Original research or treatment paper

Reading between the lines: Source code documentation as a conservation strategy for software-based art

Deena Engel[1], Glenn Wharton[2,3]

[1]Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, New York, NY, USA, [2]Conservation Department, Museum of Modern Art, New York, NY, USA, [3]Museum Studies Department, New York University, New York, NY, USA

Conservation expertise required for software-based art varies depending on the nature and function of its components. Our focus in this study is technology, specifically related to the impact of changes and upgrades to the operating environment that can adversely impact future exhibition of software-based art. In our research to date, we found that each specific work requires individual analysis and conservation strategies due to unique technical risks. We also concluded that artist-generated source code is a primary risk for software-based works. We then devoted the next phase of our research to a closer examination of risks associated with source code. The purpose of the research reported in this article is to investigate whether examining and documenting the source code can inform conservation practice. A corollary second goal is to define relevant best practices for documenting source code for software-based art. In order to address these questions, we selected two artworks at the Museum of Modern Art for a collaborative study using students and faculty from both the Museum Studies and Computer Science departments at New York University. This collaboration helped ensure that technology skills complemented a deep understanding of art history in the museum context. We based the methodology for our study on current software engineering practices and composed diagrams and narrative documents to reflect what we found in the source code. We also relied on artist interviews to explore the requirements and goals of the system, and user manuals to assist in understanding the implementation and physical installation of the works. It was our hypothesis that once the behavior of software-based art is understood by combining a standard software engineering approach with considerations specific to artist and museum needs, conservators and programmers will be better prepared to address changes in the operating environment. Based on our experience, we found this to be true. We conclude this paper with plans for our next phase of research.

Keywords: Computer, Digital art, Documentation, Software, Software-based art, Source code

Software is the best way I've found to express myself. When I work in other media, the results somehow always seem worse in reality than in my head. The software I create, however, has a magical quality: it ends up being better than what I originally imagined (Reas & Fry, 2007).
– Martin Wattenberg

Introduction

Over the past 50 years, technology-based artists have written computer programs to produce a range of art forms. Early computer graphics, also known as algorithmic depiction (Lieser, 2009, p. 53) employed code to generate images such as plotter and inkjet prints (Wood, 2008). As computer technology evolved, many artists developed programs to create 3D images, animation, and virtual space worlds. A variety of terms are used to describe this expanding field, including digital born art, computer art, and computer graphic art (Tribe & Jana, 2009). Web, internet, .Net, or simply 'Net art' refers to works that can be rendered for exhibition on the Internet and thus overlaps with some of the other areas of digital-born art (Paul, 2007).

We use the term 'software-based art' in this article to refer specifically to works of art created by artists who

Correspondence to: Glenn Wharton, Museum Studies Department, New York University, 240 Greene Street, Suite 406, New York, NY 10003, USA. Email: glenn.wharton@nyu.edu

Received December 2012; revised paper accepted October 2013
DOI 10.1179/2047058413Y.0000000115

write computer programs to render their work. The software-based art that we study includes database art and art that stems from visualizations of data, text, algorithmic depiction, interactive technologies based on user input and/or environmental stimuli, and related areas. We use the term 'source code' to refer to the computer program which contains instructions for the computer. Source code is written in a specific programming language that is available in a human-readable form such as a text file.

Conservation expertise required for software-based art varies depending on the nature and function of its components. For instance, conserving images that are printed by a plotter or printer and conserving sculptures that are rendered by the use of a 3D printer fall under more traditional methodologies for conservation of paper and plastic, whereas conserving the code and associated data that were sent to the 3D printer would fall under the realm of software preservation. Net art requires expertise in conserving internet applications, digital archives, and interactive art: a broad territory that includes specialists in computer science, information science, library science, and time-based media art.

The conservation of software-based art includes the concerns of maintaining artist intentions for public experience and retaining conceptual integrity that are covered in the growing body of literature on conserving contemporary art (Wharton, 2005; Laurenson, 2006; van Saaze, 2009), but are not significantly addressed in this article. Our focus in this study is technology, specifically related to the impact of changes and upgrades to hardware that render the software obsolete; changes and upgrades to operating systems that render the software obsolete; changes and upgrades to programming languages and software applications used to create the artwork that render the software obsolete; changes and upgrades to specifications for any associated external multimedia files that are required to run the work; and risks to the files themselves with respect to bit integrity. Implications for good practice based on these risks include the acquisition of source code, software libraries that were used in the work, software licenses, copyright, archival source files for multimedia components, and rigorous backup and monitoring procedures.

The analysis and recommendations reported in this paper are part of a larger study of the technical risks associated with software-based art carried out since 2005 at the Museum of Modern Art (MoMA), when Glenn Wharton initiated a time-based media conservation survey of the collection. In 2009, Deena Engel led a risk assessment of three software-based works as part of the *Matters in Media Art* project funded by the New Art Trust. Since then, risk assessments were performed on 13 additional works. In our research to date, we found that each specific work requires individual analysis and unique conservation strategies due to specific technical risks.

After performing an initial survey of the collection, we conducted conservation risk analysis of three works (Appendix 1), each presenting a unique set of conservation risks. The first was John Maeda's *Reactive Books*, which is a series composed of five programs written in the programming language called 'C', each exhibited on its own Macintosh (an iMac running the Macintosh operating system OS/9) to render shapes, colors, animations, sounds, and other effects. The second is *I Want You to Want Me* by Jonathan Harris and Sep Kamvar, which culls data from the internet into a database that is used to provide an interactive experience including computer graphics, animation, photos, video, music, and data visualizations based on the database. The code is written in the C++ language. The third artwork is Teiji Furuhashi's *Lovers,* which uses sound and moving images to create a world in memory of those who have died of AIDS. Unfortunately the artist died an early death, and the code was not obtained by the museum.

To carry out these assessments, we developed a risk assessment form (Appendix 2), which allowed us to define conservation considerations with respect to hardware and software used in software-based art. After analyzing the collected data on these three artworks, we concluded that artist generated source code is a primary risk for these collections. We devoted the next phase of our research to a closer examination of risks associated with source code. The purpose of the research reported in this paper is to investigate whether examining source code can inform conservation practice. A corollary second goal is to define relevant best practices for documenting source code in museum collections.

How the study was designed

In order to address these questions, we selected two other software-based artworks at MoMA for further study: *33 Questions Per Minute* by Lozano Hammer and *Shadow Monsters* by Phil Worthington. In both cases, we sought the artist's permission and communicated with them about analyzing the source code. The two works were selected because their source code contains files of sufficient length and complexity to assess the results of their technical documentation. More specifically, they were chosen because they differ significantly in the following ways:

1. One was written in Delphi, which is a derivative of Pascal and not commonly used in the United States whereas the other was written primarily in Processing, which is a programming environment

that uses Java, a widely adopted language throughout the world including the United States. In addition, Processing was developed primarily for visual artists and Delphi was not.

2. One uses and displays text and can be configured to run in either English, German, or Spanish. The other uses sounds and shapes to display apparently animated images.
3. One supports some viewer interaction and can also be run independently of viewers. The other is an explicitly interactive work of art.
4. One of the artists worked in collaboration with a programmer, whereas the other artist wrote most of the code himself although he collaborated with specialist programmers on some of the more intricate technical aspects of the work such as the integration of sound.

Our study was conducted using a collaborative model in which students and faculty from both the Museum Studies and Computer Science Departments at New York University worked together in order to best ensure that technology skills would complement a deep understanding of art history and the museum context. Holding meetings upon completion of the two studies with conservation, curatorial, and IT staff at MoMA further supported the collaborative strategy. The meetings offered the students an opportunity to present their findings at the museum and respond to questions from different areas of staff expertise.

In particular, conversation with IT staff led to discussion about how to best document the source code. None of the software-based acquisitions at MoMA came with significant code documentation. Given the recent arrival of software-based art in museum collections, there are no industry standards for code documentation. Conversations among our collaborators also addressed whether or not additional narrative descriptions as documentation would inform future conservation interventions in a constructive way.

Software maintenance and documentation: a software engineering perspective

Writing documents is a significant part of software development. Software developers have to write various documents such as the specifications of the software, bug tracking reports, and users' manuals. (Horie & Chiba, 2010)

Keeping program documentation synchronized with the source code it purports to explain is one of the thorniest issues facing the software engineering and technical writing communities today. (Pierce & Tilley, 2002)

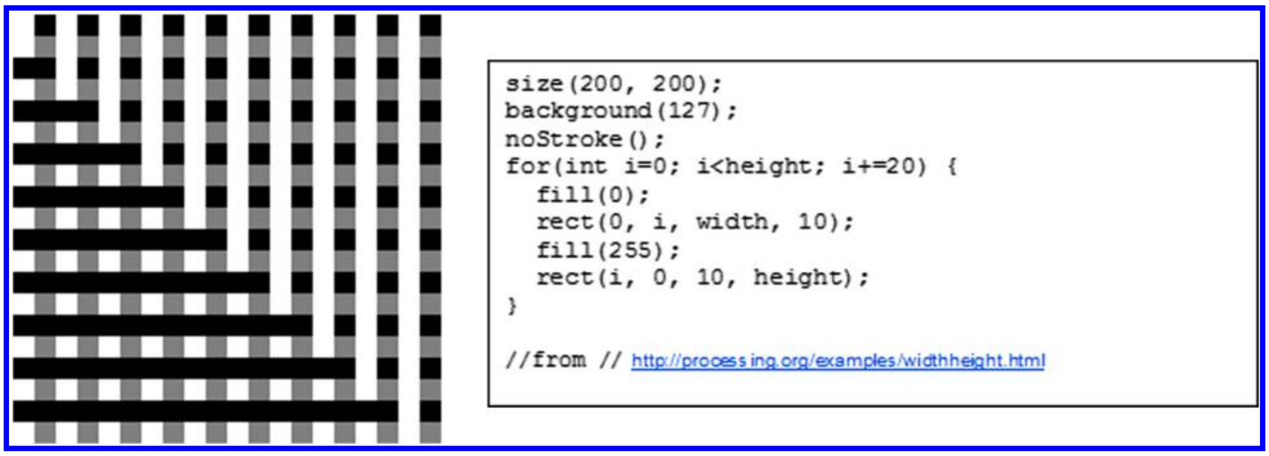
Software that is used in a real-world environment must change or become less and less useful in that environment. (Stroulia & Systä, 2002)

Source code is written by a computer programmer and is 'readable' to one who is trained in programming and in that programming language in particular, just as a document in French is readable to anyone who can competently read French. For example, Fig. 1 contains a 'drawing' that was done in Processing 2+ followed by the source code that creates it.

Without knowledge of Processing or Java, it would not be clear that the source code in Fig. 1 (right side) would render the image in Fig. 1 (left side). However, it is clear that this code is human-readable and further, that if one understands how to interpret terms such as *for, rect, fill,* etc., then this code becomes human-comprehensible as well.

Source code can also include comments; comments are specifically directed at human readers and are ignored by the computer. Programmers use comments to clarify the role of a given block of code, to cite the name of the programmer and a date when a particular modification occurred, and other important notations. In the example in Fig. 1, the source for this example is noted as a comment:

//from http://processing.org/examples/widthheight.html



```
size(200, 200);
background(127);
noStroke();
for(int i=0; i<height; i+=20) {
  fill(0);
  rect(0, i, width, 10);
  fill(255);
  rect(i, 0, 10, height);
}

//from // http://processing.org/examples/widthheight.html
```

Figure 1 Computer generated image on the left, with source code used to generate it, written in Processing, on the right.

Software maintenance

Software maintenance is an engineering term that is analogous to *conservation* with respect to software-based art. Software maintenance is defined as 'the process needed to ensure that a software system continues to satisfy users' requirements. This relevant phase is applicable to software systems developed by using any software life cycle model and programming language' (Scanniello *et al.*, 2010). There is a great deal of literature on this topic within the software engineering community (Forward & Lethbridge, 2002; de Souza *et al.*, 2005; Das *et al.*, 2007; Correia *et al.*, 2010; Scanniello *et al.*, 2010). In the software industry, studies show that 'software maintenance is, by far, the predominant activity in software engineering' (de Souza *et al.*, 2005):

> *It* [*software maintenance*] *is needed to keep software systems up-to-date and useful: Any software system reflects the world within which it operates; when this world changes, the software needs to change accordingly…Maintenance is mandatory…Programs must also be adapted to new computers (with better performance) or new operational systems.* (de Souza *et al.*, 2005)

Software maintenance often requires that programmers who did not participate in the development of a software application must participate in modifying the code to comply with new requirements. This has prompted a focus on software documentation over many years. It has been cited within the software engineering field that, as a general rule, the lack of up-to-date system documentation protocols for many applications is one of the main problems in the industry that complicates software maintenance (Pierce & Tilley, 2002; de Souza *et al.*, 2005; Das *et al.*, 2007).

Software documentation

A software document may be described as an artifact intended to communicate information about the software system. It is aimed at human readers who are involved in the production and/or maintenance of the software (Forward & Lethbridge, 2002; de Souza *et al.*, 2005). Several studies show that the two most important documentation artifacts are the source code and the comments it contains (Huang & Tilley, 2003; de Souza *et al.*, 2005; Das *et al.*, 2007). The next most important software documents are data models in any reasonable form along with non-technical user-point-of-view narrative-style documents such as system documentation describing requirements on how the system is used (de Souza *et al.*, 2005; Scanniello *et al.*, 2010).

The term *reverse engineering* with respect to software refers to 'taking a final project, dissecting it to understand its functionality, and obtain design and other useful information. Software Reverse Engineering is used to analyze a system's code, documentation, and behavior to create system abstractions and design information' (Ali, 2005). Reverse engineering is typically conducted by analyzing and understanding the source code without running it; by running the code and documenting the behavior of the program based on a variety of inputs; or by a combination of these approaches.

We based the methodology for our study on the software engineering model to examine the source code and comments for both works of art; and to create diagrams and narrative documents to reflect what we found in the source code and comments. We also relied on documentation such as artist interviews to explore the system requirements and goals of the system; and user manuals provided by the artists to assist us in understanding the implementation and physical installation of the works for exhibition. This approach is analogous to the software engineering approach with respect to the variety of software-based artifacts that are available.

Software engineers and computer programmers often consult with colleagues who worked on the original production system (Das *et al.*, 2007); this is analogous to our consultations with the artists and their programmers. Further, it has been shown that documentation is more effective when written in an interdisciplinary team, in one case, by assigning technical communicators to work with technical developers (Wong & Tilley, 2002). In our case, we included a Studio Art major and a Museum Studies graduate student respectively as well as one faculty member each from the Computer Science and Museum Studies departments within our working group for each project, in order to ensure that the resulting documentation would be relevant to all of our anticipated readers, both IT specialists and to other museum staff members.

We were aware that differences between the software industry and the museums would become clear in our study due to the different goals of software engineers who maintain software applications and museum staff who conserve software-based art. In the software industry, an application need not be run after it is no longer in use, assuming that the legacy data are properly archived or made available to a different application. Museum conservators, however, are often concerned with how an application ran when it was developed in order to be able to conserve and re-exhibit the work as intended by the artist.

Despite fundamental differences, many of the software documentation goals in industry and for art museums are aligned; the following are examples of our approach with respect to software-based art specifically:

1. We used the source code and comments in the source code as reliable document artifacts for writing additional system documentation.
2. We produced narrative documents that are useful to the users (in this case the conservators and curators as well as IT specialists).
3. We produced visual documents (charts, graphs, etc.) to explicate the system on a higher level for systems staff and programmers.
4. We documented each system including the hardware and operating system along with noting each programming language used and the specific version.
5. We documented any underlying relevant system configurations and possible modifications that would change the way the software runs. For example, one of the works can be set to run in English, German, or Spanish.
6. We reviewed all of the code to determine which components are no longer in use (if any).
7. We documented specific aspects of how the program works which are relevant to understanding it as a work of art: for example, in one of the works, a random sentence is generated in a particular way.
8. We assessed whether modifications had been made to otherwise standard code libraries and where the code libraries were obtained.
9. We assessed how and whether hardware is specifically addressed. For example, the program for one of the works is written to run on a specific monitor.

However, there are other goals for documenting code in software-based art that are not analogous or routinely considered when documenting software applications in the software engineering industry:

1. The user's (or viewer's) aesthetic experience must be thoroughly documented and understood.
2. All code that relates to the artist's process of design in the work – e.g. segments of code that are no longer in use but might provide insight into the artist's earlier design, comments in the code that signify specific aesthetic decisions such as ranges of a color to be used and other information – remain very important to an understanding of the work.
3. The role of any operating system, programming language, and/or hardware limitations current at the time that the work of art was produced should be considered and documented for aesthetic significance. For example, the speed at which an animation runs or the color space available for a specific design could have an aesthetic significance with respect to a work of art.
4. Many artists use proprietary software to build their works of art such as MAX/MSP rather than open source programming languages, which are widely available. Potential conservation risks as well as potential additional expense must be considered with respect to conserving such works of art.

It was our hypothesis that once the behavior of a work of software-based art is thoroughly understood and documented, using a standard software engineering approach along with additional considerations specific to software-based art and museum needs, conservators and programmers will have a better understanding of their choices and decisions with respect to conservation. For example, system documentation based on the software-based artifacts described above would inform both IT and museum staff on how difficult and costly it might be to re-write the source code for a new hardware or software environment, on whether it might be possible to migrate the software to more modern equipment, and on whether to consider alternatives such as exhibiting a video of the software-based artwork running rather than running the software itself.

Case study research

33 Questions Per Minute by Lozano Hammer (Mexican/Canadian, born 1967)

This is a work of art that is exhibited on 21 LCD screens. Sentences are randomly generated at a speed of 33 per minute (described by the artist to be the fastest presentation that remains readable to the viewer)[1] and displayed 'moving across' on the many screens. The work is also variable. For instance, the artist authorizes the use of keyboards for viewers to type in text in some installations (Fig. 2).

The software was written using Delphi, which is a derivative of Pascal. Delphi is a proprietary language, currently owned by Embarcadero Technologies. None of the participants in this project had ever used Delphi but once the documentation for Delphi and Pascal was found online, the project moved forward quickly. Following are some of the document artifacts that were written by the project team:

- File Overview: This one-page document lists all of the primary project files in the source code directory along with a role (e.g. 'This file is automatically created by Delphi for the purpose of… '); with notes (e.g. 'The contents of this file are described in detail in the document called … '); along with grouping the files into logical units such as code units, form units for the user interface, additional files such as the files that contain the textual segments, system configuration information, and files that are no longer in use).
- Project Flowchart: This one-page visual artifact defines a high-level view of how the system runs.
- Chart of Uses: This one-page chart gives a programmer a quick overview and reference to see which programs call the most frequently used functions.

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

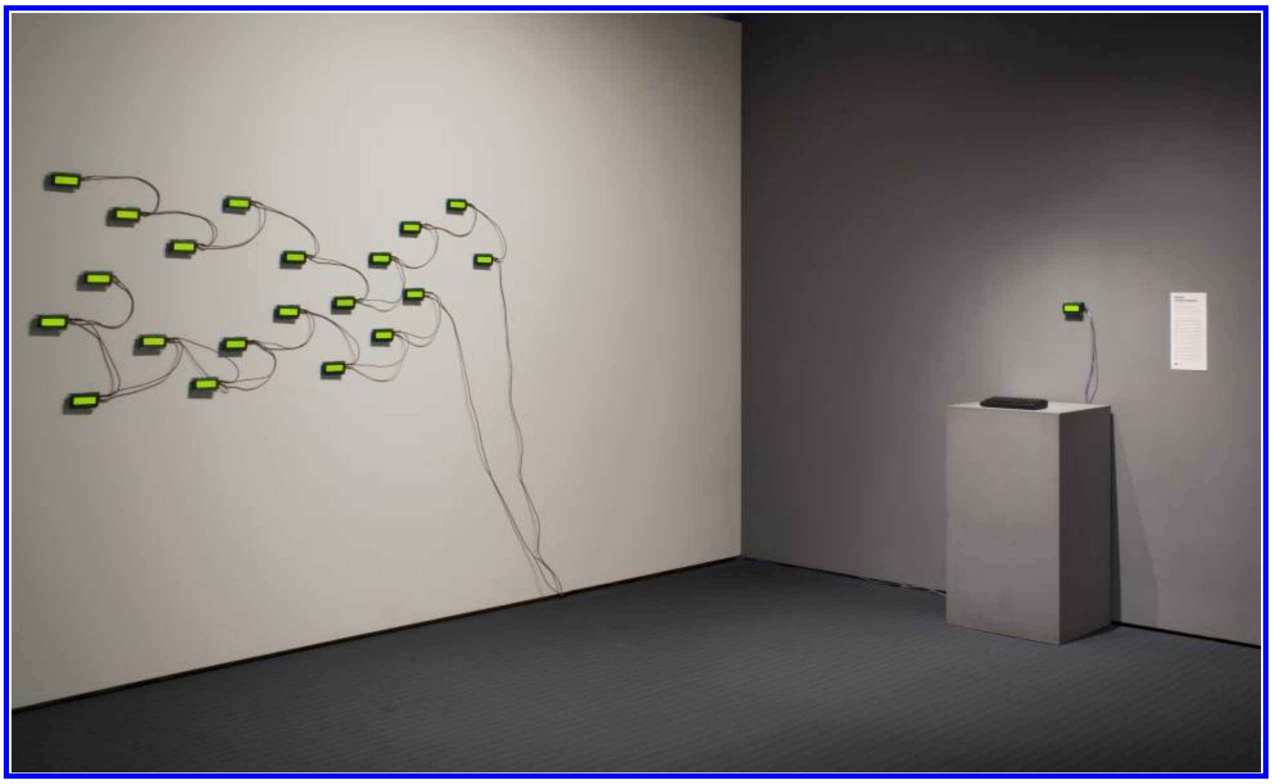


Figure 2 Rafael Lozano-Hemmer, Mexican/Canadian, born 1967, *33 Questions Per Minute*. 2000, 21 LCD screens, computer, Fund for the Twenty-First Century, Museum of Modern Art. 354.2006.

- Source Code Narrative: Each program (in this case, every '.pas' file) is documented with the following information:
  - ○ A one-paragraph summary of the role this program plays in the system.
  - ○ A flowchart or calling tree of the procedures used within the program.
  - ○ A list of all of the procedures and a brief narrative description of each.
  - ○ Particular attention is paid to aspects of the system that determine the aesthetic results for the work of art: for example, the algorithm used to randomly generate the lines of text is examined in detail along with an explanation as to how the code outputs coherent results whether it is running in English, Spanish, or German.
- A flowchart and narrative description are provided to clarify how the phrase lists used to generate the questions are set up for each language (English, Spanish, and German).
- Specific documentation is provided on how the log files, configuration, and setup files are implemented (Appendix 3).

Shadow Monsters by Phil Worthington (British, born 1977)

Shadow Monsters is an interactive work written in Processing, a programming environment that uses Java and was designed for visual artists. As the viewer stands before a wall and moves around, animal-like shapes making animal-like sounds are generated based on the viewer's movements. This work is a play on traditional Victorian shadow puppets (Fig. 3).

Documentation on Processing in particular and on Java in general is widely available. However, not all of the libraries, or program files used for specific tasks such as sound effects, are available in the version that was used. It became clear that when a museum asks an artist to provide his or her source code for the work of art, any additional specialized libraries should also be included, even if the artist did not modify them. Changes to the source code for the physics and sound effects libraries in particular were at times difficult to ascertain for this work.

In Java, one type of excerpt of source code that forms a coherent and logical set of instructions is called a 'class'. A class can contain 'methods' which further define how the program behaves. For this project, we found that writing a description for each class along with a brief (one- or two-paragraph) description for each method served well to organize the information about this work. For example, the *contour* class contains code to manage the silhouette of the person captured by the camera, while the *monster* class contains code to draw the 'monsters'. Special attention was paid in detailed explanations as to how sound files were selected, for example, and the programming steps that occur to update the 'monsters' as they track the silhouettes.

For our project, the code documentation was its own artifact in the form of a word-processing

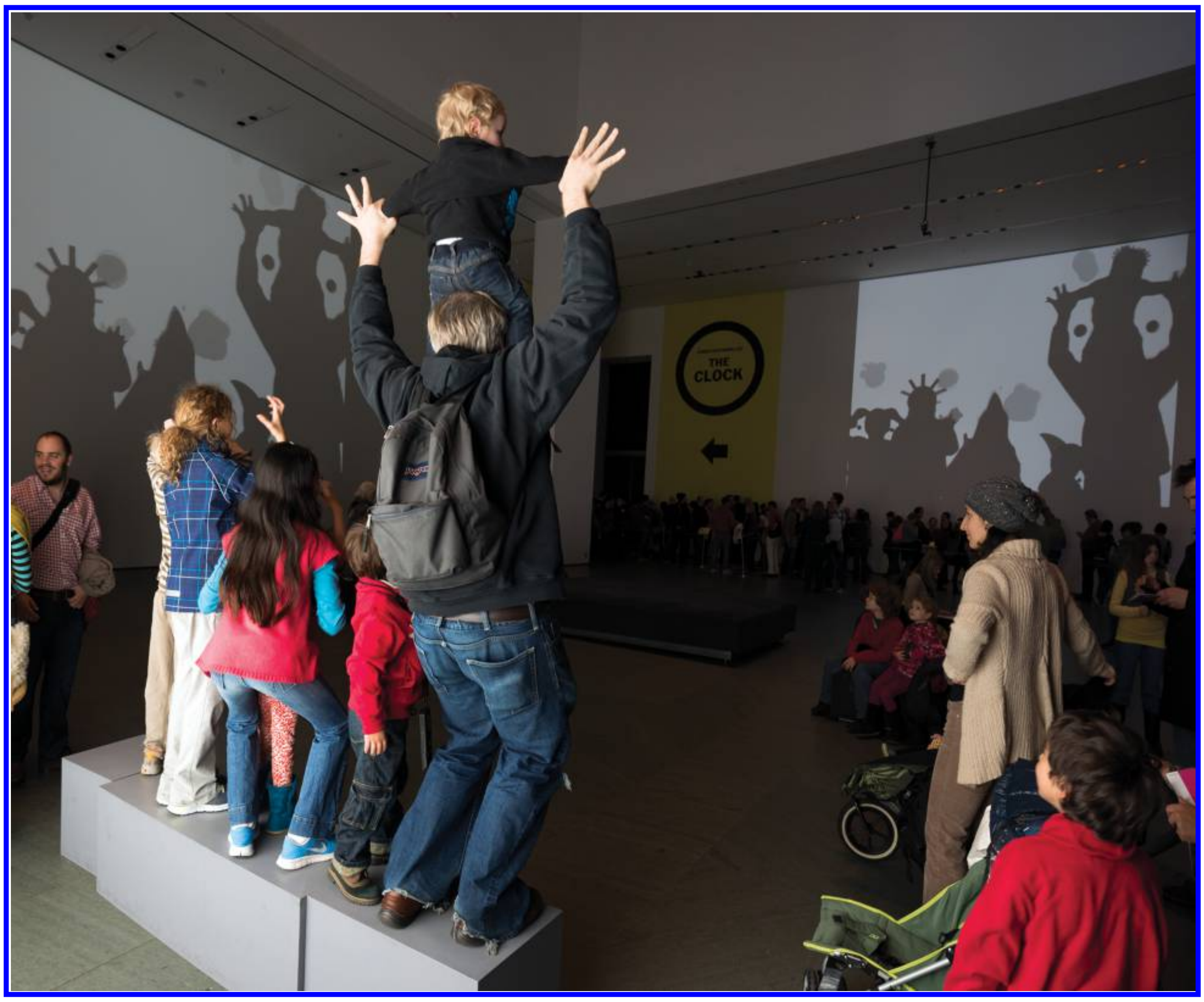


Figure 3 Philip Worthington, British, born 1977. *Shadow Monsters*. 2004-ongoing. Java, Processing, BlobDetection, SoNIA, and Physics software. Gift of the designer. Museum of Modern Art. 485.2008. Credit: Joseph O. Holmes/josephholmes.io.

document (Appendix 4), but an alternative is to insert the narrative descriptions and explanations of the algorithms as text documents into a clean set of the source code files at appropriate locations. This would assist future programmers in understanding how the program works.

Benefits of code analysis and documentation

In addition to documenting the function of each segment of the code, we evaluated the potential for learning more about the artists' working processes through code analysis. For instance, we investigated what we could learn about artist intentions for aesthetic attributes of their work. The following questions were addressed in this aspect of the research:

- If there are fragments of code that are no longer called in the present version of the work, could these lines of code be studied as part of the development of the work in the same way that a curator or art historian studies an artist's preliminary sketches in order to better understand a completed painting or sculpture?
- How is color used, and is the color space specified by the artist?
- Do external multimedia files (sound, moving images, etc.) play a role in the work and with what intent? Are all of the multimedia files included in the installation actually used? Are the multimedia files altered in any way by the work of art?
- If the work is interactive, would an understanding of the code allow future conservators to appreciate all of the possible scenarios that could be generated from the software and better understand their aesthetic significance? For example, a video or similar recording of an interactive work while on exhibition could capture viewers' responses, viewers' input and the results, but would not necessarily capture all of the possible scenarios depending on the specific viewers' actions.
- If the work relies on a database, could an understanding of the source code and data structures assist curators and conservators to better appreciate the role of the data in the final work? Will future data sets be required to keep the work current?
- Are there additional aesthetic aspects that should be defined such as 'rules' behind a randomized aspect of the work?

Other questions that we sought to address with this research concern the future needs of conservators, curators, IT staff, and computer scientists:

- What kind of technical documentation can best serve future exhibition and conservation of the artworks? Should one strive to comment individual lines of code and/or logical groups of lines of code ('blocks' of code) using standard software engineering methodologies? Would a technical narrative suffice? Or would a combination of documenting the source code directly and narratives best be used together to describe specific functions or sub-routines within the overall system?
- What kind of software documentation techniques would best serve future computer scientists charged with emulating or migrating the code to a newer environment?
- Which approaches are most cost-effective for the museum?
- If there are different answers to the above questions for different artworks, can a model be developed to identify best technical documentation strategies for different categories of software-based art?

Following this line of analysis, we found the following:

1. Documenting blocks of code that were clearly no longer in use provided insights into the software engineering or the artistic process, depending on whether the software was written by the artist or by a programmer working with the artist. This form of digital archaeology is parallel to the examination of sketches and maquettes made by artists in traditional media. In *33qpm*, which was not written by the artist, we found code from an earlier version that likely reflected earlier hardware requirements. In *Shadow Monsters,* which was written by the artist, we found blocks of code that the artist had likely used to test specific scenarios. The artist then forgot to remove these blocks of code (or left them in for future testing purposes). In this case, documenting the code that is no longer used helped us to illustrate how the artist anticipated the program would behave. We also found code re-writes and other indicators that gave us a sense of this artist's process.
2. Both works of art use random selections in their presentation. It would be impossible to understand how the randomized questions are created in *33qpm* without the source code; however, reading the source code made it very clear exactly how the words and phrases are stored, retrieved, and manipulated to create reasonable-sounding (if meaningless) questions. Examining the source code clarified for us in *Shadow Monsters* how and when the sound files were used. Although the selection of the sounds seemed arbitrary to the viewer, examining the source code allowed us to precisely describe the degree of randomness as well as regular patterns within the program used to play the sounds.
3. If a work of art requires interactivity, as *Shadow Monsters* does, the source code provides the documentation artifact needed to understand which movements (e.g. shaking one's head, lowering one's arm, etc.) would produce which types of 'monsters'. The monster is created from a specific set of inputs (e.g. a certain number of shapes to represent 'teeth' and 'hair' are represented in a specific way). In this case, the results are randomized only some of the time, so again, it would be impossible to understand the work clearly from viewing it alone. We found that documenting the source code meant that the IT member of our team could succinctly describe these algorithms to the art expert member of our team.
4. Further, in both works of art, we found that the museum studies students had questions based on their understanding of the contextualization of the work, along the lines of 'could the monsters do such-and-such?' 'How many monsters can appear at one time?' 'Are there any alternative colors for the 33 questions' text'? These questions and many others prompted close scrutiny of the code and could be answered by computer science members of the team in citing specific lines of code.
5. Based on our experience, we believe that the underlying source code along with documenting the data structures for a work of database art would clarify and enhance our understanding of that work of art, both from an aesthetic perspective and to inform conservation decisions.

Conclusion

From these two case studies, we learned the following about technical documentation of source code in the museum context:

1. The literature in software engineering states that source code and comments in the source code are the most important documentation readily available on all systems (Huang & Tilley, 2003; de Souza *et al.*, 2005; Das *et al.*, 2007) and we found that to be true in the museum as well.
2. Additional documentation that has been rated as most useful by software engineers includes visuals (flowcharts, charts, graphs) and narrative descriptions, for a higher-level view. We found in these two case studies that a technical narrative which follows the structure of the software along with flowcharts or other artifacts; combined with a descriptive narrative to capture the specifically aesthetic issues, worked well to supplement the technical comments and description of the code (see Appendices 3 and 4.)
3. The museum should document software as soon as reasonably possible after acquisition because documentation on the programming language(s) and any libraries used are more likely to be available.
4. An interdisciplinary team helped to ensure that the resulting documentation would serve curatorial, conservation, and IT interests in the museum. The

exchange of ideas in this collaboration also resulted in explorations and discussions that would not have happened otherwise.

Based on the successful outcome of this study, we envision future research to complement this work using software documentation methodologies to study a work of database art; a work of art that combines several programming languages; and a work of art that uses customized hardware. We plan to explore additional standard software engineering practices such as the use of UML diagrams with Java programs to see if this approach would meet museum needs in such cases. UML diagrams are based on the 'Unified Modeling Language' which is a widely accepted engineering standard used to model computer programs such as applications and how they work. We also plan to continue collaborating with artists and programmers to better understand their concerns as they develop new forms works of software-based art.

Acknowledgements

We are grateful to the artists Rafael Lozano-Hemmer and Philip Worthington for allowing us access to their source code for the purpose of this study. Our communications with the artists gave us a better understanding of their creative process and their concerns for future presentation of their work. In addition, we thank the students at New York University who conducted the code analysis and provided additional observations: Susana Delgadillo, Howard Jing, Kelsey Lee, Liz Pelka, Chris Romero, Erin Schoenfelder, and Albert Yau. We are also indebted 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, Chief Technology Officer Juan Montes, and James Heck, Director of Infrastructure. The following people read drafts of this article and offered valuable suggestions for improvement: Alex Bochannek, Computer History Museum, Jim Coddington, MoMA, Mark Hellar, Hellar Studios LLC, and Craig Kapp, Department of Computer Science New York University.

Appendix 1
Artworks referred to in the text

The Reactive Square by John Maeda; 1994; Accession Number 460.2006.1–2
http://www.moma.org/collection/object.php?object_id=102214

Flying Letters by John Maeda; 1995; Accession Number 461.2006.1–2
http://www.moma.org/collection/object.php?object_id=102218

12 o'clocks by John Maeda; 1996; Accession Number 462.2006.1–2
http://www.moma.org/collection/object.php?object_id=102219

Tap, Type, Write by John Maeda; 1998; Accession Number 463.2006.1–2
http://www.moma.org/collection/object.php?object_id=102220

Mirror, Mirror by John Maeda; 2006; Accession Number 71.2007
http://www.moma.org/collection/object.php?object_id=105813

I Want You to Want Me by Jonathan Harris & Sep Kamvar; 2008; Study Collection Number SC527.2008
http://www.moma.org/interactives/exhibitions/2008/elasticmind/index.html#/116/

Lovers by Teiji Furuhashi; 1994; Accession Number 331.1998
http://www.moma.org/interactives/exhibitions/1995/videospaces/furuhashi.html

33 Questions Per Minute by Rafael Lozano-Hemmer; 2001–2002; Accession Number 354.2006
http://www.moma.org/collection/object.php?object_id=102431

Shadow Monsters by Philip Worthington; 2004-ongoing; Accession Number 485.2008
http://www.moma.org/collection/object.php?object_id=110196

Appendix 2
Software-based art risk assessment template

The following template was designed by the authors to assess complex software-based art in MoMA's collection. It serves as a tool for organizing existing documentation, assessing risk, and developing priorities for conservation and collections management.

Catalogue information

- Artist
- Title
- Date
- Accession number
- Department

Introduction

Describe the work in general terms including the following as appropriate, use excerpts of artist interviews and other source materials

- Exhibition history
- Installation environment/installation variability
- Performance/interactive components
- Media technologies (computer hardware, computer software, multimedia technologies)
- Equipment and technologies

- Media (specify archival and exhibition media formats)
- Display equipment (dedicated or non-dedicated)

- Sculptural objects: (how it is deployed, role of the object)

Technical specifications

Hardware

- Computers (technical specifications required for custom hardware)
- Displays
- Peripherals (keyboards, mice, numeric keypads, external hard drive, etc)
  - For each peripheral: note if it is required for user interactivity
- Multimedia: (video cameras, sound system, projection system(s))
- Miscellaneous (e.g. equipment required to capture current data)

Software

- Software platform
  - Operating system
  - Hardware required to run the work
  - RAM, disk storage requirements for data tables maintained, disk storage requirements for data generated during installation, processor speed, color requirements
- Custom application software (note: the following is needed for each custom software application as well as for any additional libraries, e.g. MIT's Processing)
  - Source code language, version, compiler/IDE used, author/artist, proprietary vs. open source (at the time the work was created and at the time of this report)
- Proprietary software required (e.g. Adobe's Flash Player, a specific browser)
  - Name, version, dependencies
- Data Source for works that rely on data analysis … for each data set:
  - Data source, data table structure(s), database format (e.g. MySQL)
  - Is there a permanent stable data set available?

Multimedia components

- Asset (magnetic tape, image file, etc) – for each one:
  - How it is deployed
  - Properties – where they apply (size, sampling rate, duration, etc)

Installation specifications

- Room specifications: (e.g. dimensions, entrance and exits, light-lock corridor specifications, wall and ceiling finish, floor finish, equipment visibility, special signage, plinths, benches)
  - Light source and light levels specifications
- Equipment installation specifications
  - Hardware and multimedia setup/cables/brackets/ceiling supports
  - Interactivity setup
  - Acoustic qualities and audio levels/number of speakers/speaker format (e.g. Mono/Stereo/Dolby 5.1)
  - Specifications on running the software
  - Projection format/projection distances/image size and relationships to walls, ceilings, floors
  - Equipment cupboard (access, shelving, etc):
  - Power requirements (e.g. volts, AMPS, number of sockets)
- Installation maintenance: Are staff required to adjust, check, and/or otherwise manage the piece? If so, on an hourly/daily/weekly basis?

Technical history

- Hardware used in earlier installations
- Software used in earlier installations
  - Versions of the software used in previous installations
  - Upgrades to the code (specific to this piece)
  - Upgrades to the language(s)
  - relevant general software
  - operating system history as it impacts the language
  - Any changes in operating system selection from earlier installations

Condition assessment

- Functional assessment with respect to hardware, software, and components
  - Is the work fully functioning including all components?
  - Are any modifications necessary?
  - Are there provisions to trap errors and re-start?
- How 'buggy' is the code?
- Functional assessment with respect to artist specifications:
  - Speed, color, user interactivity, etc
  - Installation environment (e.g. light levels, audio levels, height, placement)

Risk assessment

- Hardware and equipment vulnerabilities:
  - Condition concerns
  - Proprietary/copyright concerns
  - Equipment obsolescence
- Software vulnerabilities:
  - Does the museum have a copy of all of the source code?
  - Operating system dependencies
  - Proprietary software concerns
  - Changes to programming languages (e.g. Python 3 is not backwardly compatible)
  - Data sources (for works based on data analysis) – does the museum have a stable and permanent set of data to use to run the application?
  - Internet issues (for net art)
- Media vulnerabilities
  - Vulnerability over time: multimedia files and file-formats

    - Stability of image, sound, and video data formats in storage
    - Future availability to view/play image, sound, and video data formats
  - Vulnerability over time: storage media (e.g. laser disks) used for media
    - What is the vulnerability over time?
    - What are the storage requirements?

Recommendations
- Short term/low cost
- Long term/higher cost

References
- Relevant publications and multimedia websites
- Internal documents: user manuals, technical specifications, installation specifications, artist interview transcripts

Author(s)
Report date

Appendix 3
Documentation from *33 Questions per Minute*

For additional information about the Delphi language, see Embarcadero Developer Network website <http://edn.embarcadero.com/delphi>.

For documentation about Delphi and Pascal, see Delphi Basics website <Delphi Basics – http://www.delphibasics.co.uk/Article.asp?Name= DelphiHistory>; Delphi Object Oriented Features <http://www.derangedcoder.net/programming/general/comparingObjectOrientedFeatures.html>.

Figure 4 is a sample of the documentation produced for *33 Questions per Minute.* It is a diagram of how the software stores phrases. The words and phrases are retrieved in the specified language (English, German, or Spanish) from a text file and then made available to the software so that each list can be readily identified and accessed. Examples of phrases in one list include 'Won't you', 'Who should'; another list contains verbs such as 'masquerade' and 'bother'; yet another list contains adverbs such as 'cordially' and 'proactively', and so forth. This chart demonstrates that there is an array (a data structure) that identifies up to 26 lists of phrases in a specific language and that each element of that array in turn consists of another array that contains the specific words and phrases within each given list. The program also stores the ten most recent phrases that were chosen in order to prevent duplication within a short time (Fig. 4).

Appendix 4
Documentation from *Shadow Monsters*

For additional information on the Processing language, see <http://processing.org>.

Figure 5 is a sample of the documentation produced for *Shadow Monsters.* It is a text description of a sound library developed by the artist. Although the sounds that play in the piece (grunts, groans, burps, and so forth) might seem to the viewer to happen randomly, there is a logic behind which sound files are played in this program and at what times. In this documentation sample, we have described how the

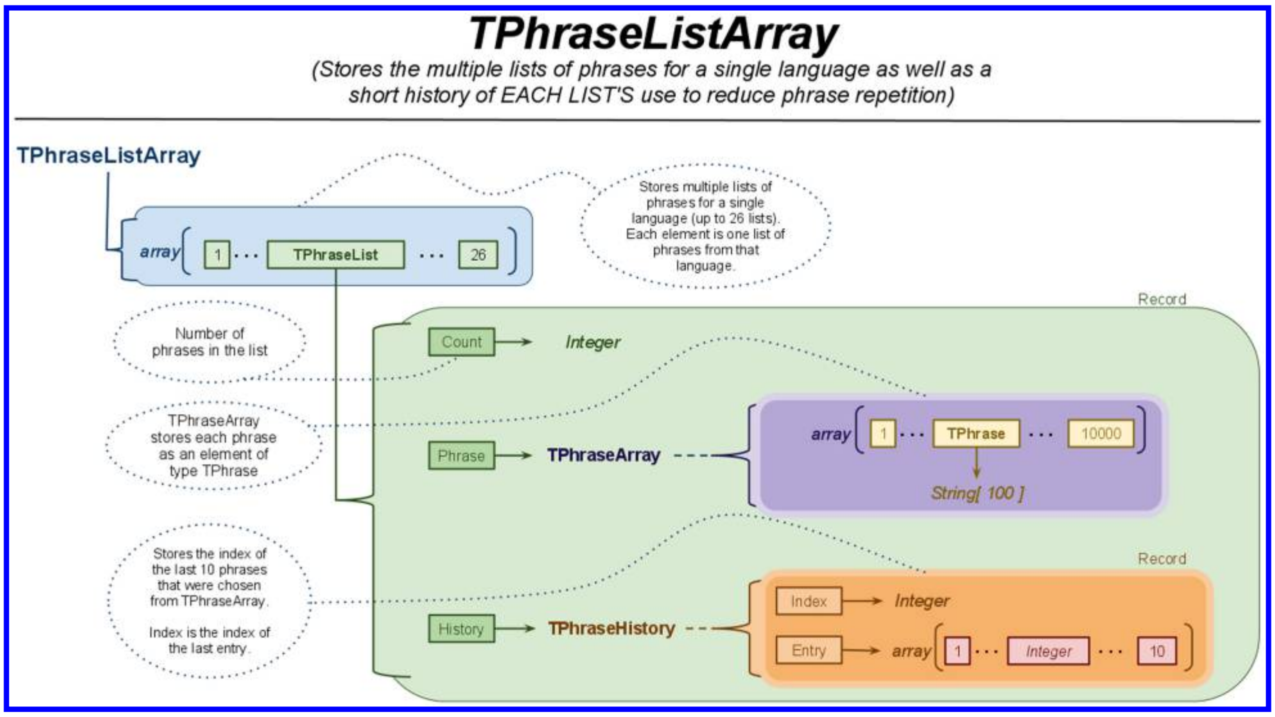

TPhraseListArray
(Stores the multiple lists of phrases for a single language as well as a short history of EACH LIST'S use to reduce phrase repetition)
TPhraseListArray
array 1 ... TPhraseList ... 26
Stores multiple lists of phrases for a single language (up to 26 lists). Each element is one list of phrases from that language.
Record
Number of phrases in the list
Count → Integer
TPhraseArray stores each phrase as an element of type TPhrase
Phrase → TPhraseArray
array 1 ... TPhrase ... 10000
String[100]
Record
Stores the index of the last 10 phrases that were chosen from TPhraseArray.
Index is the index of the last entry.
Index → Integer
History → TPhraseHistory
Entry → array 1 ... Integer ... 10

Figure 4 Visual documentation for *33 Questions per Minute.* From student source code documentation report by Delgadillo, S., Lee, K., Pelka, L., Schoenfelder., & Yau, A. April 25, 2011.

SoundLibrary class in Java is set up and the ways in which sounds are produced. This type of documentation, although narrative, is geared towards a technical audience and could be inserted as comments into appropriate locations in the source code.

Sound Library

The *SoundLibrary* class is responsible for loading and playing the sounds that appear during the lifetime of Shadow Monsters. Sounds can be classified into four types: burps, low-pitched voices, medium pitched voices, and high-pitched voices. The pitch of the voice is determined by the size of the monster's mouth.

Sounds are loaded into the *Sonia* library, and the file names are hardcoded inside of the *SoundLibrary* constructor. Sounds are not randomly chosen, but are played in a specific order.

The methods of the sound library are: newSample(int lib, String filename), playSnd(float size), playBurp(float size), and nextAvailable Snd(int lib).

void newSample(int lib, String filename): This method loads a new sound file and inserts it into the sound library.

Sample playSnd(float size): This method plays a sound. It takes a mouth size as an argument. A large mouth will produce a low-pitched sound, a medium sized mouth will produce a medium pitched sound, and a small mouth will produce a high-pitched sound. The playSnd method will call the nextAvailableSnd method to play the next sound that is not currently playing.

Sample playBurp(float size): This method plays a burp. Mouth size has no effect on the burp, and the sound is selected by the nextAvailableSnd method.

Sample nextAvailableSnd(int lib): This helper method is used to select the next available sound. It iterates through one of the four categories of sounds. It checks each of the sounds to see if it is currently playing or not. The first sound that is not currently playing will be queued up to play. If all sounds are playing, then nothing is returned. The position of the returned sound is recorded, and the next time this function is called, the first sound clip that will be checked will be the sound clip positioned directly after the returned sound. This means that the voices and burps of the shadow monsters are played in the same order each time.

Figure 5 Text documentation for *Shadow Monsters*. From student source code documentation report by Howard Jing, 4 April 2012.

References

Ali, M.R. 2005. Why Teach Reverse Engineering? *SIGSOFT Software Engineering Notes*, 30(4): 1–4.

Correia, F.F., Aguiar, A., Ferreira, H.S. & Flores, N. 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.

Das, S., Lutters, W.G. & Seaman, C.B. 2007. Understanding Documentation Value in Software Maintenance. In: E. Kandogan & P.M. Jones, eds. *Proceedings of the 2007 Symposium on Computer Human Interaction for the Management of Information Technology (CHIMIT '07), 30–31 March 2007*. New York: ACM, Article 2.

de Souza, S.C.B., Anquetil, N. & de Oliveira, K.M. 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, pp. 68–75.

Forward, A. & Lethbridge, T.C. 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, pp. 26–33.

Horie, M. & Chiba, S. 2010. Tool Support for Crosscutting Concerns of API Documentation. In: M. Südholt, U. Hohenstein, J.-M. Jézéquel & B. Baudry, eds. *Proceedings of the 9th International Conference on Aspect-Oriented Software Development (AOSD '10), 15–19 March, 2012*. New York: ACM, pp. 97–108.

Huang, S. & Tilley, S. 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, pp. 93–9.

Laurenson, P. 2006. Authenticity, Change and Loss in the Conservation of Time-Based Media Installations. *Tate Papers* [accessed 14 October 2013]. Available at: <http://www.tate.org.uk/download/file/fid/7401>

Lieser, W. 2009. *Digital Art*. Saarbrücken, Germany: Tandem Verlag GmbH.

Paul, C. 2007. Challenges for a Ubiquitous Museum: Presenting and Preserving New Media [accessed 14 October 2013]. Available at: <http://www.neme.org/571/preserving-new-media>

Reas, C. & Fry, B. 2007. *Processing: A Programming Handbook for Visual Designers and Artists*. Cambridge, Massachusetts: MIT Press, pp. 160–3.

Pierce, R. & Tilley, S. 2002. Automatically Connecting Documentation to Code with Rose. In: *Proceedings of the 20th Annual International Conference on Computer Documentation (SIGDOC '02) 20–23 October, 2002*. New York: ACM, pp. 157–63.

Scanniello, G., Gravino, C., Risi, M. & Tortora, G. 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.

Stroulia, E. & Systä, T. Spring 2002. Dynamic Analysis for Reverse Engineering and Program Understanding. *Applied Computing Review*, 10(1): 8–17.

Tribe, M. & Jana, R. 2009. *New Media Art*. Köln, Germany: Taschen GmbH.

Van Saaze, V. 2009. Authenticity in Practice: An Ethnographic Study into the Preservation of *One Candle* by Nam June Paik. In: E. Hermens & T. Fiske, eds. *Art Conservation and Authenticities: Material, Concept, Context*. London: Archetype Publications, pp. 190–8.

Wong, K. & Tilley, S. 2002. Connecting Technical Communicators with Technical Developers. In: K. Haramundanis & M. Priestley, eds. *Proceedings of the 20th Annual International Conference on Computer Documentation (SIGDOC '02) October 20–23, 2002*. New York: ACM, pp. 258–62.

Wharton, G. 2005. The Challenges of Conserving Contemporary Art. In: B. Altshuler, ed. *Collecting the New: Museums and Contemporary Art*. Princeton: Princeton University Press, pp. 163–78.

Wood, D. 2008. *Imaging By Numbers: A Historical View of the Computer Print*. Evanston, Illinois: Mary and Leigh Block Museum of Art, Northwestern University.