Software Archaeology and the Preservation of Code-based Digital Art

Francis T. Marchese; Computer Science Department, Pace University; New York, NY/USA

Abstract

The long-term conservation of code-based digital art remains an open issue. Recently, we have proposed the use of software engineering methodologies to create rigorously structured documentation that will support archival preservation of a digital artwork with the intent of future installation. In this paper we expand this notion by proposing that the software engineering process, and its artifacts provide a means for systematically organizing and comparing a collection of digital artworks. In so doing, software engineering transforms existing artisanal preservation procedures into a schematized process that can be integrated with traditional art conservation practice. Software preservation also becomes an activity associated with software archaeology: the systematic study of software systems through the recovery and examination of outstanding material evidence, such as source code, tests, and design documentation. The paper focuses as well on the practice of software archaeology as it applies to the systematic study of software-based artworks to: reveal patterns in their underlying architectures, preserve archetypal software, maintain a historical record, and select artifacts for preservation.

Introduction

Conservators of software-based art must archivally manage an assortment of artworks in order to make them displayable at any time in the distant future. Digital artists employ an expanding range of computer languages. Software interfaces, formats, and protocols are evolving. Globally accessible resources either disappear or become redistributed. And computer hardware will become obsolete. Conservators have taken a two-pronged approach to preserving a digital artwork: technology preservation, in which replacement parts are stockpiled; and document compilation, in which extensive documentation is assembled to help define and contextualize the artwork. Both these approaches remain problematic. Museums and cultural institutions neither have resources to stockpile computer parts, nor the ability to routinely maintain artworks to extend their lifespans. Since museums collect far more artwork than they can exhibit at any particular time, all remaining art may be expected to rotate from storage into galleries pursuant to curatorial discretion, with the exception of works that either define a museum's collection or are critical to the art canon. In such environments it may be decades before artworks are reinstalled. As a result, routine maintenance of these works becomes managerially prohibitive because of time, staffing, and financial constraints. This leaves open the prospect that when an artwork is finally scheduled for installation, it may not be possible to do so, because either part or all of the artwork will have reached technical obsolescence.

We have proposed a solution to the preservation of software-based digital art that addresses these issues by recasting artwork as a unified collection of documents [1][2]. It posits a long term view of digital art preservation in which a curator and conservator five hundred years in the future should be able to install today's digital art, something commonplace today with traditional art from five hundred years ago (e.g. Medieval and Renaissance). We contend that software engineering methodologies should make it possible to construct rigorously structured documentation, that will support archival preservation of a digital artwork intended for future installation. By definition, software engineering is the process of applying a systematic, disciplined, quantifiable approach to problem analysis, system and software design, its development, operation, and maintenance [3]. Software engineering methodologies focus on both the software product, and the process used to create and maintain it. In the latter case, the software life-cycle is an extension of the business life-cycle, and defined by the business process management model (BPM) [4]. As such, its tools and techniques may be integrated into a museum's conservation practice.

In this paper we expand this notion by proposing that the software engineering process and its artifacts provide a means for systematically organizing and comparing a collection of digital artworks. In so doing, software engineering transforms existing artisanal preservation procedures into a schematized process that can be integrated with traditional art conservation practice. Software preservation also becomes an activity associated with software archaeology: the systematic study of software systems through the recovery and examination of outstanding material evidence, such as source code and design documentation. Here, we put forward the practice of software archaeology as a holistic approach to the systematic study of software-based artworks in order to reveal patterns in underlying architectures, preserve archetypal examples of code-based art, select artifacts for preservation, and maintain the historical record.

In the following section we expand the definition of software archaeology stated above. This is followed by sections that: define a theoretical model for analyzing artworks, discuss the use of reverse engineering recovering structural information, and compare some examples.

Software Archaeology

Booch [5] defines software archaeology as "the recovery of essential details about an existing system sufficient to reason about, fix, adapt, modify, harvest, and use that system itself or its parts." This is a tactical approach to software understanding in which the results of research are utilized to maintain the digital artifacts it studies. A strategic approach to artifact understanding comes from the field of archaeology itself, in which research

results are categorized, compared, and interpreted over time. As defined, archaeology is the study of human activity in the past, primarily through the recovery and analysis of the material culture and environmental data that they have left behind. One goal of archaeology is to expose the structure of the past through description and classification of physical evidence so as to use it to account for the distribution of the remains of ancient societies over time and space. Another goal is the determination of a system's behavior, represented by physical remains, in order to understand underlying cultural processes. If we consider that the wide variety of digital artwork designs fall into artistic genres such web, immersive, mobile, pervasive, and more; and that each software category has its own "tribal" or "cultural" identity, meaning that the design and behaviors of each software artifact may be associated with a particular tribe of artists or digital artistic culture, then software archaeology adopts the holistic and strategic dimensions characteristic of the archaeological field.

Both tactical and strategic approaches are essential for digital archaeology. On the one hand a conservator must be able to preserve digital art for it ultimate installation. On the other, the conservator needs a global or holistic view of the kinds of digital art that must be preserved in order to define and use best practices for maintaining a particular digital artwork. Such a view allows conservator and art historian to perceive an artwork as a reflection of the conceptual space the artist had worked in at the time the art was created. The number of software components, their hierarchy, and the interconnections among them should give an idea of how the artist viewed a representation problem, and how it was transformed into a computer system. As such, the works within each group should share a common set of design constructs, the evolution of which could be monitored over time. Hence, archaeological analysis of digital artwork should yield important answers to questions about:

- Authorship – which parts were written by whom (The hand of the artist? Or was the artist the designer who contracted out construction?).
- Educational/Cultural context – who influenced the artist conceptually or technologically.
- Craftsmanship – how well the software was written and system built, how well the parts fit together.
- Aesthetics – Well conceived and designed software possess an elegance and refinement comparable to any other beautifully created or designed object.
- Development Process – design strategies used by the artist.
- Technical Context – what development tools, libraries, environments were available at the time the artwork was created.
- Theoretical Foundations – theories/paradigms of computer organization, algorithm use, and data design.

Table 1 (adapted from [6]) collects together the components of the strategic and tactical components associated with software archaeology, organizing them under the headings of "preservation" and "usage", respectively. Preservation assumes a holistic perspective, concentrating on the acquisition, evaluation, organization, and integration of digital artwork into a software collection. One process important to the preservation of digital art is the development of design patterns [7]. A design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. Originally introduced by the architect Christopher Alexander to solve recurring problems that arose in the design of towns, buildings, and construction activities, patterns offer conceptual and concrete guides to problem solving [8]. As part of a strategic preservation initiative, creation of design patterns through analysis of digital art collections should provide general solutions to problems common to the long-term maintenance of each genre.

Table 1: Software Archaeology — Dual Perspectives

Preservation:	**Usage:**
Strategic	**Tactical**
Software Repository	Legacy Systems
Software Preservation	Software Reuse
Rights Management	Reverse Engineering
Software Selection	Software Construction
Software History	Leaving a Legacy for Future Generations
Classic Software	Integration
Design Patterns	Software Maintenance
Time: Centuries	Time: Decades or Less

The tactical approach to software archaeology in Table 1 assumes the artwork to be a legacy system in which its conceptual and technological underpinnings are frozen in time. This is a natural assumption for most digital artworks, given that when art museums acquire such works they should be complete, functioning, with no further additions or modifications anticipated. Reverse engineering becomes an important part of the maintenance strategy behind this artwork, in which it is analyzed to identify its component parts, and the interrelationships among those parts, in order to either create representations of the system in another form or representations at a higher level of abstraction [6]. Reverse engineering generally involves extracting design artifacts to synthesize more general abstractions. It does not necessarily involve changing the system under study or creating a new system based on the reverse engineered system, although these are two characteristic goals of the process. As Chikofsky and Cross have stated "reverse engineering is a process of examination, not a process of change or replication." [9] In all, the methodologies underlying software reuse, reverse engineering, and software maintenance work in concert to provide information about low and high level representations of an artwork from source code through system design documents.

Theoretical Model

A theoretical model is required to understand individual digital artworks and systematically compare them. The approach used here relies on the "4+1 View Model" of Kruchten [10] which captures multiple views or perspectives of software architecture. Models provide referents for analyzing software and formal structures or patterns for its decomposition. Software architecture represents the high-level structure of a software system. It can be defined as the set of structures needed to reason about the software system, which comprise the software's component parts, the relations between them, and the properties of both components and relations. Software architecture is concerned with the processes of

abstraction, decomposition, and composition, utilizing style and aesthetics as guides.

The "4+1 View Model" consists of *logical*, *process*, *physical*, and *development* views. Added to this are use cases or *scenarios* (the "plus one" view) that bind together the four architectural components (cf. Figure 1). The *logical* view represents the functional requirements — a collection of conceptual classes or abstractions that define and characterize the actions the artwork performs. The *process* view captures the artwork's behavior, exposing its distribution of tasks and their synchrony. Crucial to a *process* view is its display of a thread of control that reveals the sequential communication between process tasks, enabling a process to generate a particular behavior. The *physical* view describes the mapping(s) of the software onto the hardware taking into account the system's non-functional requirements. Non-functional requirements, sometimes called quality attributes, are not system features, but rather required system characteristics. They represent important aspects of the system such as performance, security, usability, adaptability, compatibility, and legal concerns. The *development* view describes the static organization of the software in its development environment, focusing on the organization of software modules within the software development environment. Software is packaged naturally into subsystems that are organized in a hierarchy of layers, each layer providing a narrow and well-defined interface to the layers above it. Finally, *scenarios* describe how one or more entities interact with a system by enumerating the steps, events, and/or actions which occur during engagement. Scenarios may be considered abstract representations of the most important system requirements, because at a high level they specify what the system is expected to do. As such, they impact all four architectural views, and are shown in Figure 1 as binding them together.

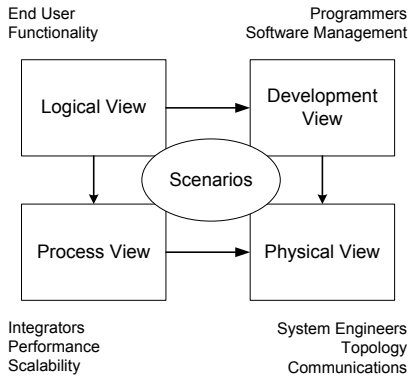

Figure 1. *"4+1 View Model" [10].*

Creating these architectural views requires a systematic reengineering approach that works backward from the software components acquired by an art museum. Such an approach is necessary because this documentation is expected to differ significantly by artwork, and exhibit varying degrees of incompleteness, inhomogeneity, and diversity in its content and format. Software engineering provides a systematic methodology for creating and maintaining documentation to support communication, preservation of system and institutional memory, and processes such as system auditing. Within this context an artwork's documentation should supply comprehensive information about its capabilities, architecture, design details, features, and limitations. It should encompass the following five components [7]:

1. *Functional Requirements* – The artwork's conceptual foundation. What it is supposed to do.
2. *Architecture/Design* – An overview of software that includes the software's relationship to its environment, and construction principles used in design of the software components.
3. *Technical* – Source code, algorithms, and interface documentation.
4. *End User* – Installation, maintenance, and user documentation.
5. *Supplementary Materials* – Anything else related to the system. This includes: legal documents, design histories, interviews, scholarly books, installation plans, drawings, models, documentary videos, websites, etc.

Each component is important to the representation of a digital system. Each may operate at a different level of abstraction or within a particular context. *Functional Requirements* documentation presents the conceptual view of what the system is expected to do. It is written to be understood by all the stakeholders who comprise an art museum's business practice. *Architecture/Design* documentation functions very much like an architect's sketch of a building, showing all its components and how they fit together. *Technical* documentation represents the bricks-and-mortar of the artwork, conveying information about how the artwork is constructed. Figure 2 displays mappings between software documentation and the "4 + 1 View Model". The open-headed solid arrows indicate how the software documentation organizes the views, while the closed-headed dash arrows show the possible contributions *Supplementary Materials* may contribute to understanding the design of these views. It should be emphasized that the reason why we perform a mapping to a standardized set of formal software engineering documents, such as those provided by the Rational Unified Process (RUP)

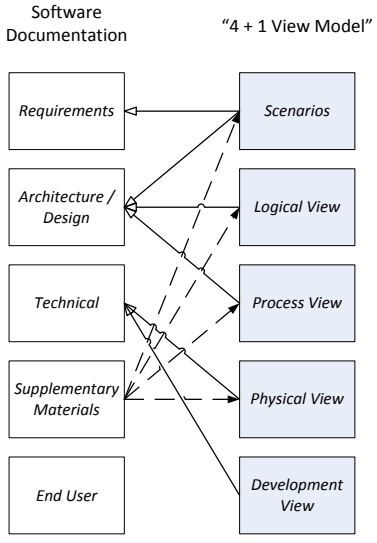

Figure 2. *"4+1 View Model" and software documentation mapping.*

[11], is that it immediately affords a comparison among artworks, thus supporting the archaeological scheme.

Reverse Engineering

Reclamation of an artwork's design is a reverse engineering process that begins with recovery of user *scenarios* – the foundation of functional requirements. One place to look for these *scenarios* is in the artwork's *Supplementary Materials,* where its temporal designs may have been depicted in storyboards, videos, and such. Otherwise, *scenarios* may be captured through observation of visitors engaging the artwork as it operates within a gallery setting. Formal usability techniques [12] may be applied to the problem in order to extract the precise dialog between artwork and viewer. Typically, a set of scripts are written to systematically define user engagement, and then are followed stepwise to explore each aspect of interaction. *Supplementary Materials* may contribute to the *Physical View* as well, supplying non-functional requirements defining how a gallery visitor is expected to engage the artwork. If the artwork has been provided as source code, this contributes to the *development view* that makes up the *Technical* documentation. If only executable code exists, then it may be possible to decompile it to source code, something that can be done if the artwork has been constructed with popular programming languages. The *logical* and *process* views that constitute the *Architecture/Design* documentation in the form of UML (Unified Modeling Language) class diagrams that fix an artwork's static structure as a collection of basic, interconnected building blocks, and sequence diagrams that convey the messages passed between the user and the system, and among objects within the system [7], are not expected to be part of the artwork when acquired by a museum. Most artists are not trained as software engineers, and thus are not expected to create UML representations of their works, as of yet.

Given a complete set of scenarios, and employing the remaining documentation categories as the interpretive context, it should be possible to generate UML representations – in effect, produce a complete design document for the artwork from scratch [7]. Alternatively, in circumstances where the artwork has been written in a popular programming language such as C, C++, or Java, UML class and sequence diagrams may be generated automatically from either source or executable code exploiting mainstream UML CASE tools [13]. The resulting UML diagrams embody the artist's original software architecture as opposed to an architectural design that has been inferred from the remaining documentation categories.

Examples of UML class and sequence diagrams are shown in Figures 3 and 4 for a video store, a generic, non-artistic project, that is characteristic of many software systems. The class diagram represents the video store's information domain composed of conceptual classes, shown as boxes, with labels such as Customer, VideoStore, Video, VideoRental, etc., that define the main business entities. Each class is connected to others by lines of association which specify their semantic relationships such as "One CashPayment Pays for one RentalTransaction." This is a high level, or abstract class diagram, because it does not show the kinds of processing each class performs in support of the video rental procedure. The high level sequence diagram in Figure 4 represents the dialog between the clerk and system during the

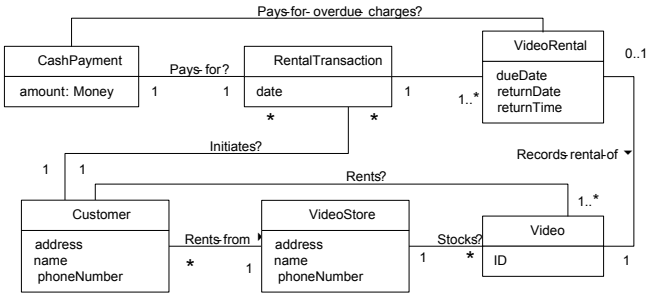

Figure 3. *Sample conceptual class diagram..*

video rental process. The right-pointing sold arrows show the clerks request while the left pointing dashed arrows show the systems response. Such a diagram would be created by translating scenario narratives. This diagram also works at the class level to express the dialog between system classes as each responds to a request for information or processing.

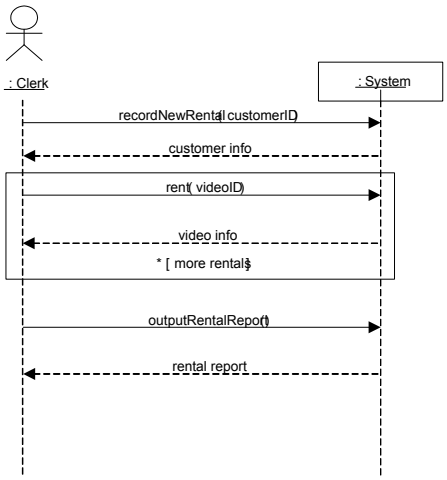

Figure 4. *Sample sequence diagram.*

Comparing Architectural Structures

The goal of our analysis was to explore the architectural structures among digital artworks. As an initial archaeological exploration, five works were selected for analyses from the *CODeDOC* exhibition of the Whitney Museum of American Art's *AirPort* website [14]. *CODeDOC* exhibits both artwork and the underlying code, allowing individuals to look under the artwork's hood. Artists were given the assignment of connecting and moving three points in space, with their source code not to exceed 8KB in size. Works selected here for analysis were created by the artists Mark Napier, Golan Levin, Brad Paley, Scott Snibbe, and Martin Wattenberg. All were interactive, graphical Java applets. Briefly, these artworks included a cartographic application (Levin), dynamic geometric art (Napier, Snibbe, and Wattenberg), and text visualization (Paley).

Architectural diagrams were generated automatically from either an artwork's Java source or compiled codes, utilizing a UML design tool. A sample design class diagram is presented in

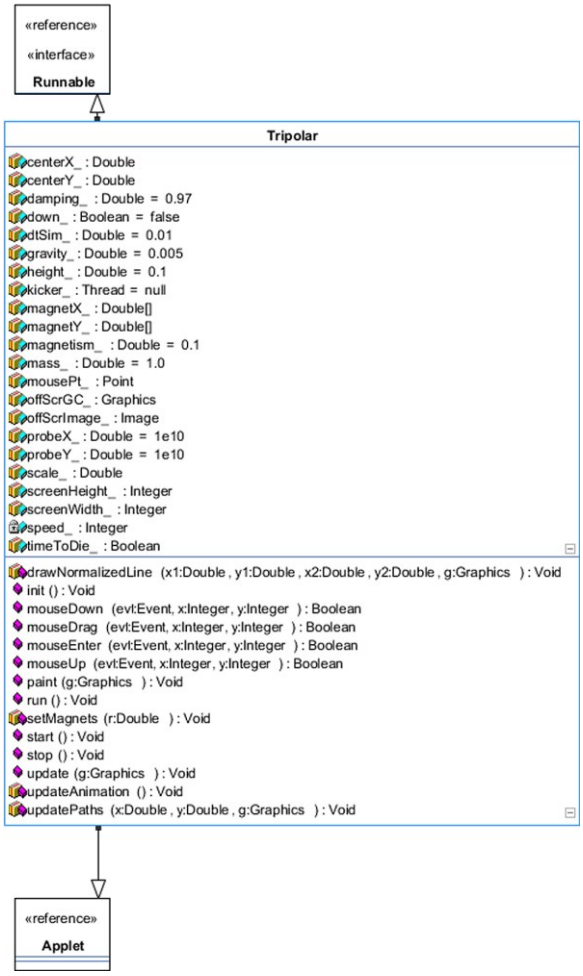

Figure 5. *Class diagram for Scott Snibbe's* Tripolar *artwork.*

Figure 5 for Scott Snibbe's *Tripolar* artwork. This diagram shows a program comprised of one software class that collects together all data (middle segment of the class diagram) and functionality (bottom segment). The other four artworks exhibit similar monolithic architectures as well. No sequence diagram is shown because there is only one defining class for each program. A sequence diagram's utility is in its ability to show collaborative behavior among classes or other interacting entities, such as that shown in Figure 4. Hence, a minimum of two classes or entities are required to produce a sequence diagram.

Given that all artists shared the same problem and constraints, there might be an expectation of similarity in design across artworks. In all works the software is tightly written. All employ similar Java libraries for interaction and graphics. Essentially, they only differ in the code related to what each portrays on the computer screen, which can differ radically across artworks. See Figure 6 for example, which displays class diagrams for Paley's and Napier's applets, respectively. Paley's is a far larger applet, involving more attributes and methods; something that comes from the algorithms used to process and render the relationships among textural items.

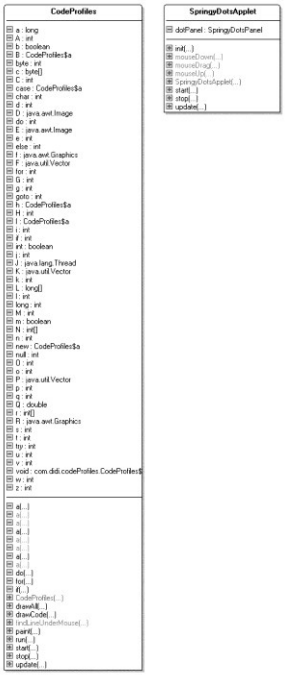

Figure 6. *Class diagram comparisons.*

Figure 6 demonstrates as well the power of class diagrams for ease of comparison. Instead of attempting to compare lines of source code, these graphical representations immediately provide a high level accounting of each software class.

The single class structure of these artworks raises a question about how to categorize artwork in general. From a software size perspective (c.f. Table 2), this artwork falls into the trivial category. Trivial programs tend to be written on a part-time basis, may be used for a single purpose or to demonstrate a concept, and are sparsely documented. From an artistic perspective, we may prefer to call these works sketches, since their purposes are similar in nature to artist sketches. In addition, the ability to create a digital sketch has been facilitated by the development of the Processing programming language [15], a streamlined version of Java, now employed to teach programming in digital art courses. With its extensive libraries for sound, graphics, video, image processing, interaction, and sensor control, it is possible to construct rich artistic experiences with a minimal amount of programing [16]. Indeed, in the decade since the *CODeDOC* exhibition, the amount of Processing-based digital artwork has exploded, some of which has found its way into major museum collections. For example, Philip Worthington's work *Shadow Monsters* is in the collection of New York's Museum of Modern Art [17].

The single class structure of these artworks raises the issue of maintaining monolithic programs as well. In general, monolithic programs tend to be difficult and time consuming to maintain, because their design and behaviors can only be understood by detailed analysis of their source coded; a problem exacerbated as program size increases, and as documentation within the software decreases. Since the *CODeDOC* artworks considered here reside at the trivial end of the software spectrum, and they only rely on the

functionality intrinsic to the Java programming language as well, their low level designs and behaviors can be deconstructed and represented using UML state charts and activity diagrams. However, all this comes at a greater expenditure of time and effort. Likewise, conservation of monolithic digital art written in the Processing language will face similar issues, in particular their reliance on libraries external to the core programming language - libraries that may have both a short lifetime, and be poorly documented.

Assessment of such concerns for monolithic code is found in our archaeological approach. The creation of class diagrams from source code provides a means for quickly evaluating the kinds of data and functionality a program contains. Comparing class diagrams among artworks allows the conservator to categorize collections of works by their functional requirements, and evaluate their reliance on external resources. Finally, based on these considerations, it should be possible to appraise an artwork's suitability for conservation.

Table 2: Software Size Categories

Category	Programmers	Duration	Size (Lines of Code)
Extremely Large	> 200	> 6 yrs.	>1,000,000
Very Large	20 - 200	3 - 6 yrs.	100,000 - 1,000,000
Large	5 - 20	2 - 3 yrs.	20,000 - 100,000
Medium	2 - 5	6 mo.– 2 yrs.	3,000 – 20,000
Small	1 - 2	1 - 6 mo.	500 - 3,000
Trivial	1	1 - 4 wks.	< 500

Summary

In this paper we have put forward the notion of applying principles of software archaeology (the systematic study of software systems through the recovery and examination of outstanding material evidence) to the long term conservation of code-based digital artwork through the application of the software engineering process, and the use of its artifacts to systematically organize and compare a collection of digital artworks. We have put forward the use of the "4+1 View Model" of Kruchten for analyzing digital artworks, and shown how this model can be translated into the formal documentation that is part of the Rationalized Unified Process, a standard software engineering methodology. In so doing, existing artisanal conservation procedures are transformed into a systematized process that can be integrated with traditional art conservation practice software, maintain a historical record, and select artifacts for preservation. Lastly, we applied our method to a collection of web-based artworks, comparing their architectures, and discussing their capacities for conservation. It was found that even though these works were considered "trivial" by contemporary software stands, the nature of their designs signals that significant effort may be required for their long term conservation.

References

[1] F.T. Marchese, "Conserving Digital Art for Deep Time," Leonardo, 44 (4), 302 -308 (2011).

[2] F.T. Marchese and M.P.K Shergill, 500 Year Documentation, Proc. DocEng'12 (Paris, France, September 4–7, 2012), pp. 157 -160 (2012).

[3] R.S. Pressman, Software Engineering: A Practitioner's Approach, 6th Edition (New York: McGraw-Hill, 2005).

[4] R.K. Ko, A Computer Scientist's Introductory Guide to Business Process Management (BPM), Crossroads,. 15 (4), 11–18 (2009).

[5] G. Booch, Software Archaeology, ACM OOPSLA (2008).

[6] E. Walker, "Tech Views," Jour. Softw.Tech, 8 (3), 1- 3 (2005).

[7] C. Larman, Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Iterative Development, 3rd Edition (Upper Saddle River, NJ: Prentice Hall, 2005).

[8] C. Alexander, A Pattern Language: Towns, Buildings, Construction (Oxford University Press, 1977).

[9] E.J. Chikofsky and J.H. Cross, "Reverse Engineering and Design Recovery: a Taxonomy," IEEE Software, 7(1), 13-17 (1990).

[10] P. Kruchten, "Architectural Blueprints — The '4+1' View Model of Software Architecture," IEEE Software, 12 (6), 42-50 (1995, November).

[11] P. Kruchten, The Rational Unified Process-An Introduction, 3rd ed. (Addison-Wesley, 2003).

[12] J. F. Dumas and J.C. Redish. A Practical Guide to Usability Testing, 1st ed. (Greenwood Publishing Group Inc., Westport, CT, USA, 1993).

[13] L. Khaled, A Comparison Between UML Tools, Proc. Second International Conference on Environmental and Computer Science, 111-114 (2009).

[14] Whitney Museum of American Art, "CODeDOC," (2002). Retrieved Feb. 1, 2013 from http://artport.whitney.org/commissions/codedoc/.

[15] C. Reas, B. Fry, and J. Maeda, Processing: A Programming Handbook for Visual Designers and Artists, 1st ed.(The MIT Press, 2007).

[16] D. O'Sullivan and T. Igoe, Physical Computing: Sensing and Controlling the Physical World with Computers. (Boston, MA, Course Technology Press, 2004).

[17] P. Worthington, "Shadow Monsters," (2004). Retrieved Feb. 1, 2013 from http://www.moma.org/collection/object.php?object_id=110196

Author Biography

Francis T. Marchese received his BS in natural science from Niagara University (1971), MS in chemistry from Youngstown State University (1973), and PhD in theoretical chemistry from the University of Cincinnati (1979). Professor of computer science at Pace University, his research spans visual computing and conservation of digital art. He is founder and co-director of the Pace Digital Gallery, and an associate editor of the ACM Journal on Computers and Cultural Heritage.