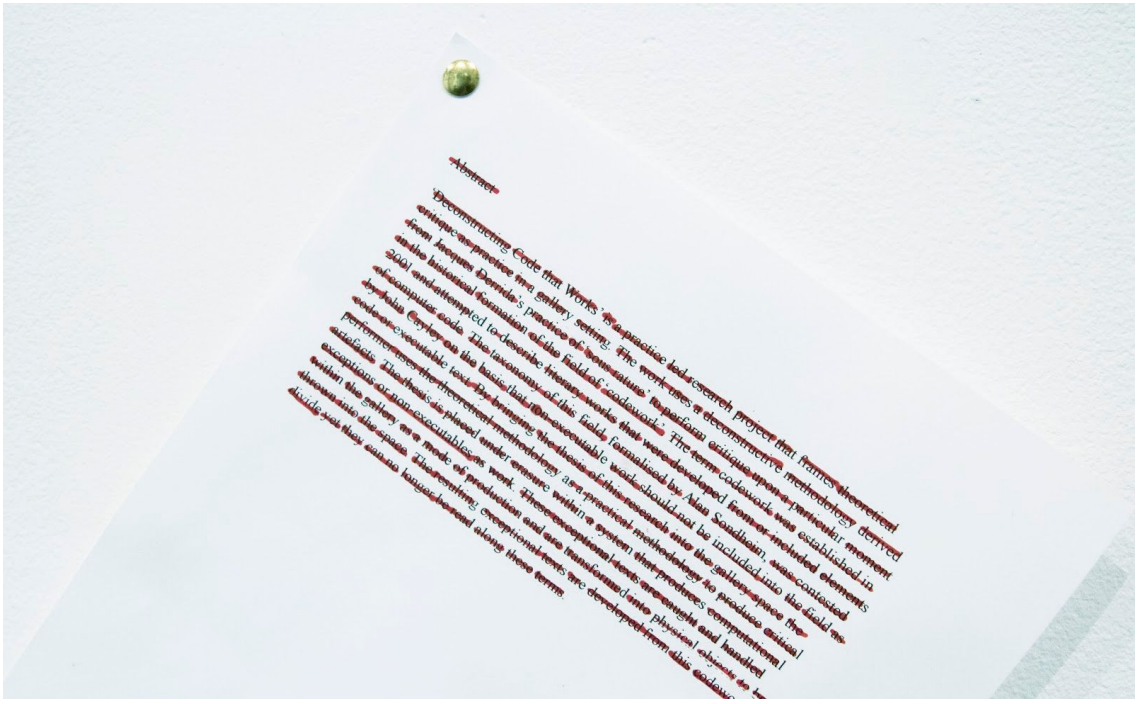
Temkin, Daniel. 2017. “FatFinger.JS.” *FatFinger.JS Website*. Accessed August 16, 2017. http://fatfingerjs.com.

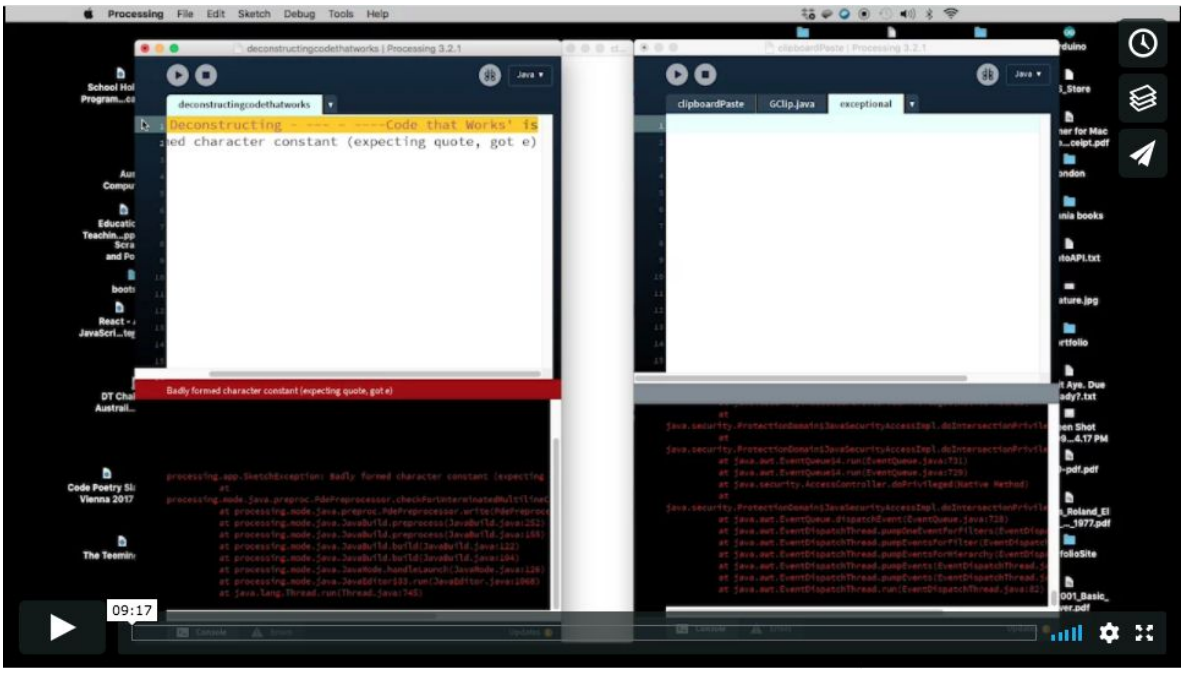
CHAPTER 3

Documentation of Exhibition

Abstract Introduction







Deconstructing Code that Works Process: https://vimeo.com/258715130

Gallery