Masthead Logo

School of Information Student Research Journal

Volume 2 | Issue 2

Article 5

January 2013

Consider the Source: The Value of Source Code to Digital Preservation Strategies

Michel Castagné
The University of British Columbia, castagne@gmail.com

Follow this and additional works at: https://scholarworks.sjsu.edu/ischoolsrj

Part of the Library and Information Science Commons

Recommended Citation
Castagné, M. (2013). Consider the source: The value of source code to digital preservation strategies. *SLIS Student Research Journal, 2*(2). Retrieved from http://scholarworks.sjsu.edu/slissrj/vol2/iss2/5.

This article is brought to you by the open access Journals at SJSU ScholarWorks. It has been accepted for inclusion in School of Information Student Research Journal by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

Consider the Source: The Value of Source Code to Digital Preservation Strategies

Abstract
One of the major challenges in the digital preservation field is the difficulty of ensuring long-term access to digital objects, especially in cases when the software that was used to create an object is no longer current. Software source code has a human-readable, documentary structure that makes it an overlooked aspect of digital preservation strategies, in addition to a valuable component for the records of modern computing history. The author surveys several approaches to software preservation and finds that, by supporting open source initiatives, digital libraries can improve their ability to preserve access to their collections for future generations.

Keywords
source code, open source, digital preservation, software preservation

About Author
Michel Castagné is a Master of Library and Information Studies candidate at the University of British Columbia. He specializes in digital libraries and preservation in an academic setting, as well as designing effective information architecture and databases.

This article is available in School of Information Student Research Journal: https://scholarworks.sjsu.edu/ischoolsrj/vol2/iss2/5

INTRODUCTION

When faced with a screen of technical software instructions to a computer (known to programmers as source code), even in a language as common as HyperText Markup Language (HTML), it is not hard to imagine how the average computer user might see the strings of verbs, abbreviations, slashes, and semicolons as little more than technical gibberish, and quickly close the editor. As long as the program or document works as described, of what benefit is peering into its internal structure? Even from a digital preservation standpoint, a similar argument might be raised: As long as file format registries are maintained and digital objects are migrated when necessary, of what benefit is the cryptic source code of millions of projects? This approach, however, does little service to the nature and value of source code, which can be seen as integral to durable software preservation, in terms of both recording modern computing history and as part of a strategy to maintain access to digital objects.

Although the burgeoning digital preservation field has been the source of a great deal of research activity in the past decade—including the formation of the Preservation Metadata: Implementation Strategies (PREMIS)[1] working group and a comprehensive reference model for designing an Open Archival Information System (OAIS)[2]—software preservation is a sub-field that has yet to be thoroughly explored. Matthews, Shaon, Bicarregui, and Jones (2010) suggest that there is a need for further "conceptual analysis," as well as the development of experience and tools for software preservation. The debate over why and how software should be preserved has several perspectives, often centered around the need to defend against format obsolescence. This article will make a survey of the issue, as well as examine the current approaches to software preservation with a view towards how source code, and the open source community in particular, can assume an important role in the digital preservation field.

DEFINITIONS AND MODELS

Definitions

A definition of "software" can encompass a surprisingly large amount of digital bits. The Institute of Electrical and Electronics Engineers (IEEE) Standard Glossary of Software Engineering Terminology defines a software product as the "complete set of computer programs, procedures, and possibly associated documentation and data designated for delivery to a user" ("Software product," 1990), while a "software item" is described as "source code, object code, job

1 PREMIS: http://www.loc.gov/standards/premis/

2 Reference Model for an OAIS: http://public.ccsds.org/publications/archive/650x0b1.pdf

control code, control data, or a collection of these items" ("Software item," 1990), or in other words, an identifiable component of a software product. Examples of software can include everything from system software, like an operating system or device driver, to programming software, such as a compiler or debugger, in addition to application software, such as web browsers, word processors, and graphic design programs. The form of software an end user typically encounters is the executable program or, in IEEE's vocabulary, "object program" ("Object program," 1990). This is compiled from human-readable source code, which is usually written by a programmer in plain text format and often annotated with explanatory comments, so that any programmer who studies the source code can learn more about how the software functions and any particular quirks it might have. Van de Vanter (2002) calls this semantic dimension of source code, including use of white space and choice of names, its "documentary structure" (p. 1).

In digital preservation, software often assumes a secondary role as a tool to view digital objects in a collection (Matthews, McIlwrath, Giaretta, & Conway, 2008). But if a software product produces a research result inaccurately, displays an object incorrectly, or ceases to function altogether, the relevant digital object or result is effectively lost, sometimes without the user even noticing. This can be the result of running an unsupported program in a new operating environment with changed or missing dependencies, or a manufacturer's decision to no longer support a format (Sandborn, 2007, p. 886). Software can also have very complex and dynamic behavior; thus, simple strategies such as preserving a copy of the object program are inadequate. There is a very clear need to preserve not only digital objects, but reliable access to these objects, which means adopting one or more approaches toward software preservation.

Models

In the United Kingdom, important research on the topic has taken place in the past decade, notably by the Software Sustainability Institute[3] and the e-Science Department,[4] with a great deal of funding for projects related to digital preservation and curation coming from the Joint Information Systems Committee (JISC), a non-departmental public body that supports higher education and research in Information and Communications Technology. Two related key studies that have emerged recently are Matthews et al. (2008) and Matthews et al. (2010). The first study proposed supplements to a draft of the InSPECT[5] report and the

3 The Software Sustainability Institute: http://software.ac.uk/

4 e-Science Department in the Science and Technologies Facilities Council in Oxford: http://www.stfc.ac.uk/e-Science

5 Investigating Significant Properties of Electronic Content: http://www.significantproperties.org.uk/

latter extends this research to propose an overall framework for software preservation, which includes a performance model, a conceptual model of software components based on the Functional Requirements for Bibliographic Records (FRBR), and an OAIS-based categorization of the significant properties of software.

First, Matthews et al. (2010) outline four major aspects of software preservation: storage, retrieval, reconstruction, and replay (pp. 92–93). The "storage" and "retrieval" dimensions are dependent on the digital preservation strategy of the repository. The authors remain neutral on this subject, but point out that it should at least ensure secure and authentic maintenance of the digital objects, with the inclusion of sufficient metadata for retrieval purposes. "Reconstruction" refers to the ability of a repository to reinstall or rebuild a piece of software from what has been stored, while "replay" refers to how well the software performs in relation to its original behavior.

Performance Model

The performance model relies on a concept of "adequacy," that is, whether the replay of a software product conforms to certain designated significant properties within an acceptable tolerance (p. 94). These significant properties are based on how the reconstructed software processes and displays data to the user. Matthews et al. (2010) include a flow chart of their performance model to illustrate the relationship between these concepts (see Fig. 1). In this chart, the software source must be processed before the software can perform. Its performance is directly linked to its ability to process input data, leading to performance of the data, which is then viewed by a user. The user interacts with the software, thus changing the performance of its input data.

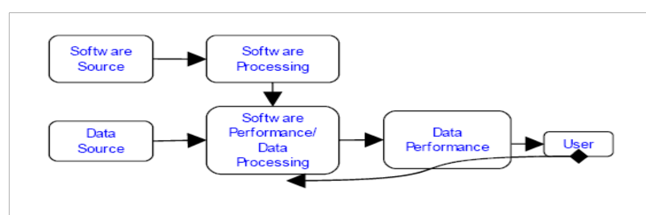

Fig. 1. Performance model of software and its input data (Matthews et al., 2010, p. 95).

Conceptual Model

The FRBR-based conceptual model is comprised of four entities that describe a "complete software system": product, version, variant, and instance. This is parallel to the FRBR entities work, expression, manifestation, and item. As a simple example, LibreOffice 3.6.2 for Mac OS X (PPC) can be broken down as:

- Product: LibreOffice
- Version: 3.6.2
- Variant: Mac OS X (PPC)
- Instance: An actual copy of the software system on a particular computer

Properties Model

The preservation properties model looks at seven main categories of software features and relates the categories to the nearest OAIS equivalent, which have been placed in parentheses here. These are: functionality *(descriptive information)*, software composition *(representation information/preservation description information)*, provenance and ownership *(provenance information)*, user interaction *(significant properties)*, software environment *(representation information)*, software architecture *(representation information)*, and operating performance *(significant properties)* (pp. 98–100). That the OAIS model falls short of comprehensively defining the significant properties of software, such as user interaction and operating performance, emphasizes its current inadequacy for software preservation.

APPROACHES TO PRESERVATION

As a software preservation framework has yet to be agreed upon and established, a number of techniques have been debated. Hong, Crouch, Hettrick, Parkinson, and Shreeve (2010) have discussed seven of these techniques, each of which has its place:

- Technical preservation
- Emulation
- Migration
- Cultivation
- Hibernation
- Deprecation
- Procrastination

Technical Preservation

Technical preservation involves the intention to maintain software and hardware in the same functional state, which usually implies purchasing spare parts when needed. Naturally, this often becomes costlier as time goes on and unusual parts become harder to find. A good example of a facility pursuing technical preservation would be the Computer History Museum in Mountain View, California, which is home to "one of the largest international collections of computing artifacts in the world," including hardware, software, documents, and ephemera (Computer History Museum, n.d.). Applying Van de Vanter's observation of the documentary structure of source code, software can be seen as a cultural artifact (in addition to being a computing artifact) and source code can be seen as the "intellectual essence" of this artifact (Shustek, 2006, p. 112). Zabolitzky (2002) notes that the source code is the only artifact containing the full information regarding the functioning of a software product, and everything else is "essentially hearsay" (p. 4). He also suggests that the availability of the source code of an operating system makes parts replacement much easier, as the code can be adjusted to allow interfacing with a different piece of hardware. Even if a software product no longer serves any practical purpose, this primary document, in addition to any related documentation or specification, is still of importance to current or future historians studying the evolution of software, and this needs to be taken into consideration by digital curators.

Emulation

It is also possible to emulate aging hardware by writing software that mimics its architecture and processes. For instance, an emulator such as Charon[6] allows a user to run various Digital Equipment Corporation platforms as virtual machines on modern personal computers, encapsulating a guest operating system within a host operating system. These types of emulators can facilitate migration and viewing of data from an old system to a virtual machine running a legacy operating system and any related software, provided, of course, that it has been preserved well. Emulation has been championed in the digital preservation field since the 1990s, notably by the computer scientist Jeff Rothenberg.[7] In order to, in turn, preserve emulation software—without creating an endless chain of emulators—Rothenberg proposed that a layer be created between the emulator and the platform, called an Emulation Virtual Machine, which would make the emulator platform-independent for the foreseeable future (Van der Hoeven & Van

---

6 Charon: http://www.winvms.com/

7 Such as his widely cited article from 1995, "Ensuring the Longevity of Digital Documents," published in *Scientific American, 272*(1), 42–47.

Wijngaarden, 2005). While in theory this seems like an ideal solution, his design for the concept mostly encountered skepticism. In addition to being extremely difficult to program, Bearman (1999) considers emulation to be disproportional to the needs of an archive when migration would be adequate, because he considers Rothenberg's criticisms of migration (discussed further on) to be ill-founded and without strong evidence.

That is not to say that long-term emulation no longer garners interest. Gladney and Lorie (2005) cite Bearman's criticism and note that, while it has not been refuted, they propose a more technically feasible approach: the Universal Virtual Computer. While an in-depth treatment of this concept is not within the range of this discussion, it is worth noting that Van der Hoeven, Lohman, and Verdegem (2007) have built on Gladney and Lorie's and Rothenberg's ideas to develop an open source modular emulator written in Java called Dioscuri,[8] which consists of a number of flexible, platform-independent components that emulate a simple x86 computer[9] and can transfer data between the real and emulated environment.

Migration

Migration, as alluded to above, means transporting information from one type of system or format to another. Hoorens, Rothenberg, Van Orange, Van der Mandele, and Levitt (2007) state that format migration leads to “cumulative corruption and degradation,” as data is forced into each new “Procrustean bed” of a format (p. *x*). Evocative language aside, while this can be true in poorly planned automated migration scenarios, much like how successive runs through a machine translator can render a sentence into nonsense, software migration does not have to be not quite as random and inevitable. This type of migration involves rewriting and recompiling source code for another operating environment (Hong et al., 2010). The rewrite could range from a small tweak to a complete overhaul of the code in a new programming language. Migration can be greatly facilitated by way of the fourth option for software preservation listed by Hong et al.: cultivation.

Cultivation

Cultivation involves opening the software to outside development by sharing the source code. This can mean adopting an open source license,[10] such as the widely

8 Dioscuri: http://dioscuri.sourceforge.net/

9 At the time of writing, Dioscuri is only capable of running 16-bit operating systems, like MS-DOS. Development is under way to add 32-bit functionality and support Windows 3.11.

10 The Open Source Initiative provides an extensive list: http://www.opensource.org/licenses/category

used General Public License, or simply sharing the code privately with a group of developers. As mentioned earlier, source code has a documentary structure, which makes it a strong candidate for one of the chief semantic bearers when it comes to preserving software (Van de Vanter, 2002). By sharing code, programmers are encouraged to provide meaningful documentation of their work to make it comprehensible to others. A piece of software can then be analyzed by another programmer who can fix bugs or extend its original capabilities.

A compelling case can be made for adopting an open source license. First, a publicly available source code will help future programmers avoid the immense challenges related to reverse-engineering from the object program.[11] Further, in addition to making emulation and software migration more feasible (Zabolitzky, 2002), backwards compatibility is a high priority in the open source community (Rosenthal, 2010, p. 3). When it comes to rendering an obsolete format, the source code of an old renderer is likely to be vastly more useful than the information contained in a format registry (Rosenthal, 2010, p. 5). Rosenthal also notes that, if an open source renderer does not exist, it is unlikely that a format registry is even aware of the format (p. 5). One of the main hurdles in this open source approach, however, is that source code is considered by many companies to be a trade secret, and it can be challenging to convince a software manufacturer that there is any reason to share these secrets with anyone. Alternatively, the Library of Congress suggests that those concerned with exposing their code make an escrow deposit of documentation and source code related to "rendering software, validation tools, and software development kits" with a trusted archive (Library of Congress, 2007), a sort of hibernation.

Hibernation

Hibernation involves placing the entire software product (including documentation and significant properties) into storage, to be re-examined at a later date when it needs to be used. In this case, open source software is at an advantage, because preparation is likely to be already near completion (Hong et al., 2010). Source code itself would again be useful, as future programmers would find it much easier to migrate or emulate the software if the structure is at hand.

Deprecation and Procrastination

The final two approaches—deprecation and procrastination—are not preservation strategies as such and will not be discussed in depth here. In brief, deprecation is a way of noting that a specific software feature or practice will no longer be supported in the future, whereas procrastination means to "do nothing" (Hong et

11 A field known as software archeology.

al., 2010). Deprecation, at the very least, provides some degree of notice that interested parties should consider ways of adapting to the change.

FURTHER INFORMATION AND CONCLUSION

In light of this discussion, there are a number of current projects that contribute to the preservation of source code that are worthy of discussion. Foremost are the many open source software (OSS) repositories,[12] such as SourceForge,[13] Launchpad,[14] and GitHub,[15] which offer numerous preservation-friendly features to developers, such as version control and bug tracking, and can often host both public and private code. In the United Kingdom, the Software Sustainability Institute promotes a number of user-friendly guides[16] on how to make software durable, in addition to their research on software preservation. JISC also funds OSS Watch, an open source software advisory service that provides advice on building an open development community. There are a number of European Union-sponsored projects, including the Open Planets Foundation,[17] which provides practical digital preservation expertise to its members, and the Keeping Emulation Environments Portable (KEEP) Project,[18] which focuses on building a stable foundation for Europe's digital heritage. The IEEE also holds many annual conferences related to software engineering, two of which are of particular interest: the International Conference on Software Maintenance (ICSM)[19] and the International Working Conference on Source Code Analysis and Manipulation (SCAM).[20] All of these projects could use support, even in such a basic way as spreading awareness about software preservation issues.

One of the major challenges in the digital preservation field is the difficulty of ensuring long-term access to digital objects, especially in cases when the software that was used to create an object is no longer current. Zabolitzky (2002) notes that a proactive approach to software preservation is necessary, and that passive gathering of software is not likely to produce a comprehensive and relevant collection, nor can it ensure that the software will perform accurately when needed (p. 8). Access to source code is a major factor in a preservationist's ability to

12 An extensive list, comparing the features of each: http://en.wikipedia.org/wiki/Comparison_of_open_source_software_hosting_facilities
13 SourceForge: http://sourceforge.net/
14 Launchpad: https://launchpad.net/
15 GitHub: https://github.com/
16 Resources for developers: http://software.ac.uk/resources/guides
17 Based on a previous project called Preservation and Long-term Access through Networked Services (PLANETS): http://www.openplanetsfoundation.org/
18 KEEP: http://www.keep-project.eu/ezpub2/index.php
19 ICSM: http://conferences.computer.org/icsm/
20 SCAM: http://www.ieee-scam.org/

recreate adequate software performance and, to this end, open standards must be actively promoted, regardless of which preservation approach currently seems best. Additional requirements include a strong digital preservation framework that is tailored to the growing complexity of software and a continued discussion of ways to protect the intellectual property of software developers while preserving access to the work of software users.

References

Bearman, D. (1999). Reality and chimeras in the preservation of electronic records. *D-Lib Magazine, 5*(4). Retrieved from http://www.dlib.org/dlib/april99/bearman/04bearman.html

Computer History Museum. (n.d.). Backgrounder. Retrieved from http://www.computerhistory.org/press/backgrounder/

Gladney, H. M., & Lorie, R. A. (2005). Trustworthy 100-year digital objects: Durable encoding for when it's too late to ask. *ACM Transactions on Information Systems, 23*(3), 299–324.

Hong, N. C., Crouch, S., Hettrick, S., Parkinson, T., & Shreeve, M. (2010). *Software preservation benefits framework.* [PDF document]. Retrieved from http://www.software.ac.uk/attach/SoftwarePreservationBenefitsFramework.pdf

Hoorens, S., Rothenberg, J., Van Orange, C., Van der Mandele, M., & Levitt, R. (2007). *Addressing the uncertain future of preserving the past: Towards a robust strategy for digital archiving and preservation.* [PDF document]. Retrieved from http://www.rand.org/pubs/technical_reports/TR510.html

Library of Congress. (2007). Sustainability factors. Retrieved from http://www.digitalpreservation.gov/formats/sustain/sustain.shtml

Matthews, B., McIlwrath, B., Giaretta, D., & Conway, E. (2008). *The significant properties of software: A study.* [PDF document]. Retrieved from http://www.jisc.ac.uk/media/documents/programmes/preservation/spsoftware_report_redacted.pdf

Matthews, B., Shaon, A., Bicarregui, J., & Jones, C. (2010). A framework for software preservation. *The International Journal of Digital Curation, 5*(1), 91–105.

Object program. (1990). In *IEEE Standard Glossary of Software Engineering Terminology* (IEEE Std 610.12-1990). Retrieved from http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=159342&isnumber=4148

Rosenthal, D. S. H. (2010). Format obsolescence: Assessing the threat and the defenses. *Library Hi Tech, 28*(2), 195–210.

Rosenthal, D. S. H., Robertson, T., Lipkis, T., Reich, V., & Morabito, S. (2005). Requirements for digital preservation systems: A bottom-up approach. *D-Lib Magazine, 11*(11). Retrieved from http://www.dlib.org/dlib/november05/rosenthal/11rosenthal.html

Sandborn, P. A. (2007). Editorial: Software obsolescence—complicating the part and technology obsolescence management problem. *IEEE Transactions on Components and Packaging Technologies, 30*(4), 886–888.

Shustek, L. (2006). What should we collect to preserve the history of software? *IEEE Annals of the History of Computing, 28*(4), 112–111.

Software item. (1990). In *IEEE standard glossary of software engineering terminology* (IEEE Std 610.12-1990). Retrieved from http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=159342&isnumber=4148

Software product. (1990). In *IEEE standard glossary of software engineering terminology* (IEEE Std 610.12-1990). Retrieved from http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=159342&isnumber=4148

Van de Vanter, M. L. (2002). The documentary structure of source code. *Information & Software Technology, 44*(13), 767–782.

Van der Hoeven, J., Lohman, B., & Verdegem, R. (2007). Emulation for digital preservation in practice: The results. *The International Journal of Digital Curation, 2*(2), 123–132.

Van der Hoeven, J. & Van Wijngaarden, H. (2005). *Modular emulation as a long-term preservation strategy for digital objects.* [PDF document]. Retrieved from http://www.iwaw.net/05/papers/iwaw05-hoeven.pdf

Zabolitzky, J. G. (2002). Preserving software: Why and how. *Iterations: An Interdisciplinary Journal of Software History, 1*, 1–8.