SOURCE CODE ANALYSIS AS TECHNICAL ART HISTORY

DEENA ENGEL[1], AND GLENN WHARTON[2]

[1] *Department of Computer Science, Courant, Institute of Mathematical Sciences, New York University*
[2] *Museum Studies Program, New York University*

As part of its program to conserve software-based artworks, the Museum of Modern Art undertook a risk analysis of 13 works that use a variety of software programs, programming languages, and code libraries. Risks assessed in this study include the potential impact due to changes and upgrades to hardware, operating systems, and programming languages that would render the software obsolete. The assessment made clear that one of the museum's primary conservation strategies should be building technical documentation about the artworks. Consequently, the museum undertook a second project to build documentation about the software and hardware dependencies for 13 software-based works. While analyzing artist-rendered source code, the researchers in some cases discovered hidden information about the working methods of the artists and their programmers. This information includes the development of aesthetic properties such as color, movement, and sound. The discovery of these clues to the artists' concerns broadened the scope of the research to include an argument for source code analysis as a tool for technical art history. In this article the authors describe the potential for adapting documentation methods from software engineering for conservation purposes, and further argue for using these methods in art-historical research.

KEYWORDS: *Computer, Digital art, Documentation, Media Conservation, Software, Software-based art, Source Code, Technical art history*

1. INTRODUCTION

The Museum of Modern Art (MoMA) launched a program in 2005 to assess risks associated with its media art collections. Among the most vulnerable media works identified in the assessment were those driven by artist-generated software. A primary concern is that commercial hardware and operating systems frequently change, requiring active management of the software to keep it functional. In 2009, the museum launched a project to build technical documentation for the software-based works in the collection. This research, designed to aid in conservation strategies, included artist and programmer interviews, research on the hardware, software, and media components of the artworks, and technical documentation of the source code.

Knowing that MoMA staff did not have all of the expertise or time needed for the project, the media conservator developed partnerships with three programs at New York University (NYU) to aid in conducting the research: the Computer Science Department of the Courant Institute of Mathematics, the Moving Image Archiving and Preservation Program, and the Museum Studies Program. The project included 13 artworks. Under the guidance of MoMA staff and university faculty,[1] students interviewed 11 artists and 2 programmers.

Following these interviews, three artworks were selected for an additional research project with computer science students at the Courant Institute and graduate students in Museum Studies to investigate the potential of adapting models from software engineering to create technical documentation of the source code. The aim of creating this documentation was to provide information for future programmers to recompile or re-write the code for new operating environments.

The initial findings of this project were published in an article, "Reading between the lines: source code documentation as a conservation strategy for software-based art" (Engel and Wharton 2014), that describes three software engineering models used to document source code (see Appendix 1 for artwork descriptions). The original aim of the present article was to report the potential for a fourth method of code documentation, UML (Unified Modeling Language) diagrams. Yet during the project, the authors came across a new finding that altered the course of the research and broadened the scope of the study. Technical research on artist-generated source code not only serves conservation, but it can also aid art-historical research on artists' aesthetic aims and their working methods.

DOI: 10.1179/1945233015Y.0000000004

Technical studies in the service of art history are not new, but their application to software-based art is novel. The original framing of art conservation as a science was an effort to better understand mechanisms of material deterioration in order to prevent further decay and provide conservators with material knowledge to aid their cleaning, repair, and aesthetic integration strategies. For over a century, this scientific research has increasingly been combined with traditional connoisseurship to better understand the materials and techniques used by artists in the production of their work. Since the founding of the Doerner Institute and first museum conservation laboratory in late-nineteenth-century Germany (Clavir 2002) and the pioneering research at Harvard University's Fogg Art Museum in the early-twentieth century (Bewer 2010), a number of museums have created conservation science laboratories to research their collections. More recently, universities have developed courses in technical studies, and there is a growing body of literature on the results of technical investigations that some now refer to as *technical art history* (Ainsworth 2005; Considine 2005; Hermens 2012; Hill Stoner 2012). Technical knowledge of artworks facilitates an understanding of aesthetic intentions, of opportunities and limits afforded by methods and materials of production, and of broader questions about creativity and the nature of creative processes.

As we demonstrate in this article, technical research of software-based artworks provides many of the same benefits to art historians and conservators. Our aims in this article are to describe the technical research conducted by MoMA and NYU, summarize the findings that now include the use of UML diagrams in documenting source code, and argue for code analysis as a new form of technical art history.

2. Source code Documentation as a Conservation Strategy

2.1 software maintenance

Programmers write source code in specific programming languages. It is typically stored in a text file so it is human-readable. Like any language, those who know how to read it understand source code. Just as spoken languages are comprehensible to people who know their grammatical rules and vocabulary, a computer program in Java can be read and understood by computer programmers who have studied and worked with Java. Programming languages are born and fade at a much faster pace than spoken languages. As programming languages are replaced and become extinct over time, new programmers will not be trained in reading them.

As reported in the first publication from this study, technical documentation of source code will provide valuable information to future programmers to understand how artist-generated software programs work. They will be able to use the documentation to maintain the software when there are changes to the underlying operating system and/or to the hardware. For works of art that include multi-media items, technical documentation will also be helpful when there are changes to relevant audio, video, and image formats and standards.

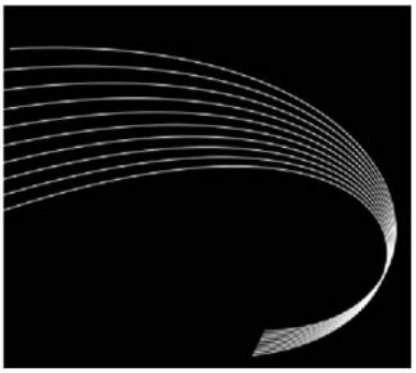
Software maintenance is a standard practice in software engineering that seeks to correct and improve upon computer programs over the time that they are in use, and technical documentation of a program's source code is a core activity of this discipline (Ali 2005; Das et al. 2007; de Souza et al. 2005; Scanniello et al. 2010; Stroulia and Systä. 2002; Wong and Tilley 2002). Various methods of documentation have been developed to provide programmers with tools for software maintenance (Correia et al. 2010; Forward and Lethbridge 2002; Huang and Tilley 2003; Tilley 2009; Trese and Tilley 2007). In the first phase of our research, we demonstrated the use of three of these methods for museum use: code annotation, narrative description, and visual documentation. Figures 1–4 illustrate these methods from our research, along with UML diagrams, a fourth method that was investigated in the second phase of the study.

2.2 code annotation

Figure 1 depicts an example of documentation in which comments were added to the source code itself. Such annotation can be added to specific statements, blocks of code, or other structural elements within the program. The source code in black font within the text box is compiled into software that transmits information to the computer to produce the image seen on the computer screen. The text in the blue font (following the double-slash (//) in the text box) is technical documentation that will assist future programmers in deciphering the code. The computer does not use these comments embedded in the source code when the program runs.

2.3 narrative description

Figure 2 illustrates an example of narrative documentation that describes the software application as a whole, or specific aspects of how the software behaves. This narrative excerpt was written by Ben Wagle, a computer science student at NYU as part of our documentation of *Thinking Machine 4* by Martin Wattenberg and Marek Walczak. It provides the reader with a description of a method (block of code) in which a "chess piece" is rendered.



```
void setup() {
  size(640, 360);  // set up the window size for the drawing
  stroke(255);     // the lines will be drawn in white
  noFill();        // shapes may be drawn but not filled in
}
// Note: This drawing is responsive to the user as he or she moves the mouse.
void draw() {
  background(0);   // the background is set to black
  // a "for loop" signals a repeating pattern
  for (int i = 0; i < 200; i += 20) {
    //  This line draws a Bezier curve which is responsive to the user's mouse
    bezier(mouseX-(i/2.0), 40+i, 410, 20, 440, 300, 240-(i/16.0), 300+(i/8.0));
  }
}
```

FIG. 1. This drawing of a Bezier curve is a sample from the Processing website from the "Examples" section which supplements the "Tutorials" section on the site. The code in black font in the text box generates the drawing, while the annotation in blue font describes the function of the code. When the program runs to render the drawing, the resulting design is responsive to the user; it shifts and changes as the user moves his or her mouse. This animated behavior is clear from the code but not from the still image as it is captured on a printed page. Source: http://processing.org/examples/bezier.html (accessed March 23, 2014).

ChessPieceGraphics.java:

This file contains all the dimensions and functions necessary to draw the actual chess pieces during the games of chess. What is notable to mention is that the pieces are actual geometrically constructed shapes that the artist created himself. They are not images that were saved in a file and read into the program. This makes maintaining the program very easy, as there are no image files to keep track of. Working with files from the Think.Chess package, ChessPieceGraphics.java defines a scale (pieceScale= .75); and then calls the corresponding draw function in order to draw the different pieces that make up a set in chess.

FIG. 2. Example of *Narrative Description* documentation of source code from *Thinking Machine 4* by Martin Wattenberg and Marek Walczak. In this excerpt, the author describes the purpose of this specific program (ChessPieceGraphics.java) and the importance to the work (in this case, to be sure that the reader understands that the "chess pieces" in *Thinking Machine* are dynamically drawn and not taken from static image files such as .GIF or .PNG files.) Further, the author clarifies a scale that is used (.75) which will be applied to the drawing program which would be called to actually render the "chess pieces." (See fig. 6 for images of some of the rendered "pieces.")

2.4 VISUAL DOCUMENTATION

In some cases we found that using charts and diagrams provided the best way to succinctly describe the structure and components of the software that we studied. The program structure for *33 Questions Per Minute* is relatively complex. The diagram in fig. 3 is a flowchart that describes the order in which specific sections of the computer program are run to process the data and produce the output. This diagram was generated collaboratively by five NYU computer science students who worked on this project and were members of our research team: Susana Delgadillo, Kelsey Lee, Liz Pelka, Erin Schoenfelder, and Albert Yau.

2.5 UML DIAGRAMS

In the second phase of our study, we investigated the use of UML (Unified Modeling Language) diagrams as a fourth method of documenting source code. UML diagramming is a standard approach used to create a visual representation of the components and aspects of a software application (Schattkowsky et al. 2005; Gray et al. 2010; Flint et al. 2004). UML (http://www.uml.org/) is an open source notation that was developed in 1995

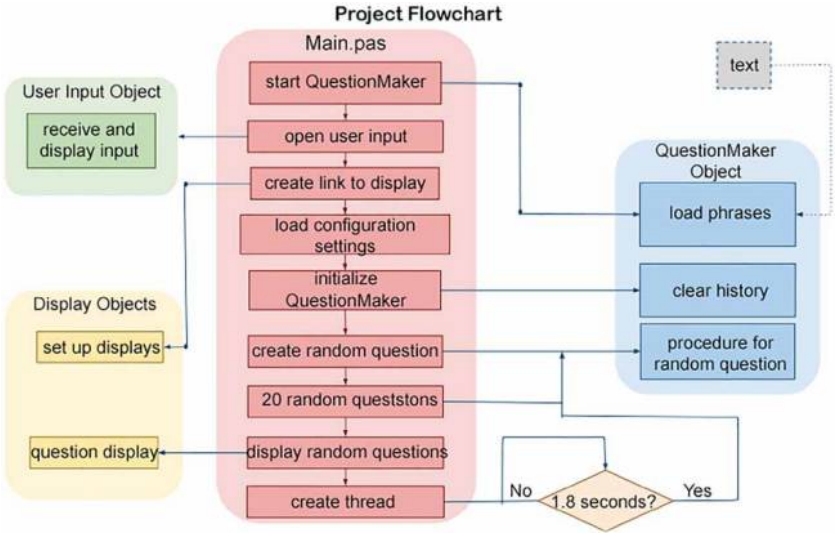


FIG. 3. A simplified version of the flowchart for *33 Questions Per Minute*. All of the important program file groups are listed in relation to when they are actually called or run within the work of art. The program begins with *main.pas* and a programmer can see from this chart which steps are executed next by following the arrows. In some cases, such as the wait time in the pale diamond on the lower right, a condition must be met before the next program is called.

and is independent of any specific programming language or programming approach. UML offers a formal and widely known methodology for describing even large and complex software applications in a succinct and visual way.

We introduced UML diagrams into our study with *Thinking Machine 4*, which is written in Java. In this case study, we used UML to focus on the use of object-oriented programming in Java, which is a programming paradigm that allows the developer to model elements and behaviors within the program in a coherent way. Further research could be done to determine how and whether conservators would benefit from learning UML to participate in the documentation process. The small excerpt in fig. 4, developed by NYU computer science students Anthony Spalvieri-Kruse and Ben Wagle, is from the UML diagram of *Thinking Machine 4*.

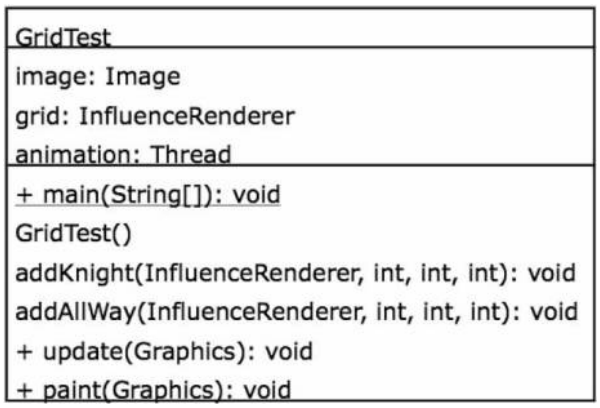


FIG. 4. An example of an excerpt from a UML diagram for *Thinking Machine 4*. This excerpt describes a class, or block of code as it is used in Java, that the artists used to test whether the chessboard and wave motion were working properly. The top line refers to the name given to this specific *class*, which in this case is the descriptive term "GridTest" (a programatically "safe" way to spell the name "grid test" or "testing the grid"). The middle section of the box contains information about what is included in this class and how that information is represented. The bottom section lists the permitted operations as a way to describe the behaviors for this class.

2.6 SUMMARY OF TECHNICAL DOCUMENTATION FINDINGS

Each of these four approaches can assist conservation in different ways. Code annotation provides information for future programmers who may be asked to migrate, update, or re-compile it for future display. This method offers a high level of detail, including line-by-line documentation that is beneficial to future programmers. Narrative documentation, either in an external document or stored within the source code as lengthy comments, provides a succinct description of a specific block of code. Visual diagrams such as flowcharts and UML diagrams give future programmers an overview of the system as a whole, and how different aspects of the software work together.

There are additional ways in which software documentation can assist conservators. For instance, a good study will point out weaknesses in the system that could lead to future problems: we found a "bug" in *Thinking Machine 4* which causes the system to stop running if specific game sequences are "played."

The code documentation process highlights files such as data sets, multi-media files, drivers for hardware, external code libraries used to supplement the artist/programmer's code, and other required files that are uncovered as the source code is studied. This helps conservators make sure that the museum has obtained all of the files and data required to exhibit the work and conserve it properly. Software documentation also aids the conservation process by providing specific information about the work from a technical standpoint. This is information that is not necessarily evident when observing the artwork visually.

It is important to keep in mind that software engineers and art conservators have different objectives in their documentation strategies. The aim of software engineering maintenance is to sustain the functionality of software until it is upgraded to a new version, or replaced altogether. The aim of software art conservation is to maintain the artist's vision at the time the software was created for future public experience of their work. This may include software upgrades and replacement, or it may require preservation of original software and/or hardware environments.

Media conservators must navigate between technical possibilities and artist concerns. Often artists have not fully thought through the implications of future conservation, and the conservation interview is the first time they attempt to articulate their thoughts. Media conservation, like other areas of conservation, involves careful research and discussion in order to negotiate strategies that may sometimes alter the work in the name of preservation. As in other areas of the discipline, conserving software-based art requires managing change, with full documentation of decision-making rationale for future scholars and decision makers.

3. Analysis of Research Findings

Based on our research, we found that there are multiple benefits from documenting the source code of software-based art. These benefits can be divided into three broad categories that are discussed in the sections below

- documentation for software maintenance;
- documentation to understand aesthetic intent of the artist; and
- documentation of unused code to understand the artist's working methods.

3.1 documentation for software maintenance

In our study, we documented various aspects of the software application within the operating environment to help maintain the software. Among these aspects of the software application are the role of systems configuration files, the specific interface to the operating system, how the work of art addresses specific and/or customized hardware, and how the software manages and uses libraries and multi-media file formats. These aspects of the software are relevant to many software applications and are not specifically related to the artistic nature and aesthetic goals of software art. However, we documented these aspects of the software with respect to aesthetic implications of the artist/programmer's decisions. The following examples illustrate how standard software documentation strategies applied to the conservation of artworks in our study.

3.1.1 system configuration

Some artists/programmers use systems configurations data to prepare a work for re-exhibition or to exhibit the same work of art in different environments or in different ways. For example, *33 Questions Per Minute* runs in English, German, or Spanish. A configuration file is used to store the language setting (among other parameters), which is determined at the time of installation and is respected throughout the software application so that the program appears to run only in English, German, or Spanish. All of the vocabularies required in each language are included in the setup files and the program is written in such a way as to create grammatically reasonable (if semantically nonsensical) questions in each of the three languages within one set of source code and installation files. Therefore, the artist/programmer can deliver the same software program for exhibition regardless of where it will be shown and the language (English, German, or Spanish) can be selected as part of the installation and setup process at the time of each exhibition.

3.1.2 operating system specific issues

We found that the artists at times used features of the underlying operating system or programed on the operating system level in order to render their works in specific ways. For example, *Shadow Monsters*, which can crash if there is too much activity to process (such as a number of active children creating "monsters" at the same time), uses Mac-Unix scripts to restart when necessary; this is nearly transparent to the viewer.

3.1.3 custom and/or unusual hardware

The software written to output the text to multiple screens for *33 Questions Per Minute* is hardware-specific and an examination of the source code and a review of the drivers was needed to understand how the hardware was addressed in this work. There are also examples of other works of software art in MoMA's collection that use customized, unique hardware built specifically for the work of art.[2] In our earlier work, we studied the importance of documenting the operating environment for each artwork,

including the operating system and specific version, the programming language and specific version, and specific configurations in the hardware requirements such as the minimum amount of RAM needed and other issues (Engel and Wharton 2014). In reviewing the source code, we found it important to review and confirm any specific hardware requirements so that the software would run correctly. This information further informs conservation decisions.

3.1.4 SOURCE CODE FROM SPECIFIC LIBRARIES

From a conservation perspective, we discovered code libraries are of concern to the museum during acquisition as well as exhibition. Programmers often use libraries that are collections of pre-written code which programmers can re-use; these collections are sometimes written by a third party and typically not included in the programming language itself. Different libraries may be written to meet specific needs of different programmers. For example, a physics library typically contains code to model physical effects such as rendering the "bouncing" motion of a spring, the pattern of a wave, how hair might move in a breeze and other effects. Phil Worthington used a physics library for special effects in *Shadow Monsters*, but the source code for the physics library was not included with the rest of the source code provided to the museum. *Shadow Monsters* also uses a sound library to facilitate rendering the sounds of the "monsters" (the grunts, groans, and squeals). The source code from these libraries would be required if one were to update and re-compile the program at a later date, but it was not included at the time of acquisition and no longer appears to be available from other sources. It was only through the process of documenting the source code that we were able to ascertain which libraries were used and thus determine which libraries or code modules were present or missing. When a code library is available for study, it is important to determine whether the artist/programmer had further modified that library. If the artist/programmer modifies a library, then future upgrades or modifications would require that the upgraded library reflect the artist/programmer's original intent or changes. This is a vital aspect of conservation for a work of software art.

3.1.5 MANAGING AND UNDERSTANDING THE USE OF MULTIMEDIA FILES

As we described in our earlier work, it is important that museums acquire uncompressed file formats for multi-media components of artworks (Engel and Wharton 2014). In this study, we analyzed the source code for further information about multi-media files and how they are used. For example, in *Shadow Monsters*, the sound files are located within the hierarchy of folders designated to organize files for a variety of sounds including titles such as "high sounds," "low sounds," "burps," and others. The artist uses a library to play the selected sounds that are stored as *.aif* files.

3.2 DOCUMENTING AESTHETIC INTENT OF THE ARTIST

In addition to analyzing source code for software maintenance, we found that important aesthetic information is often written into the code. The research allowed us to study the work of art through topics such as the use of color, randomization, speed (for animated images), and the construction of images (programatically or from external files). Thus, documenting the source code allowed us to better understand the aesthetic intent of the artist in the following ways.

3.2.1 COLOR

The study of pigments and dyes plays a significant role in the technical analysis of traditional artworks such as paintings, drawings, and textiles. Colors in a digital world are defined by a series of numerical values and described in terms of color space. For example, both of the *Processing* works that we studied use *RGB*: the red, green, and blue additive color model used for electronic systems. In the case of software art, the colors used are clearly defined in the source code. In MoMA's *Thinking Machine 4* the colors used when the program is "thinking" about which "move" to make next are derived from a range of numerical values which are modified programatically to render the series of yellow and orange hues for one player's "moves" as seen in the arches that are drawn. The other player's "moves" use another set of numerical values for the arches in differing shades of green. By examining the source code, one can identify and (thereby reproduce if necessary) specific colors that are used.

3.2.2 SPEED (FOR ANIMATED WORKS)

An examination of the source code reveals information about the relative speeds at which an artwork will run as well as the artist's intent as to the speed and periodic or occasional variations in the speed. These aesthetic qualities of the work are not easily understood by simply observing the work as a viewer.

We found in studying the source code for *Shadow Monsters* that the artist used several programming techniques to simulate a pause in movement at various times. However, by examining the code, it became clear that due to the way in which these pauses are written, they could disappear if the work were run on a computer with a faster processor. This aesthetic consideration was documented for future re-exhibition.

In another example, in the case of *Thinking Machine 4*, we learned from the source code that the amount of "thinking time" allotted for the computer's "turn" is in

fact randomly determined within a specific range of values. In this case, the viewer's perception that the computer "thinks" for a different amount of time at each move is correct, and the time allotted for each "move" is unpredictable. However, the length of duration of the computer's "turn" is expressed as a length of time (measured in seconds). In this case, a change in the hardware such as a faster processor should not impact the viewer's experience of the work.

3.2.3 RANDOMIZATION AND AESTHETIC CONSIDERATIONS

It is often difficult to ascertain from viewing a work of software art whether, where, and how randomization is used in the execution of the software. However, an analysis of the source code clarifies the use (or absence) of randomization.

We saw above that the duration of time that the computer "thinks" between "moves" in *Thinking Machine 4* is based on a randomly generated value. In other cases, aspects of a work appear to be random but actually follow a specific and predictable underlying pattern. For example, the groans and grunts of the "monsters" in *Shadow Monsters* give an impression that they are randomly selected but in fact, the source code specifies the order in which specific sounds are played as well as which sounds are played for which shapes (by evaluating the size of each monster's "mouth").

In yet a different example, the words and phrases in *33 Questions Per Minute* are randomly selected from a carefully planned series of text files that reflect the given language's grammatical structures (English, Spanish, or German). In this way, a grammatically coherent statement is built from a series of randomly selected words or phrases organized to ensure a grammatically correct question, even though the result is intended to be nonsensical.

We saw above that the amount of time that the computer "thinks" before each chess "move" in *Thinking Machine 4* is randomized. However, to our surprise, we learned from the source code that the specific chess games that are "played" in MoMA's *Thinking Machine 4* are not random at all or even generated at the time of play. The work is driven by detailed scripts that execute in the same order with the same result each time the game is "played."

3.2.4 USE OF IMAGES

Although it was not visually clear to us before our study, we learned from the source code that the shapes that are used for "teeth" and "hair" in *Shadow* Monsters are derived from a collection of .PNG image files such as those in Figure 5. These files were provided to the museum along with the source code and must be conserved along with the rest of the digital files that comprise this work.

FIG. 5. Three examples of image files that are used to render "teeth" in *Shadow Monsters*. Each of the shapes in this figure is meant to be a single "tooth" in *Shadow Monsters* and is stored in a single .PNG file. These files were supplied to the museum by the artist upon installation. For example, the center "tooth" is from a file called *tooth_canine.png*. When *Shadow Monsters* is running, this image would be displayed repeatedly along a contour, to resemble a line of "teeth" in that specific monster's "mouth."

In *Shadow Monsters*, a selected image, such as one of the three in Figure 5 is displayed in a repeating pattern along the contour of the monster's "mouth" to designate a row of "teeth" or along the monster's "arm" or "head" to designate "hair". The .png filenames are prefixed with "tooth_" or "skin_" to differentiate how they are used.

Thinking Machine 4 however renders the chess pieces programatically and dynamically during the "game" by "drawing" the images rather than using stored image files (fig. 6).

From a conservation perspective, different approaches are required depending on how the images are rendered. In the case of Shadow *Monsters*, the original image files must be appropriately conserved. In the case of *Thinking Machine 4*, the source code used to render the images must be appropriately maintained in order for the work to run in the future.

We studied the dynamically generated shapes and images in the source code of *Thinking Machine 4* in detail in order to better understand the potential for variability in shape and color, as well as for changes that might occur when the "chess piece" "moves." Studying the source code gave us further insight into how the artist/programmer envisioned the "chess pieces" as it is clear in the source code how the images are algorithmically developed.

FIG. 6. "Chess pieces" in *Thinking Machine 4*. The five designs in this figure depict "chess pieces" which are dynamically drawn when *Thinking Machine 4* runs. In this case, the software "draws" each chess piece during the execution of the program; there are no separate image files, such as those in fig. 5, to store or display.

Finally, we found many examples in which source code analysis reveals information about the artist's process and the development of the work of art. These are aspects of software that would likely not be examined or documented at all for most software engineering applications.

3.3 DOCUMENTING UNUSED CODE TO UNDERSTAND THE ARTIST'S WORKING METHODS

A third category of benefit from source code documentation derives from analysis of unused code. Unused code can contain earlier attempts to accomplish a specific task or effect, the information that a programmer needed for testing during development, comments about the development, or comments containing other information useful to the programmer during the development process. It is not necessary to remove unused code or comments for the application to run, so programmers sometimes leave unused code and comments in the source code. We have found that segments of source code that have been "commented out" or which do not execute can provide information on a work of software art that is analogous to obtaining preliminary sketches of a painting, the underlying charcoal drawing on a canvas, or a *grisaille* under the pigmented glazes in an oil painting. We found that in examining the unused source code, along with data sets and configuration values that the artist/programmer used for testing, we can learn about the artist's process and aesthetic intentions.

In *Thinking Machine 4* for example, the artists had set up several "chess boards" programatically. In one scenario, they considered a blue background, using two shades of blue for the board itself and a blue theme. In another scenario, the artists used shades of ivory and brown in combination before they settled on the chessboard colors that we currently see.

We found many examples of residual evidence of the artist/programmer's experiments in the no-longer-used source code. For example, sample data sets that the artist used for testing allow us to better understand how the artist visualized a work before exhibition. *Shadow Monsters* includes a number of "test values" that impact how the program runs according to what the artist anticipated and wanted to see. *Thinking Machine 4* contains a series of values with comments to determine the width of the arches, the specific color mixtures, their orientation on the board, and other visual effects that the artists apparently experimented with.

The parameters set in *33 Questions Per Minute* include designating information about the output (font and font size for example). Our study of the configuration file and the variations in the settings helped us to better understand those aspects of the artist's process of decision-making.

Some comments in the code reflect the artist's notes while writing the program, such as this one in *Thinking Machine 4:* `//System.out.println("The game is afoot!")`. To us, finding this artist-embedded comment that "the game is afoot!" was like discovering Jackson Pollock's fingerprint pressed into a running drip of paint.

Although much can be revealed in unused source code, it is unfortunately not always available to researchers. For example, if the work is written in a language such as Java that gets compiled into software, all of the code that has been "commented out" is removed at the time of compilation. In such a case, if the museum has not acquired the artist's set of source code files, a later conservation intervention might require that the Java executable file get "de-compiled" which would result in source code but not the original comments or commented-out code. It is also possible that an artist/programmer might deliberately remove comments before giving the work to a museum. In these cases, the analysis that we describe in Sections 3.1 and 3.2 would still apply. However, from a research perspective, this is a further argument to support a museum's efforts to obtain the artist's complete collection of files that comprise the source code so that researching the artist's working processes and aesthetic intent might be available. When researchers are lucky enough to find buried comments and unused code, they are likely to learn about the artist and his or her creative process.

4. Conclusions: Source Code Documentation as Technical art History

Learning about the creative development of the works we studied and the artists' efforts to "get it right" from source code documentation came as a surprise to us. As mentioned at the beginning of the article, our original aim was to develop source code documentation strategies for software maintenance purposes. We set out on this study with two primary goals in mind.

First, we examined the benefits and drawbacks of different types of documentation used for software maintenance in software engineering. We found that our results matched those of the software engineering field in that the documents and data we produced provided valuable tools for interpreting the source code. The four documentation techniques that we used were helpful in different ways to aid future programmers in understanding the system configuration, relationships with the operating system and hardware, and use of code libraries and multi-media files.

Second, we aimed to differentiate and "fine-tune" software documentation techniques from engineering applications for application to software art. Here we found a number of areas of study where the aesthetic aspects of the art, and thus the goals of art conservation differ from standard technical documentation studies of industry software. We found that the inter-disciplinary nature of our group – comprising participants with expertise in Computer Science, Museum Studies, and Conservation – was especially helpful in assessing documentation requirements for software art. Specific concerns for artworks in museum collections include systems and configuration issues, along with aesthetic components of the work that must be documented for conservation and re-exhibition.

One of our more interesting realizations in comparing the aims of conservation and the aims of software maintenance is the difference in importance placed upon *reproducibility*. With respect to the conservation of software art, reproducing the aesthetic experience and conveying the artist's intent are of paramount importance. It is less important to the aims of software engineering. Of course the question of reproducibility is important across the sciences and in mathematics where the goal is to ensure that computational results can be repeated in the future. Setting and achieving appropriate standards for reproducibility in computation poses a number of interesting technological challenges. Recent research in this area could have application to conservation of software art, but researching this potential was beyond the scope of this study (Stodden et al. 2013).

In Sections 3.2 and 3.3, we demonstrate how source code documentation aids conservation in understanding the aesthetic intent of the artist and in learning about their working methods. It is not a great leap to see how this knowledge also informs art-historical research. Our digital archaeology expeditions into the core structure of the artworks revealed twists and turns of design. We learned about experiments in color and sound as the artists and programmers worked to achieve their desired effects. We also discovered their concerns for speed in animation, and their efforts to create an appearance of randomness with carefully generated values and predictable underlying patterns. What started with a quest to understand the technology for preservation purposes ended by opening the potential for a new form of research about how creative minds work. This sort of discovery is not new for conservators and conservation scientists. In fact, art historians have regularly benefitted from conservation studies of past artistic innovations, from early metal casting technologies to experimentation with drying oils and early photographic processes. We hope that the rich trove of hidden information we discovered embedded in source code will continue the tradition of conservation research benefitting broader art-historical understanding about creativity in the age of digital production.

Acknowledgments

This article could not have been written without the approval and encouragement from the artists in our study: Rafael Lozano-Hemmer, Martin Wattenberg, Philip Worthington, and Marek Walczak. They allowed us access to their source code for the purpose of our research. We also thank the students at New York University who conducted the code analysis and developed the documentation: Susana Delgadillo, Howard Jing, Kelsey Lee, Daniel Ng, Liz Pelka, Chris Romero, Erin Schoenfelder, Anthony Spalvieri-Kruse, Ben Wagle, and Albert Yau. We are grateful to our colleagues at MoMA who joined in our efforts, including chief conservator Jim Coddington, Kate Carmody, and Paul Galloway of the Architecture and Design Department, assistant media conservator Peter Oleksik, digital repository manager Ben Fino-Radin, chief technology officer Juan Montes, and James Heck, director of infrastructure. The article benefitted from the generous comments of Jim Coddington, Joshua Clayton, Mark Hellar, Joanna Phillips, and Alexander Wharton.

Appendix 1

The source code for the following three artworks was documented in this study. Technical descriptions of the artworks are provided in this appendix rather than the body of the text since knowledge of all components of the works is not necessary to understand the findings and analysis of the study. The authors encourage interested readers to learn more about these works by visiting the websites referred to for each work.

33 Questions Per Minute by Rafael Lozano Hammer, 2001–2002

http://www.moma.org/collection/object.php?object_id=102431 (accessed March 22, 2014)

http://www.lozano-hemmer.com/33_questions_per_minute.php (accessed March 22, 2014)

33 Questions Per Minute is a work of text art that has been exhibited in English, Spanish, and German versions. This work of art is typically exhibited on 21 small LCD screens, with a keyboard available for public interaction in the gallery. Sentences are randomly generated on the screens at a speed of 33 per minute, a rate which the artist describes as the fastest presentation that remains readable to the viewer.[3] The software was written using *Delphi*, which is a derivative of Pascal. *Delphi* is a proprietary language, currently owned by Embarcadero Technologies.

Shadow Monsters by Phil Worthington, 2004-ongoing

http://www.moma.org/collection/object.php?object_id=110196 (accessed March 22, 2014)

Shadow Monsters is an interactive multi-media work in which animal-like shapes are generated from viewer's movements and projected on a wall in the gallery and accompanied by animal-like sounds. The source code is written in *Processing*, a programming environment that uses Java and was

designed for visual artists (Reas & Fry 2007). Worthington employed pre-written program files called *libraries* for specific tasks such as playing the sounds; and for emulating special effects such as spring-like movements, "blowing hair," and other visual aspects of the work.

Thinking Machine 4 by Martin Wattenberg and Marek Walczak, 2003–2004

http://www.turbulence.org/spotlight/thinking/ (accessed March 22, 2014)

Written in *Processing* (which uses Java), *Thinking Machine 4* is a game of chess displayed on a screen in the gallery. Prior to each move, a ripple effect followed by orange and yellow or green curved lines (depending on which "side" has the current "turn" to "play") appear on the chessboard emanating from each chess piece, to indicate some of the many possible moves at that moment in the game as though to reveal how the computer "thinks." Wattenberg used customized *Processing* libraries in writing the source code. MoMA's version is not interactive, but the artists created a version that can be played online: http://www.turbulence.org/spotlight/thinking/chess.html (accessed March 22, 2014).

Notes

1 MoMA staff and NYU faculty who supervised and advised on the project included the authors as project directors, and Howard Besser, Director of the Moving Image and Archiving Program at NYU; Mona Jimenez, associate director of the Moving Image and Archiving Program at NYU; Peter Oleksik, assistant media conservator at MoMA; and Ben Fino-Radin, digital repository manager at MoMA.

2 An example of a work at MoMA that uses customized software is *I Want You to Want Me* by Jonathan Harris and Sep Kamvar, 2008. Accession number SC527.2008. http://iwantyoutowantme.org/ (accessed 03/22/14).

3 From interview with the artist conducted at MoMA by Sarah Resnick, Barbara London, and Glenn Wharton on June 27, 2006.

References

Ainsworth, M. W. 2005. From connoisseurship to technical art history: the evolution of the interdisciplinary study of art. *The Getty Conservation Institute Newsletter* 20 (1): 4–10. <http://www.getty.edu/conservation/publications_resources/newsletters/20_1/feature.html> (accessed March 22, 2014).

Ali, M. R. 2005. Why teach reverse engineering? In *SIGSOFT Software Engineering Notes* 30(4): 1–4.

Bewer, Francesca B. 2010. *A Laboratory for Art: Harvard's Fogg Museum and the Emergence of Conservation in America, 1900–1950*. New Haven: Yale University Press.

Clavir, M. 2002. *Preserving What is valued: museums, conservation and first nations*. Vancouver, BC: University of British Columbia Press.

Considine, B. 2005. Recent initiatives in technical art history. *The Getty Conservation Institute Newsletter* 20(1): 21–24. <http://www.getty.edu/conservation/publications_resources/newsletters/20_1/news_in_cons2.html> (accessed March 22, 2014).

Correia, F. F., A. Aguiar, H. S. Ferreira, and N. Flores. 2010. Patterns for consistent software documentation. In *Proceedings of the 16th Conference on Pattern Languages of Programs (PLoP '09), 28–30 August 2009*. New York: ACM. Article 12, 7 pages.

Das, S., W. G. Lutters, and C. B. Seaman. 2007. Understanding documentation value in software maintenance. In *Proceedings of the 2007 symposium on Computer human interaction for the management of information technology (CHIMIT '07), 30–31 (March)*. New York: ACM. Article 2.

de Souza, S. C. B., N. Anquetil, and K. M. de Oliveira. 2005. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information (SIGDOC '05) 21–23 September 2005*. New York: ACM. 68–75.

Engel, D., and G. Wharton. 2014. Reading between the lines: source code documentation as a conservation strategy for software-based art. *Studies in Conservation* 59: 404–415.

Flint, S., H. Gardner, and C. Boughton. 2004. Executable/translatable UML in computing education. In *Proceedings of the Sixth Australasian Conference on Computing Education – Volume 30 (ACE '04)*, ed. R. Lister and A. Young. Vol. 30. Darlinghurst, Australia: Australian Computer Society, Inc. 69–75.

Forward, A., and T. C. Lethbridge. 2002. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the 2002 ACM symposium on Document engineering (DocEng '02) 08–09 November 2002*. New York: ACM. 26–33.

Gray, J., W. Jules, and A. Gokhale. 2010. Model-driven engineering: raising the abstraction level through domain-specific modeling. In *Proceedings of the 48th Annual Southeast Regional Conference (ACM SE '10)*. New York, NY: ACM, Article 1.2 pages.

Hermens, E. 2012. Technical art history: The synergy of art, conservation and science. In *Art history and visual studies in Europe: transnational discourses and national frameworks*. eds. M. Rampley, T. Lenain, and H. Locher. Leiden: Koninklijke Brill. 151–165.

Hill Stoner, J. 2012. Turning points in technical art history in american art. *American Art* 26(1): 2–9.

Huang, S., and S. Tilley. 2003. Towards a documentation maturity model. In *Proceedings of the 21st Annual International Conference on Documentation (SIGDOC '03) 12–15 October 2003*. New York: ACM. 93–99.

Reas, C., and B. Fry. 2007. *Processing: A programming handbook for visual designers and artists*. Cambridge, Massachusetts: MIT Press. 160–163.

Scanniello, G., C. Gravino, M. Risi, and G. Tortora. 2010. A controlled experiment for assessing the contribution of design pattern documentation on software maintenance. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '10), 16–17 September 2010*. New York: ACM. Article 52. 4 pages.

Schattkowsky, T., W. Mueller, and A. Rettberg. 2005. A model-based approach for executable specifications on

reconfigurable hardware. In *Proceedings of the conference on Design, Automation and Test in Europe – Volume 2 (DATE '05)*. Vol. 2. Washington, DC: IEEE Computer Society. 692–697.

Stodden, V., D. H. Bailey, J. Borwein, R. J. LeVeque, W. Rider, and W. Stein, eds. 2013. Setting the default to reproducible: reproducibility in computational and experimental mathematics. ICERM. http://icerm.brown.edu/tw12–5-rcem (accessed March 23, 2014).

Stroulia, E., and T. Systä. 2002. Dynamic analysis for reverse engineering and program understanding. *SIGAPP Applied Computing Review* 10: 1.8–17.

Tilley, S. 2009. Documenting software systems with views VI: lessons learned from 15 years of research & practice. In *Proceedings of the 27th ACM international conference on Design of communication (SIGDOC '09) 5–7 October (2009)*. New York: ACM. 239–244.

Trese, T., and S. Tilley. 2007. Documenting software systems with views V: towards visual documentation of design patterns as an aid to program understanding. In *Proceedings of the 25th Annual ACM International Conference on Design of communication (SIGDOC '07) October, 2007*. New York: ACM. (October): 103–112.

Tribe, M., and R. Jana. 2009. *New media art*. Köln, Germany: Taschen GmbH.

Wong, K., and S. Tilley. 2002. Connecting technical communicators with technical developers. In *Proceedings of the 20th Annual International Conference on Computer Documentation (SIGDOC '02) October 20–23, 2002*. New York: ACM. 258–262.

Author Biographies

Deena Engel is a clinical professor as well as the associate director of Undergraduate Studies for the Computer Science Minors programs in the Department of Computer Science at the Courant Institute of Mathematical Sciences of New York University. She teaches undergraduate computer science courses on web and database technologies, as well as courses for undergraduate and graduate students in the Digital Humanities and the Arts. She also supervises undergraduate and graduate student research projects in the Digital Humanities and the Arts. Prior to returning to academe, she ran a systems group in an international art auction house for 9 years. She received her master's degree in Computer Science from the Courant Institute of Mathematics at New York University. Address: Department of Computer Science, New York University, 251 Mercer Street Room 422, New York, NY 10012. Email: deena@cims.nyu.edu

Glenn Wharton is a clinical associate professor in Museum Studies at New York University. From 2007–2013 he served as Media Conservator at the Museum of Modern Art in New York, where he established the time-based media conservation program for video, performance, and software-based collections. In 2006 he founded INCCA-NA, the International Network for the Conservation of Contemporary Art in North America. He served as its first executive director until 2010. Glenn received his PhD in Conservation from the Institute of Archaeology, University College London, and his MA in Conservation from the Cooperstown Graduate Program in New York. His most recent book is titled *The Painted King: Art, Activism, and Authenticity in Hawai'i*, in which he tells the story of a community-based, participatory conservation project. Address: Museum Studies, New York University, 240 Greene St. Suite 406, New York, NY 10003. Email: glenn.wharton@nyu.edu