# Conserving Software-based Artwork through Software Engineering

Francis T. Marchese
Department of Computer Science
Pace University
New York, NY, USA
fmarchese@pace.edu

***Abstract* — A long term strategy is offered for conserving software-based digital art in perpetuity based on software engineering practice. Software engineering is a rigorous, formalized practice approach to preservation that engages all stakeholders, including: artists, curators, and conservators; offering a breadth of methodologies and degrees of rigor that may be adapted by organizations of all sizes. In this paper the focus is on the software engineering maintenance process, and how it converts digital art into formats that should make it preservable and displayable in the deep future.**

***Keywords — digital art; software engineering; software maintenance; conservation.***

## I. INTRODUCTION

Museum visitors today can regularly view 500 year old art by Renaissance masters. Museum visitors 500 years in the future should be afforded the same opportunity for software-based artwork created today. Any conservation strategy that attempts to address issues related to the perpetual maintenance of digital artwork must contend with its ephemeral nature, the impermanence of the technological substrates upon which it is based, and its place within a museum collection that inevitably will grow in size and diversity over centuries. Because museums collect more artwork than can be possibly exhibited at any time, all art must be expected to revolve from storage into galleries in accord with curatorial discretion, with the exception of works that either define a museum's collection or are critical to the art canon. Given that decades may pass before artworks are reinstalled, it is probable that attempts to do so will fail because such works will have reached technical obsolescence.

Our solution to this problem is based on software engineering, the formal process of applying a systematic, disciplined, quantifiable approach to problem analysis, system and software design, its development, operation, and maintenance [1]. Software engineering emphasizes both the software as product, and the processes that create and maintain it. Specifically, the software engineering process transforms existing artisanal preservation procedures into a schematized process integrable into traditional art conservation practice. Software engineering plays an essential role in software development within the aerospace, high technology, and financial service industries. Since its processes are extensions of the standard business life-cycle [2], its tools and techniques may be integrated into a museum's conservation practice. Software engineering can engage all stakeholders including artists, curators, conservators, installers, maintainers, museum directors, art historians, and viewers; and can reflect and integrate this process into a museum's current best practices.

Here we focus on the software engineering maintenance process [3]. Software maintenance is the "process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment" [4]. It is this software engineering process that makes it possible for existing products to continue to be used. Maintenance is the longest process in the software engineering lifecycle, typically consuming up to 80% of the total effort expended on a software product that may exhibit decades of use. We consider how a mainstream strategy may be used to convert a digital artwork into formats that should make it preservable and displayable in the deep future as part of its maintenance.

## II. LEGACY SYSTEMS AND MAINTENANCE

When a museum acquires a digital artwork it is assumed that the work will be exhibited as is, with no further enhancements. This work and its conceptual and technological foundations are now frozen in time. In effect, it is a legacy system [5]. From a software maintenance perspective, legacy systems are considered to be socio-technical in nature, recognizing that the human factor is integral to the system's function. The components of a legacy system may be envisioned as a stack of encapsulated layers, with each layer depending on the layer immediately below for its resources, and interfacing with that layer. The highest layer is the *Artistic Experience* which directly engages the viewer. Below is the *Artistic Software* itself, that generates this experience. The *Artistic Software* layer relies on the services supplied by the *Support Software* layer to run (e.g. operating system, database managers, networking protocols, etc.). And the *Hardware* layer provides all physical components required by the layers above (e.g. CPU, video card, network adapter, sensors, actuators, etc.).

In principle, it should be possible to replace a layer in the system leaving the other layers unchanged. However, in practice, this simple encapsulation rarely succeeds. Changes to one layer of the system may require subsequent modifications

978-1-4799-3169-9/13/$31.00 ©2013 IEEE

to layers both above and below the altered level. At the *Hardware* level, it is often impossible to maintain hardware interfaces to the *Support Software* level when a fundamental change in hardware is prescribed. Similarly, a required redeployment of the artwork from one operating system to another (say, Microsoft Windows to UNIX) would engender major changes to the application software itself. Given that software interfaces, formats, and protocols continually evolve; resources will either disappear or become redistributed; and computer hardware is guaranteed to become obsolete; at the very least, software-based artwork will need to be adapted to accommodate changes that arise for components within these layers.

The largest expenditure of effort during maintenance is the discovery process in which the source code is searched to exposes domain knowledge and an implementation strategy. Domain knowledge describes the environment in which the program functions, and provides a context for understanding the various characteristics of the software. Artistic theory and practice upon which the artwork is founded is such an example. Implementation strategy represents the way the system has been built, encompassing code-level knowledge (e.g. theories of data structures and algorithms) as expressed in the programming language used for the artwork. Because the code contains all data representations, and the fine details of its processing capabilities, extracting conceptual information is difficult, because each logical grouping of code must be interrogated to understand its purpose. Indeed, source code analysis requires maintainers to spend up to 60% of their time searching for the knowledge required to effectively conserve the software [7]. The interweaving of domain knowledge with implementation strategy makes understanding, maintaining, adapting, reusing and evolving the software difficult, time-consuming, error-prone, and hence expensive.

Software evolves as part of maintenance practice [8]. Evolving software is characterized by an increasing complexity that arises from the inevitable modification of its configuration over time to accommodate change. Complexity is exhibited in two ways. Design complexity, in which the artwork's overall configuration exhibits a composition, arrangement, and interconnection among its component parts that is poorly designed. As an example, consider a poorly designed architectural plan for a house in which rooms are oddly sized and arranged, making their utility and accessibility difficult. And code complexity, in which a program is difficult to understand. Here the programming logic may contain too many variables and control paths, or reveal obscure programming language constructs; making the code difficult to trace. For software that began its life inadequately designed and coded, enormous maintenance costs are possible, because these systems cannot easily support change [9].

In general, there are four types of maintenance that digital artwork may undergo [6]. Perfective maintenance improves software functionality in response to requested changes. Corrective maintenance corrects errors identified within the software. Adaptive maintenance alters the software in response to changes within the software environment. And preventative maintenance updates software to improve its future maintainability without changing its functionality. Preventative maintenance offers the prospect of transforming a digital artwork into a format that will improve its maintainability. The goal is to define the artwork independent of its implementation, that is, to represent the artwork at a conceptual level higher than source code. There are two key advantages. First, the artwork is completely defined at a level of abstraction that supports investigation of "what" the system does and "how" it does it without resorting to sifting through source code that may be poorly documented. For adaptive maintenance of an artwork, this means a more efficient method for not only finding components of the artwork that require modification, but also understanding their function. Second, it places the artwork's conceptual model and design in a time-independent state. When the museum received the original artwork, it was implemented in the technology of its time. This placed it in a time-dependent state that diminishes its ability to be adaptable to the technology of a distant future time period, because its design and implementation become less compatible with future technological advances as time progresses. In contrast, a time-independent representation can be translated into whatever technology is available that befits its implementation, because its overall structural and behavioral schemes, and processing details (e.g. data structures, algorithms) are specified in a collection of representations that are devised for human understanding; and not bound to any technological substrate. Just as an architectural design of a home can be constructed employing a variety of building materials, these conceptual representations of software may be transformed into functioning software employing programming languages.

For the preventive maintenance process to be successful it must work from an expanded definition of software that not only includes source code, executable programs, and data, but also analysis and design documents; operations, system, installation, and implementation manuals; and any other documentation relevant to its functionality. Such a breadth of documentation improves the probability of the preventive maintenance process recovering the software's conceptual foundation and domain knowledge. This process is applicable to the conservation of software-based art as well, because the goal of contemporary digital art conservation practice is to capture an artwork's essential properties, so it may be understood, maintained, and displayed at a future date [10] by employing an extended set of documentation [11] – [13]. This expanded notion of an artwork's identity as a collection of concepts and artifacts mirrors preventative maintenance, and is the starting point for a formal identification and documentation employing the principles and practices from reverse engineering.

## III. REVERSE ENGINEERING

Reverse engineering is a maintenance strategy in which a system is analyzed to identify its component parts, and the interrelationships among those parts for the purpose of either creating representations of the system in another form or representations at a higher level of abstraction [14][15]. Reverse engineering commonly involves recovering system design. Its goal as a maintenance strategy is to capture an artwork's identity, and transform it into a maintainable collection of representations that embody its functional, design,

interface, and environmental requirements. Fundamental to reverse engineering are the creation and maintenance of documentation to support software evolution, communication, preservation of system and institutional memory, and processes such as system auditing. Here, documentation supplies comprehensive information about an artwork's capabilities, architecture, design details, features, and limitations.

The documentation used here is associated with the Rational Unified Process (RUP) [16], a general object oriented software engineering methodology that has been used to reverse engineer legacy systems [17]. Its documentation encompasses the following five components: *Functional*, *Architecture/Design, Technical, End User,* and *Supplementary Materials*. *Functional Requirements* are the artwork's conceptual foundation that communicates what it is supposed to do. The *Architecture/Design* component provides an overview of the software that includes a definition of how it relates to its environment, and construction principles used in design of the software components. It is typically organized as a collection of diagrams or charts to show its parts and their interconnections. *Technical* documentation encompasses source code, algorithms, and interface documents. *End User* documents include installation, maintenance, and user manuals. And *Supplementary Materials* contains anything else related to the system, including: legal documents, design histories, interviews, scholarly books, installation plans, drawings, models, documentary videos, websites, etc. Taken as whole, this documentation supports both abstract and detailed descriptions of static artwork structure and its dynamic processes, providing a diversity of representations to satisfy all stakeholders.

RUP documentation is the basis for a theoretical model that is used to both examine software's architecture and formally deconstruct it. Software architecture represents the highest-level abstraction of a software system's structure. It can be defined as the set of constructs needed to reason about the software system, comprising its component parts, the relations between them, and the properties of both components and relations. The approach used here relies on the well-known "4+1 View Model" of Kruchten [18].

The "4+1 View Model" consists of *logical*, *process*, *physical*, and *development* views (c.f. Fig. 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. *Logical* and *process views* comprise the *Architecture/Design* documentation, and are 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 [16]. 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

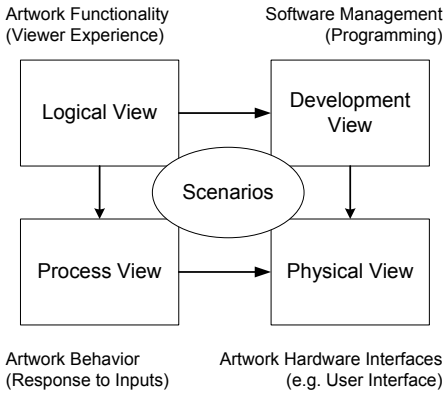


Fig. 1. "4+1 View Model"

attributes, are not system features, but instead 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. If an artwork's source code exists, then it constitutes most of the *Technical* documentation, and contributes to the *development view*.

*Scenarios* represent the "plus one" view, binding together the four architectural components. *Scenarios* describe how one or more entities (e.g. artwork viewers) 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. *Scenarios*, along with *process* and *logical views* are transformed into the *development view* as part of the software development by coding them in appropriate programming languages. This is a forward engineering process where software is packaged naturally into subsystems that are organized into a hierarchy of layers, each layer providing a narrow and well-defined interface to the layers above it.

The process of recovering an artwork's character alternates between reverse and forward engineering activities in a spiral fashion. First an artwork's attributes are recovered through reverse engineering, then validated and tested through forward engineering until the model's design coalesces into a consistent whole (c.f. [17]). The process proceeds as follows: evaluate the scope of the reverse engineering project; construct the abstract models (here the "4+1 View Model"); and recover the architecture of the software. Once the problem's scope has been determined, use cases are identified related to a viewer's engagement with the artwork. Use cases represent the distinct system behaviors associated with one or more actors external to the system. They are the analysis model's foundation, helping define the artwork's architecture, and the starting point for recovering user *scenarios*, also called *use case scenarios,* which represent the fine structure of behaviors. Use cases and *scenarios* are found in an artwork's *Supplementary Materials*, where its temporal designs may have been depicted in storyboards, videos, and such. Otherwise, use cases and

*scenarios* may be captured through observation of artwork engagement. This precise dialog between artwork and viewer may be extracted employing formal usability techniques [19]. Finally, recovery of an artwork's architecture proceeds from an initial architectural design of its structure and behaviors based on use cases. It is validated through program implementation, where each use case is tested to understand the sequence of coding steps required for successful deployment.

## IV. Discussion and Conclusions

This paper has put forward a strategy for conserving software-based digital art, grounded in standard software engineering practices. It employs the Rational Unified Process (RUP) to transform a digital artwork into a model that is independent of technology and time. As such, it addresses the biggest challenge facing conservators – the legacy dilemma, where it is expensive and risky to replace a legacy system, and expensive to maintain it too. It does so by generating a model that can be consulted and referenced throughout the artwork's extended lifetime. This model may be used for either understanding what parts of the work need to be adapted to new technologies, so it may be displayed in the near future; or completely reconstructed outright in the technology of the distant future. In this way, software engineering methods address the "migration" approach to software preservation, where the software object is adapted to the next software or hardware platform. And, once an artwork is transformed into the "4+1 View Model," any discovery process need not be subjected to the time intensive task of sifting through source code to gather knowledge about "what" or "how" the software system performs a task. Instead, consulting the conceptual model affords significant savings in maintenance effort.

The RUP methodology possesses two distinct advantages over the code-to-code translation process. First, Dugerdil's [17] procedure works from a worst case scenario where the original software developers are no longer available to provide information, and no reliable documentation exists – including source code. This is a genuine possibility, given that many artists will not release source code to museums. As such, digital artwork reengineering must proceed from a combination of interrogation of the functioning artwork and exploration of RUP's *Supplementary Materials*. Second, Dugerdil's procedure and similar approaches have already been shown to work in information system reengineering, so the process of creating an appropriate maintenance strategy for digital art conservation becomes an act of adapting a standard technique instead of creating anew.

Perpetual maintenance of only one artwork has been considered. But museums are expected to conserve hundreds or thousands more from different eras, where each individual artwork embodies the technology of its time. Translation of these artworks into time-independent representations such as Kruchten's "4+1 View Model" via RUP allows groups of artworks to be assessed as a whole. Such an approach provides bases for formal software risk management [20] and the development of evolvable conservation strategies for collections as a whole.

Finally, conservation of an artwork's hardware should be addressed. An artwork's hardware requirements are extracted as part of RUP. Given that the long term conservation of computer hardware remains problematic, RUP documentation would be essential to the redesign and construction of an artwork's hardware in the distant future. This approach remains open for exploration.

## References


[1] R.S. Pressman, Software Engineering: A Practitioner's Approach, 6th ed. New York: McGraw-Hill, 2005.

[2] R.K. Ko, "A computer scientist's introductory guide to business process management (BPM)," Crossroads, vol. 15, no. 4, pp.1 – 18, 2009.

[3] K.H. Bennett and V.T Rajlich, "Software maintenance and evolution: a roadmap," in Proceedings of the Conference on The Future of Software Engineering (ICSE '00). New York: ACM, 2000, pp.73-87.

[4] IEEE, "IEEE Standard Glossary of Software Engineering Terminology," IEEE Std. 610.12-1990. Institute of Electrical and Electronics Engineers, 1990.

[5] J. Ransom, I. Sommerville, and I. Warren, "A method for assessing legacy systems for evolution," in Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR 98), March 8–11, 1998 (Washington, DC: IEEE Computer Society, 1998), pp. 128 - 134.

[6] B.P. Lientz and E.B. Swanson, Software Maintenance Management, Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 1980.

[7] P. Selfridge and R. Brachman, "Supporting a knowledge-based Software information system with a large code database, in Proceedings of the AAAI-90 Workshop on Knowledge-Base Management, 1990.

[8] N.H. Madhavji, J. Fernandez-Ramil, and D. Perry, Software Evolution and Feedback: Theory and Practice. John Wiley & Sons, 2006.

[9] D. Rowe, J. Leaney, and D. Lowe, "Defining systems evolvability - a taxonomy of change," International Conference and Workshop: Engineering of Computer-Based Systems, 1994, pp. 541-545.

[10] P. Laurenson, "Authenticity, change and loss in the conservation of time-based media installations," Tate Papers, Autumn (2006). Retrieved June 30, 2013 from http://www.tate.org.uk/research/publications/tate-papers/authenticity-change-and-loss-conservation-time-based-media.

[11] A. Depocas, J. Ippolito, and C. Jones, eds, Variable Media Approach, Guggenheim Museum Publications and The Daniel Langlois Foundation for Art, Science, and Technology, 2003.

[12] ERPANET, "The archiving and preservation of born-digital art workshop," Briefing Paper for the ERPANET Workshop on Preservation of Digital Art, 2004. Retrieved June 30, 2013 from http://www.erpanet.org/events/2004/glasgowart/briefingpaper.pdf.

[13] T.A. Yeung, S. Carpendale and S. Greenberg, "Preservation of art in the digital realm," in The Proceedings of iPRES2008: The Fifth International Conference on Digital Preservation, (London: British Library, 2008).

[14] E. Walker, "Tech views," Jour. Softw.Tech, vol. 8, no. 3, pp. 1- 3, 2005.

[15] E.J. Chikofsky and J.H. Cross, "Reverse engineering and design recovery: a taxonomy," IEEE Software, vol. 7, no. 1, pp. 13-17, 1990.

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

[17] P. Dugerdil, "Using RUP to reverse engineer a legacy system," The Rational Edge, IBM, September 2006.

[18] P. Kruchten, "Architectural blueprints — the '4+1' view model of software architecture," IEEE Software, vol. 12, no. 6, November 1995, pp. 42-50.

[19] J. F. Dumas and J.C. Redish, A Practical Guide to Usability Testing, Westport, CT : Greenwood Publishing Group Inc., 1993.

[20] B.W. Boehm. "Software risk management: principles and practices," IEEE Softw., vol. 8, no. 1, January 1991, 32-41.